

Increasing Awareness of Delocalized Information to Facilitate API Usage

Uri Dekel

CMU-ISR-09-130

December 2009

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

James D. Herbsleb, Carnegie Mellon University (Chair)
Brad A. Myers, Carnegie Mellon University
Andre van der Hoek, University of California, Irvine
Gail C. Murphy, University of British Columbia

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2009 Uri Dekel

This research was sponsored by the National Science Foundation under grant nos. IIS-0414698 and IIS-0534656, and by generous donations from IBM Corporation, Accenture Corporation, the Alfred P. Sloan Foundation and the Software Industry Center at Carnegie Mellon University.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Carnegie Mellon University, IBM Corporation, Accenture Corporation, or the U.S. Government.

Keywords: Software Engineering, Collaborative Design, UML, Documentation, API, Javadoc

For Lee Ann and Elliott.

Abstract

Application Programming Interfaces (APIs) play a crucial role in modern software development, acting as abstract building blocks that allow engineers to focus on what makes their programs unique without having to constantly reinvent the wheel. While API authors convey how a method should be used via documentation, the text is delocalized from the source code that invokes that method, so its consumption requires additional effort from users of the API.

This dissertation presents the notion of “directives”, important clauses in the documentation of some methods that demand action or attention from their callers. It then demonstrates via a lab study that developers who are writing or examining code invoking these methods may fail to notice these clauses in the documentation text, or even to read the text at all. This lack of awareness precludes subjects from resolving bugs in our study and may cause serious faults in real world scenarios. This problem is particularly severe in polymorphic situations.

The thesis of this dissertation is that by overlaying visual cues on particular function calls in the source code, we can make developers aware of the presence of directives in the documentation of the call targets. Further, by listing them explicitly when this text is read, we can increase the prospects of the directives actually being consumed. These interventions would not significantly distract users.

To validate this thesis, we created eMoose, a plug-in for the Eclipse IDE that realizes these techniques. We loaded it with a set of directives that were found in a systematic survey of the Java standard library. In our lab study, the tool increased awareness of the directives without significantly distracting its users.

This work provides three primary contributions to software engineering. First, it reveals a weakness in the usability of API documentation that can lead to severe errors in the use of these APIs. Second, it demonstrates that decorating links is an effective and non-distracting way of making users aware of delocalized information. Third, it demonstrates that a similar problem of knowledge delocalization may occur in software design as a result of the representational choices made by designers.

Acknowledgments

This work is dedicated to the memory of my late grandparents, Arie and Bracha Levy, whose dreams of my future success encouraged me to meet the goals they once imagined, and to the memory of their mothers, Hannah Levy and Hannah Moskowitz, who perished in the Holocaust.

Thank you, Lee Ann, for being an unwavering source of support; I could not have done it without you. And thank you for giving me the most wonderful gift of all, our son Elliott. Thank you, Mom, for always being there for me when I needed you, despite the trials of distance and time.

Thank you, Jim, for not losing faith—or patience—even when things did not go according to plan. Thank you, Brad, for your guidance and attention to detail. Thank you, Gail and Andre, for your valuable advice, as well as your willingness to work with me.

Finally, I would like to acknowledge the generous financial support from IBM, Accenture, and the Software Industry Center at Carnegie Mellon University and its sponsors, especially the Alfred P. Sloan Foundation. This research was also supported by the National Science Foundation under Grants No. IIS-0414698 and IIS-0534656.

Contents

1	Introduction	1
1.1	Background	1
1.2	Representations and notation in collaborative design	2
1.2.1	Findings	2
1.3	The neighbor knowledge awareness problem	3
1.3.1	The neighbor knowledge awareness problem in design diagrams	3
1.3.2	The general neighbor knowledge awareness problem	4
1.3.3	The neighbor knowledge awareness problem in code	6
1.4	Approaches to the neighbor knowledge awareness problem in code	8
1.5	Thesis Statements	10
1.6	Dissertation Organization	10
2	Studying the Environment in Collaborative Software Design	11
2.1	Background and related work	11
2.1.1	Representing completed designs	12
2.1.2	Representing early designs	12
2.1.3	Design support tools	13
2.2	Study overview	13
2.2.1	Background	13
2.2.2	Goals	14
2.2.3	Settings	14
2.2.4	Methods, and subjects	15
2.3	Results - use of canvases	16
2.3.1	Canvas containment	16
2.3.2	Canvas types	16
2.3.3	Dealing with canvas size limits	18
2.3.4	Rescaling	18
2.3.5	Partitioning and merging canvases	20

2.3.6	External documents	23
2.3.7	Modifications and deletions	23
2.3.8	Rapid access	24
2.4	Results - Team Structure and focus	26
2.4.1	Maintaining individual focus on artifacts	28
2.4.2	Drawing the focus of others to an artifact	30
2.4.3	Inferring peer focus	31
2.5	Results - Notations	32
2.6	Discussion and tool implications - Use of canvases	32
2.6.1	Support for hierarchy of canvases	32
2.6.2	Knowledge preservation and canvas hierarchy changes	33
2.6.3	Organizing canvases	33
2.6.4	Canvase sizes, splitting, and merging	34
2.6.5	Private and public space	34
2.6.6	Canvas types	35
2.7	Discussion - Drawing activities and focus	35
2.7.1	Drawing activities and versions	35
2.7.2	Individual focus	35
2.7.3	Attracting peer focus to an artifact	36
2.8	Threats to validity	37
2.8.1	Impact of settings	37
3	Studying the Notations and Representations in Collaborative Software Design	39
3.1	Study overview	39
3.1.1	Background	39
3.1.2	Goals	40
3.1.3	Methods	41
3.2	Results: Alternate notations in individual artifacts	42
3.2.1	Use of UML and divergences	42
3.2.2	Adapting to evolution	43
3.2.3	Diverging from UML to work with custom levels of structure	51
3.2.4	Summary – Collaboration on individual artifacts	55
3.3	Results: Representing heterogeneous information	57
3.3.1	Using independent diagrams	57
3.3.2	Combining diagrams	58
3.3.3	Introducing peripheral information into diagrams	60

3.4	Results: Dependencies between diagrams	62
3.4.1	Diagram evolution across canvases	63
3.4.2	Noncontiguous artifact evolution	66
3.4.3	Dependencies on multiple diagrams	66
3.4.4	Coping with multiple artifacts	68
3.5	Discussion	70
3.5.1	Overview and contributions	70
3.5.2	Use of UML as an idiom	70
3.5.3	Divergence from UML	71
3.5.4	Order of evolution	73
3.5.5	Need for preserving context	74
4	Directives	77
4.1	What are directives?	77
4.1.1	The principle of least-surprise	77
4.1.2	Surprising clauses	78
4.1.3	Properties of directives	81
4.1.4	Property 1 - Directives require or imply client action	81
4.1.5	Property 2 - Directives should not be trivial, expected, or common	82
4.1.6	Property 3 - A directive should be relevant to most callers and scenarios	83
4.1.7	Property 4 - Significant consequences to lack of awareness	83
4.2	Related work	83
4.2.1	Documentation and comments	83
4.2.2	Studies of task comments	86
4.3	Introduction to the taxonomy of directives	87
4.4	Imperative directives - Restrictions	87
4.4.1	Deprecated methods	88
4.4.2	Methods made public due to language issues	88
4.4.3	Specific callers	91
4.4.4	Object state qualifications	93
4.4.5	Discourage use	94
4.5	Imperative directives - Alternatives	96
4.5.1	Deprecated methods	97
4.5.2	Specific callers	97
4.5.3	Use in special cases or purposes	99
4.5.4	Discouraged use	101

4.5.5	Better fit	103
4.6	Imperative directives - Protocols	104
4.6.1	Instructions about repeated invocations	105
4.6.2	Placement in sequence	107
4.6.3	Directives that depend on state	109
4.7	Imperative directives - Parameters	110
4.7.1	Restricting simple parameter value by contents	111
4.7.2	Restricting complex parameter values by contents	113
4.7.3	Restricting parameter value by type	114
4.7.4	Restricting parameter value by origin	115
4.7.5	Restricting parameter value by receiving object state	116
4.7.6	Warning of unexpected behaviors	116
4.7.7	Mutability	117
4.8	Imperative directives - Return values	118
4.8.1	Mutability	118
4.8.2	Cleanup	118
4.8.3	Limitations	119
4.8.4	Typing issues	120
4.9	Information directives - Side effects	121
4.9.1	Lazy creation	121
4.9.2	Cascading effects	122
4.9.3	Affecting multiple parts of the state	122
4.9.4	Destruction of existing state	123
4.9.5	Delayed side effects	124
4.10	Information directives - Limitations	124
4.10.1	Null effect	124
4.10.2	Accuracy limitations	125
4.10.3	Implementation limitations	126
4.10.4	Limited domain	127
4.10.5	No event triggering	127
4.10.6	Cannot support particular goals	128
4.11	Informative directives - Performance	129
4.11.1	Inherently expensive operations	129
4.11.2	Presenting more efficient alternatives	130
4.12	Information directives - Threading	131
4.12.1	Thread safety	132

4.12.2	Blocking	132
4.12.3	Performance	133
4.13	Conclusions	134
5	The eMoose tool	135
5.1	Knowledge space	136
5.1.1	Creating embedded knowledge items	136
5.1.2	Exporting and sharing knowledge items	137
5.1.3	Rating directives	138
5.2	Contextual model	138
5.3	Contextual presentation	140
5.4	Augmented <i>JavaDoc</i> hover	142
5.5	Related work	142
5.5.1	Improving reference documentation	143
5.5.2	Search tools for delocalized knowledge	144
5.5.3	Comment parsing and checking tools	144
6	Comparative Lab Study of eMoose	147
6.1	Intent and goals	147
6.2	Study design decisions	148
6.2.1	Decision to use multiple tasks	148
6.2.2	Decision to use limited set of directives	149
6.2.3	Decisions on codebase scale	150
6.2.4	Decision to use customized programs	151
6.2.5	Decision to use fixed codebases	151
6.2.6	Decision to use debugging tasks	152
6.2.7	Decisions on time limits and measurements of success	152
6.2.8	Decision to have two conditions	153
6.2.9	Choice of APIs	154
6.2.10	Decisions on amount of training	155
6.2.11	Decision not to use think-aloud or gaze tracking	155
6.2.12	Policy on answering questions	156
6.3	Study procedures and subjects	156
6.3.1	Subject recruitment	156
6.3.2	Subject characteristics	157
6.3.3	Preliminary procedures	158
6.3.4	Process for each task	159

6.3.5	Analysis technique	160
6.4	First debugging task, based on the JMS API	161
6.4.1	The JMS API	162
6.4.2	The directive for this task	162
6.4.3	Codebase and task	163
6.4.4	Results - Success rates	170
6.4.5	Results - Successful subjects in the control condition	171
6.4.6	Results - Unsuccessful subjects in the control condition	172
6.4.7	Results - All subjects in the experimental condition	173
6.4.8	Results - Contrasting results for the two conditions	175
6.4.9	Discussion - Explaining the difficulties of controls	177
6.4.10	Discussion - Explaining the impact of <i>eMoose</i>	177
6.5	Second debugging task, based on the JMS API	179
6.5.1	The directive for this task	179
6.5.2	Codebase and task	181
6.5.3	Results - Success rates	182
6.5.4	Results - Successful subjects in the control condition	182
6.5.5	Results - Unsuccessful subjects in the control condition	183
6.5.6	Results for successful subjects in the experimental condition	188
6.5.7	Discussion	189
6.6	First Swing Task	191
6.6.1	The directive for this task	191
6.6.2	Task description	193
6.6.3	Results - Success rates	196
6.6.4	Results - Control condition	201
6.6.5	Results - Experimental condition	203
6.6.6	Discussion	205
6.7	Second Swing Task	208
6.7.1	Directive for this task	209
6.7.2	Task Description	209
6.8	First Collections Task	209
6.8.1	API and directive for this task	211
6.8.2	Task Description	214
6.8.3	Results	215
6.8.4	Discussion	217
6.9	Second Collections Task	218

6.9.1	Directive for this task	218
6.9.2	Task description	220
6.9.3	Results	220
6.9.4	Discussion	222
6.10	Additional general findings	223
6.10.1	Repeated readings	223
6.10.2	Hovering over non-call elements	224
6.10.3	Use of the autocomplete mechanism	225
6.10.4	Use of mouse for highlighting	225
6.10.5	Ordering and learning effects	226
6.11	Exit Survey	226
6.11.1	Statements on “Marked calls”	226
6.11.2	The <i>JavaDoc</i> Hover	229
6.11.3	Polymorphism	232
6.11.4	General questions	233
6.12	Limitations	235
6.12.1	Artificial nature of tasks	236
6.12.2	Debugging nature of tasks	236
6.12.3	Subject background	237
6.12.4	Set of directives	237
7	Consistency in identifying directives	239
7.1	Introduction	239
7.2	Study design and materials	240
7.2.1	APIs used in the study	241
7.3	Results for methods used in the <i>eMoose</i> study	242
7.4	Consistency over the entire set	243
7.4.1	Number of subjects identifying tagging a directive in each method	244
7.4.2	Number of methods with directives identified by each subject	245
7.4.3	Reliability between subjects	245
7.5	Discussion and threats to validity	246
8	Conclusions and Future Work	251
8.1	Retrospection	251
8.2	Contributions	252
8.2.1	Improving the understanding of representation choices in software design	252
8.2.2	Identifying and demonstrating the neighbor knowledge awareness problem in code253	

8.2.3	Providing a technique to increase awareness of directives	254
8.3	Open questions and future research directions	254
8.3.1	Building directive collections	254
8.3.2	Filtering directives and decorations	256
8.3.3	Pushing other types of knowledge and working in other domains	258
A	Reproductions of booklets used in our studies	261
A.1	Materials from the <i>eMoose</i> lab study	262
A.1.1	Introduction booklet	262
A.1.2	Tasks booklet	267
A.1.3	Debriefing questionnaire	280
A.2	Materials from the directive tagging study	284
A.2.1	Introduction booklet	284

List of Figures

1.1	An example of delocalization between entities and associated assumptions	3
1.2	An example of network for the design diagram of Fig. 1.1	4
1.3	Sample subgraph for the general neighbor knowledge awareness problem	5
1.4	Sample graph for the neighbor knowledge awareness problem in source code	6
1.5	Javadocs for the <code>JLayeredPane.putLayer</code> method in JAVA SWING	7
1.6	Javadocs for the <code>Connection.setClientId</code> method in JAVA JMS	7
1.7	Code excerpt for creating a queue in JAVA JMS	7
1.8	Javadocs for the <code>QueueConnectionFactory.createQueueConnection</code> method in JAVA JMS	8
1.9	Code excerpt for creating a queue in JAVA JMS, with <i>eMoose</i> decorations	9
1.10	Javadocs for the <code>Connection.setClientId</code> method in JAVA JMS with <i>eMoose</i> additions	9
2.1	Photos showing canvases containing other canvases	17
2.2	Inappropriate use of specific canvas types	18
2.3	Photos showing how designers force too many sticky-notes into a limited container . . .	19
2.4	Forcing excessive contents into limited canvas space	19
2.5	Rescaling a diagram	20
2.6	Examples of canvas partitioning	21
2.7	Examples of canvas merging	22
2.8	Cleanup	23
2.9	A canvas is updated in bursts, implicitly creating versions	25
2.10	Canvases moved to secondary storage	26
2.11	Team formation in the full-day group	27
2.12	Team formation in the half-day group	28
2.13	Maintaining personal focus on one item	29
2.14	Maintaining personal focus on multiple items	29
2.15	Gesturing to another person about an item at close proximity	30
2.16	Inferring focus from gaze	31

2.17	Pointing at a remote artifact	36
3.1	First step in domain model diagram of team A	44
3.2	Second step in domain model diagram of team A	44
3.3	Adding methods to the domain model diagram of team A	45
3.4	Final domain model diagram of team A	46
3.5	First step in the solution model diagram by team E	47
3.6	Solution model diagram by team E becomes an entity-relation diagram	47
3.7	More relations are added to the solution model diagram by team E	48
3.8	Final form of the solution model diagram by team E	48
3.9	Early form of the functional domain model by team A	49
3.10	Final form of the functional domain model by team A	50
3.11	A use case diagram composed by team F	52
3.12	Connector notations in an object diagram of team F	52
3.13	Connector notations in an object diagram of team G	53
3.14	Creaing a class diagram with sticky-notes	54
3.15	Creating custom hierarchies with sticky notes	54
3.16	Alternative use case diagrams	56
3.17	Implementations steps included within in a class diagram	58
3.18	A class diagram embedded among use-case steps	59
3.19	Architecture presented in class diagram form	59
3.20	External elements in class diagrams by team A	60
3.21	Examples added using subclass notation in class diagram by team B	61
3.22	Sample record added to class diagram by team C	62
3.23	Team E recreating the data model	64
3.24	Revised data model by team E	65
3.25	Team E building an architectural model	67
3.26	Holding diagrams to increase locality	69
4.1	Documentation of method <code>String.replaceAll()</code> in JAVA 5	79
4.2	Documentation of method <code>String.replaceAll()</code> in JAVA 6	79
4.3	Documentation of method <code>ITextView.getVisibleRegion()</code>	80
4.4	Source code of Eclipse method <code>ProjectionViewer.getVisibleRegion()</code>	80
4.5	Javadocs of <code>setClientId</code> with enumerated clauses	82
5.1	Knowledge items that are automatically created from to-do comments	137
5.2	Sample class hierarchy	139

5.3	Relevancy tree example	139
5.4	Contrast level of method decorations adjusted by ratings	141
5.5	Example of JavaDoc hover for polymorphic code	143
6.1	Summary of tasks in the lab study	149
6.2	Text of online recruitment ad for lab study	157
6.3	Text of online recruitment ad for lab study	157
6.4	Template for timing data tables	161
6.5	Sun's illustration of the JMS programming model	162
6.6	Documentation of <code>QueueConnectionFactory.createQueueConnection()</code> in JMS	163
6.7	Source code listings for <code>SenderToQueue.java</code> - upper half	164
6.8	Source code listings for <code>SenderToQueue.java</code> - lower half	165
6.9	Source code listings for <code>SynchQueueReceiver.java</code> - upper half	166
6.10	Source code listings for <code>SynchQueueReceiver.java</code> - lower half with <i>eMoose</i> annotations	167
6.11	JavaDocs for the <code>MessageConsumer.receive()</code> method with <i>eMoose</i> directives	168
6.12	JavaDocs for the <code>QueueConnection.createQueueSession()</code> method	169
6.13	JavaDocs for the <code>QueueSession.createQueue</code> method with <i>eMoose</i> directives	169
6.14	JavaDocs for the <code>QueueSession.createReceiver</code> method with <i>eMoose</i> directives	169
6.15	Method mnemonics in the codebase for Task 1	170
6.16	JavaDocs for the <code>QueueConnectionFactory.createQueueConnection()</code> method with <i>eMoose</i> directives	175
6.17	Javadocs for the <code>Connection.setClientId</code> method in JAVA JMS (Reproduced with added emphasis)	180
6.18	Constructor for the <code>DurableSubscriber</code> inner class	181
6.19	Method mnemonics in the codebase for Task 1	182
6.20	Javadocs of <code>setClientId</code> with enumerated clauses	184
6.21	Log of visible regions of SCID documentation for subject 4	185
6.22	Log of visible regions of SCID documentation for subject 5	186
6.23	Log of visible regions of SCID documentation for subject 7	187
6.24	Log of visible regions of SCID documentation for subject 24	188
6.25	Timing data for successful subjects in experimental condition of Task 2	189
6.26	Javadocs for the <code>JLayeredPane.putLayer</code> method in SWING	192
6.27	Javadocs for the <code>JLayeredPane.setLayer</code> method in SWING	192
6.28	Outline area in the Web-based JavaDocs for the <code>JLayeredPane</code> class	193
6.29	Initial state of the <code>JLayeredPaneDemo</code> program	193
6.30	State of the <code>JLayeredPaneDemo</code> program after Duke is moved to layer 3	194
6.31	Error state of the <code>JLayeredPaneDemo</code> program after Duke is moved to layer 1	195

6.32	Source code of the <code>actionPerformed</code> method in the <code>JLayeredPaneDemo</code> class	195
6.33	Folded code outline for the <code>JLayeredPaneDemo</code> class	197
6.34	Source code of the constructor in the <code>JLayeredPaneDemo</code> class	198
6.35	Source code of the <code>createColoredLabel</code> and <code>createControlPanel</code> methods in the <code>JLayeredPaneDemo</code> class	199
6.36	Source code of the <code>createAndShowGUI</code> and <code>main</code> methods in the <code>JLayeredPaneDemo</code> class	200
6.37	Javadocs for the <code>JLayeredPane.setPosition</code> method with <i>eMoose</i> directives	203
6.38	Javadocs for the <code>Container.add</code> method in AWT with <i>eMoose</i> directives	208
6.39	Javadocs for the <code>Collection.containsAll</code> in standard JAVA library	211
6.40	Javadocs for the <code>Collection.retainAll</code> in standard JAVA library	211
6.41	Javadocs for the <code>Bag.containsAll</code> in <code>apache-collections</code>	213
6.42	Javadocs for the <code>Bag.retainAll</code> in <code>apache-collections</code>	213
6.43	Source code for first collections task	214
6.44	Javadocs for the <code>Map.put</code> in the standard JAVA library	218
6.45	Javadocs for the <code>BidiMap.put</code> in the standard JAVA library	219
6.46	Javadocs for the <code>DoubleOrderedMap.put</code> in the standard JAVA library	219
6.47	Source code for second collections task	221
7.1	Number of methods tagged as having directives by number of subjects (up to end of <code>JLayeredPane</code>)	244
7.2	Number of methods tagged as having directives by number of subjects (up to middle of <code>JComponent</code>)	245

List of Tables

3.1	Groups for which video footage was recorded	41
6.1	Assignment of lab study subjects into groups for each task	158
6.2	Timing data for successful controls in control condition of Task 1	171
6.3	Timing data for unsuccessful controls in control condition of Task 1	172
6.4	Timing data for successful subjects in experimental condition of Task 1	174
6.5	Timing data for unsuccessful subjects in experimental condition of Task 1	174
6.6	Comparison of certain behaviors across both conditions	176
6.7	Timing data for successful subjects in control condition of Task 2	183
6.8	Timing data for unsuccessful subjects in control condition of Task 2	183
6.9	Timing data for successful subjects in control condition of Swing 1	201
6.10	Timing data for unsuccessful subjects in control condition of Swing 1	202
6.11	Comparison of proportion of reading time in each category in first Swing task	204
6.12	Comparison of total reading time in each category in first Swing task	204
6.13	Timing data for successful subjects in experimental condition of Swing 1	205
6.14	Timing data for controls in first collections task	216
6.15	Timing data for the control condition of the second collections task	222
7.1	Cohen's Kappa values for methods up to end of <code>JLayeredPane</code>	246
7.2	Cohen's Kappa values for methods up to middle of <code>JComponent</code>	246
7.3	Cohen's Kappa values for methods up to end of <code>JComponent</code>	247
7.4	Presence of directives in JMS queue methods	248
7.5	Presence of directives in SWING methods (annotated by everyone)	249
7.6	Presence of directives in SWING methods (annotated by some subjects)	250

Chapter 1

Introduction

1.1 Background

Software development is an intensive form of *knowledge work* [46]. While performing design, development, and maintenance work, individual developers and teams continuously form or acquire and subsequently consume *knowledge of their activities* and *knowledge of the resulting artifacts*. Both types include *objective knowledge* - facts about the actions and artifacts, such as the order of editing actions and the names of the resulting code entities. Both types also include *subjective knowledge* - perceptions, opinions and feelings about these actions and artifacts, such as the underlying intentions, goals, and risks behind implementation strategies.

Since there is typically a temporal gap between the inception or encounter with knowledge and its first or final use, memory plays a critical role. Unfortunately, while the information is initially stored in the developer's *organic memory* [5], it is subject to degradation and potential loss. In addition, organic memory cannot directly be shared with others. Instead, software developers make use of persistent mediums such as whiteboards, papers, and computers to capture knowledge and subsequently access it or communicate it to their future selves or peers. For example, the relics of design sessions are typically documents, diagrams, and computerized models [16], while those of coding activity are typically the code itself, internal and external documentation, and peripheral records in collaboration-support tools.

In theory, persistent artifacts can capture all important knowledge, preserve it indefinitely, and eventually be used to recall that knowledge. In practice, however, this potential may not be fulfilled due to several classes of breakdowns: The knowledge may not be captured fully in the first place, it may become outdated over time, it may be difficult to interpret, or it may simply not come into awareness in situations where it can be useful. Such breakdowns can result in memory failures that can carry serious penalties for the developer or the organization.

This research is therefore concerned with the questions of how and what knowledge is captured and preserved, how it is subsequently recalled and consumed, and how we can assist software developers in doing all this in order to reduce memory breakdowns.

Naturally, the above questions are very general, and can represent an immense range of issues that depend on the specific development phase and style, as well as the kinds of knowledge involved. Accordingly, this dissertation is primarily concerned with exploring and addressing two specific scenarios: the use of diagrams in software design, and the use of method documentation in coding activities. In the first scenario, we focus on preservation and interpretation, while in the second we are focused on awareness of preserved knowledge. The knowledge obtained about these scenarios may be generalizable to other phases and activities.

This dissertation makes three major contributions to software engineering: First, it presents new findings on the use of notations and representations in software design. In doing so, it highlights a potential problem where designers focused on an entity in one diagram are not aware of knowledge and decisions previously associated with the same entity in other diagrams. Second, it presents empirical evidence that a similar problem exists in programming, where developers fail to become aware of important documented details about the methods they invoke. Third, it proposes a novel approach to this problem based on the presentation of cues on the calls, and presents empirical evidence for its potential effectiveness.

The rest of this introduction is organized as follows: I begin by briefly describing my study of representations and notations in collaborative design in Sec. 1.2. A major implication of this study involves the delocalization of knowledge, which I generalize into a “neighbor knowledge awareness” problem and apply to function documentation in code in Sec. 1.3. My approach to this problem is described in section 1.4. Finally, I present the thesis statement in Sec. 1.5, and the dissertation organization in Sec. 1.6.

1.2 Representations and notation in collaborative design

A good starting point for investigating the questions of knowledge capture and consumption is to address them in the domain of collaborative software design. Design is an early phase in virtually every software project and has tremendous impact on subsequent activities. It is also a highly visual activity that can produce informal or formal diagrams. By studying design, we can learn more about how engineers may capture and represent knowledge as soon as it is generated, and how they subsequently consume it. Design is often also collaborative, with multiple stakeholders presenting ideas, discussing, negotiating, etc. This gives us an opportunity to learn how knowledge is exchanged, preserved, and used by multiple individuals.

Prior studies [16] and casual observations demonstrated that designers generate a lot of diagrams. Many of these diagrams appear to be based on existing formalisms but with differences that are typically attributed to the early or handwritten nature of these diagrams. Accordingly, attempts to support such work focused on helping designers sketch [72, 14] and then complete and formalize these diagrams [23] or on bringing formal modeling functionality to entire teams [21, 89, 64]. However, little is known about these representation choices: How do designers pick a representation? How do they diverge from it? What functions does this divergence serve, what roles does it play in the preservation and use of knowledge, and what are its implications and side effects?

To answer these questions, I conducted a series of observational studies at the *OOPSLA DesignFest* event, where randomly-assigned teams of experienced designers spend several hours designing solutions to given problems. While such settings are clearly different from industrial scenarios, they allow us to observe the emergent choices that designers make when isolated from organizational practices and conventions.

1.2.1 Findings

The first major contribution of this dissertation is the set of findings from these studies and their implications, which will be discussed in subsequent chapters.

I observed that though designers borrow idioms and notations from formalisms like UML, they tend to combine them in unexpected ways and with varying semantics, create new notations on the fly, and change the level of structure or abstraction of existing diagrams. They also frequently work out designs by concurrently evolving multiple artifacts that can be at different physical locations.

While this freedom facilitates the creative process and allows its adaptation to the evolution of the the design, it negatively affects interpretability. Some improvised representations are inherently ambiguous, and many cannot be understood without additional contextual details from the time of their creation. These details may include the sequence of drawing actions, locations of diagrams, use of spoken notations, and perhaps most interestingly, passive references to artifacts such as those established by gestures and gaze.

I therefore argue that designers can be supported by offloading the need to preserve contextual details for future interpretation and transferring it to automated tools. Though such a solution for design is not attempted in the scope of this work, it inspired work on a similar solution for computer-based development activities, which is described as a secondary contribution of this dissertation.

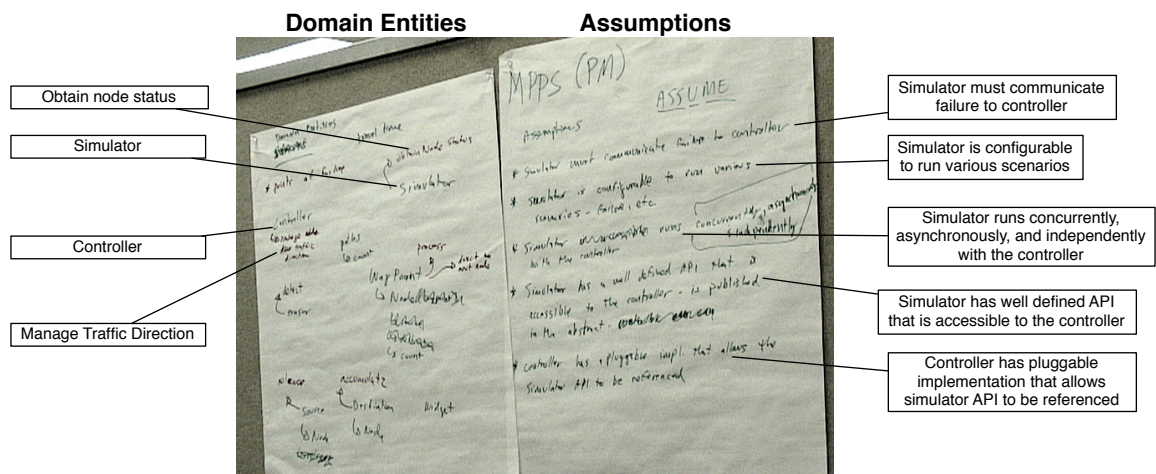


Figure 1.1: An example of delocalization between entities and associated assumptions

My observations also uncovered a more pressing problem that inspired most of this dissertation: Due to the concurrent use of multiple representations, information that is associated with an entity and which might be needed in the context of multiple artifacts is only captured in one location and may not be recalled when these other locations are examined. For example, the diagrams in Fig. 1.1 were created by a design team that was brainstorming an informal model of the domain while listing assumptions on a separate sheet. Certain entities, such as the simulator and controller, appear in the diagram and are also mentioned in the assumptions. A reader encountering these entities in the domain diagram may not be aware of the assumptions about them, especially if the papers are moved. Note that though these issues are particularly problematic for future readers, I observed designers missing decisions and assumptions made earlier during the same session.

1.3 The neighbor knowledge awareness problem

1.3.1 The neighbor knowledge awareness problem in design diagrams

The above problem inspired the core of this dissertation. It represents a class of problems in consuming data that had been effectively captured and preserved, which we term *neighbor knowledge awareness* problems.

This dissertation argues that design diagrams as described above can be treated as an abstract network, as illustrated in Fig. 1.2. The nodes of this network represent items in the diagrams and can be grouped by the diagrams what contains each of them. Nodes are either *instances of entities*, such as the

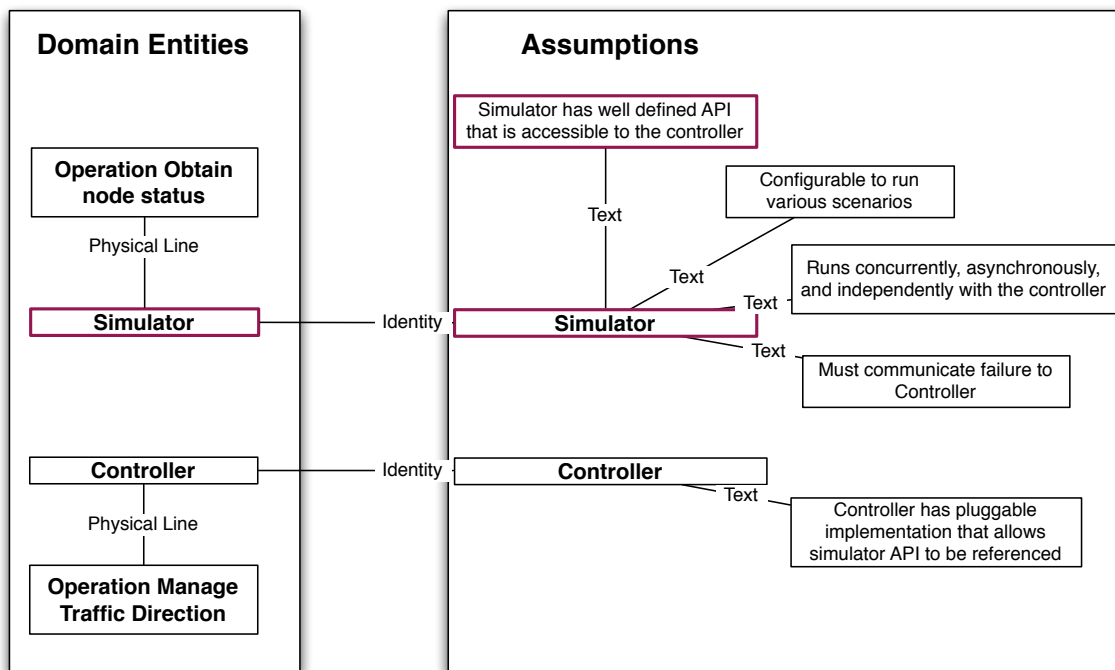


Figure 1.2: An example of network for the design diagram of Fig. 1.1

term `Simulator` in each of the diagrams, or *knowledge elements* such specific as assumptions or decisions. The network also consists of three types of relations: the identity between instances of the same entity in different artifacts, the physically-rendered connections between entities in the same artifact (e.g., arrows and lines), and the conceptual connections that are formed by the association of knowledge item to an entity.

Using this network, I state the *neighbor knowledge awareness problem in design artifacts* as follows: *Given the network implied by the set of artifacts in a design project, how can we help a user examining an entity in one artifact become aware of the knowledge associated with that entity in other diagrams?* For example, how does a visitor to the `Simulator` node in the domain entities canvas become aware of the existence of a simulator node in the assumption canvas, and through it become familiar with the assumption about having a well defined API?

1.3.2 The general neighbor knowledge awareness problem

The above problem can be generalized to general collections of documents and their corresponding networks, so that we can ask: How do we increase the prospects that a visitor to a node that represents an entity in one document become aware of the availability of relevant knowledge associated with that entity in another document?

We will use the term *artifacts* to represent an independent visual container for information and elements. Examples of artifacts include documents, canvases, source files, bug reports, emails, etc. We say that a *rendered connection* exists within an *artifact* if two elements are visually connected by a line or some other structured connection. We say that a *conceptual connection* exists within an *artifact* if there is a nonvisual but obvious connection between elements, such as that between the subject and object of a natural-language sentence.

We will use the term *entities* to refer to atomic concepts that permeate the software development

process and are manifested in many stages. In the earlier example, the simulator is an entity that appears early as a domain object and will likely also be represented in the source code, testing plan, and other locations. In this work we are only concerned with entities that have an independent physical representation, such as an appearance in the diagram or code as a visually distinct entity. Each such appearance is termed an *entity instance* and is bound to a specific artifact (a complete diagram or file). We say that there is an *identity relation* between instances of the same entity, even if the exact form (e.g. entity name vs. class name) is slightly different.

We use the term *knowledge elements* to represent atomic information that can be associated with an entity, such as a decision or assumption. In the scope of this work, we only consider knowledge elements that are physically captured (rendered) within an artifact.

Let us define our sets: Let A be a set of rendered *artifacts*. Let I be the set of entity instances in A , and K be the set of knowledge elements in A . The relation of containment in a specific artifact divides the sets I and K into equivalence classes, so that any two items in the same class are considered *localized* and any two in different classes are considered *delocalized*.

Let us now define the network: Let $G = \{V, E\}$ be an undirected graph. Let the set of vertices V consist of the set of *instance vertices* V_I and the set of *knowledge vertices* V_K , whose members correspond to all elements of I and K . Let the set of edges E consist of three sets: $E = E_I \cup E_P \cup E_C$. Let there be an *identity edge* E_I for every two instances of the same entity. Let there be a *rendered edge* E_P for every two vertices that represent elements within the same artifact that have a line, arrow, or similar visual connection between them. Let there be a *knowledge edge* E_K for every vertex representing a knowledge item that has a *conceptual* or *rendered* connection to an instance.

We also define a *knowledge awareness probability* $p_{KA}(v_i, v_k)$ as the probability that someone examining the entity represented by v_i becomes aware of knowledge item represented by v_k . Note that the two nodes do not need to be related or even in the same artifact.

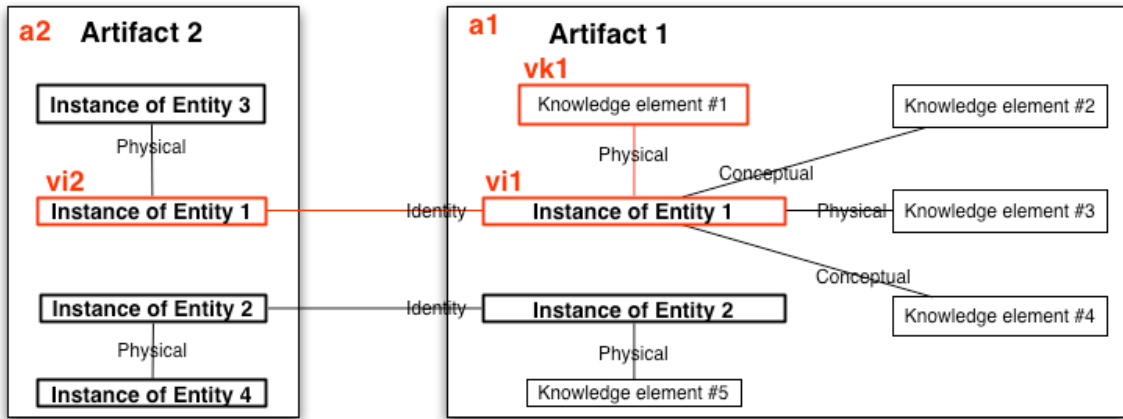


Figure 1.3: Sample subgraph for the general neighbor knowledge awareness problem

We now turn to formally define the general neighbor awareness problems. Fig. 1.3 presents a sample graph for illustration. Let $a_1, a_2 \in A$ be two distinct artifacts. Let $v_{i1} \in V_I$ represent an entity instance in a_1 , and $v_{i2} \in V_I$ represent an instance of the same entity in a_2 . Let $v_{k1} \in V_K$ represent a knowledge item in a_1 , and let $e_{k1} \in E_K$ represent a knowledge edge between v_{k1} and v_{i1} . Let $p_L = p_{KA}(v_{i1}, v_{k1})$ be the *localized probability* and $p_{NL} = p_{KA}(v_{i2}, v_{k1})$ be the *nonlocalized probability*. How do we increase p_{NL} ?

One premise behind this problem and potential solutions is that the nonlocalized probability is lower than the localized. In other words, we assume that examining an entity instance that is associated with a

knowledge element in the same artifact is more likely to result in awareness of this knowledge than when examining another instance of that entity in another artifact. Recall that our definition of knowledge elements is restricted to those relating to the entity rather than a specific instance. The validity of this premise and its implications will have to be evaluated in the context of specific instances of the problem.

In addition, we do not want to “solve this problem” for every knowledge element that is associated with some instance of an entity, as that can overwhelm users. Rather, we want to increase exposure to information that is more relevant within the current context, while not affecting or even decreasing exposure to less relevant information. Let us assume that there exists a *knowledge* relevance function $rel_K(v_i, v_k)$ that determines the relevance of a knowledge item to a *specific instance* of the entity. A more accurate goal for the knowledge awareness problem is that the proportion of increase of $p_{KA}(v_{i2}, v_{k1})$ be related to $rel_K(v_{i2}, v_{k1})$.

1.3.3 The neighbor knowledge awareness problem in code

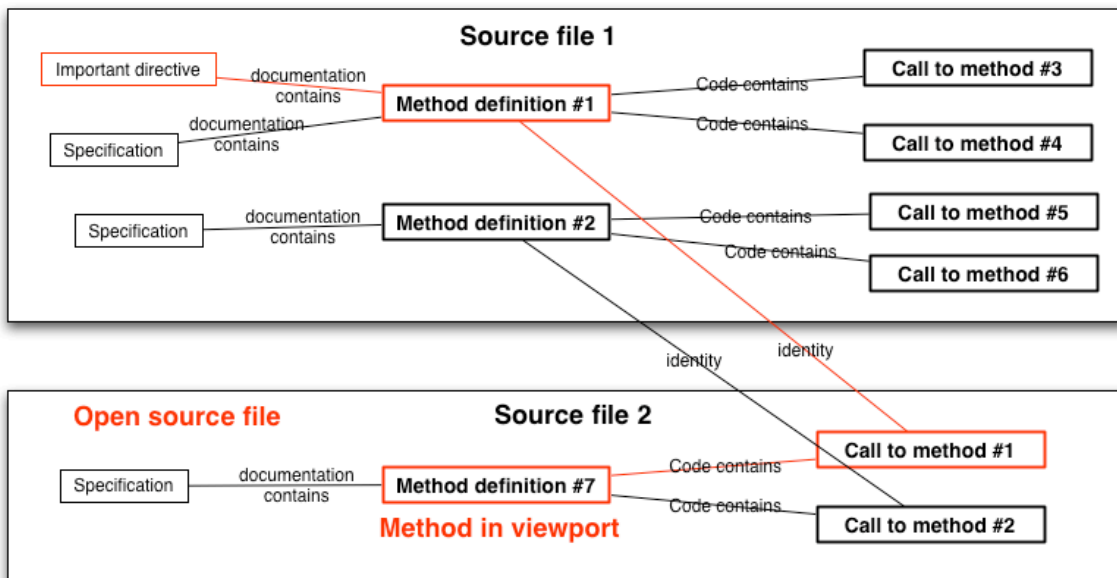


Figure 1.4: Sample graph for the neighbor knowledge awareness problem in source code

This dissertation is primarily concerned with the manifestations of the neighbor knowledge awareness problem in programming. Specifically, it addresses the problem of awareness of directives in invoked functions and especially in those defined in *Application Programming Interfaces* (APIs). In these settings, as illustrated in Fig. 1.4, we treat every source file as an artifact and every method identity as an entity. Every definition of the method is then considered an instance of its entity, and so is every call to that method. Certain clauses from the method’s documentation or metadata are considered to be our knowledge items and have a conceptual connection to the instance represented by the definition.

For example, consider a JAVA developer using the standard SWING toolkit for creating user interfaces, and specifically the `JLayeredPane` container, which manages and renders its children in different layers. Suppose that the developer is writing a function, and has references to the pane and an image within it but seeks to move the image to a different layer. As is often the case, the developer uses the auto-complete functionality for the pane until a likely match is found [80]. In this case, he is likely to come across `putLayer`, which seems like a good match, and adds the call. He may not examine the source code of that method or read its documentation, depicted in Fig. 1.5, and thus not become aware

```

void javax.swing.JLayeredPane.putLayer(JComponent c, int layer)
Sets the layer property on a JComponent. This method does not cause any side effects like setLayer() (painting, add/remove, etc). Normally you should use the instance method setLayer(), in order to get the desired side-effects (like repainting).

See Also:
  setLayer
Parameters:
  c the JComponent to move
  layer an int specifying the layer to move it to

```

Figure 1.5: Javadocs for the `JLayeredPane.putLayer` method in JAVA SWING

that the assignment does not cause a refresh and that a call to `setLayer` is needed instead. As a result, the user interface may not perform as expected, which presents a difficult debugging challenge.

Situations where the documentation of seemingly straightforward functions present some unexpected information are not rare. I have encountered many such situations in my own experience, and have received testimony of such problems from other developers. In a survey of several APIs (Chap. 4), I have identified many such potential pitfalls. Many examples will be presented in this dissertation.

In well-documented APIs, there is a detailed specification for every function that allows readers to learn everything about it. Our focus, however, is on clauses we term *directives* that only appear in the documentation of some methods. Directives can be explicit “do” or “don’t” instructions to which the caller must comply. Alternatively, they may convey information that clients may choose to act on, such as indicating that the function has unexpected side effects or performance issues, that it is not robust against certain situations, or that it is intended for use in specific scenarios.

```

void javax.jms.Connection.setClientID(String clientID) throws JMSEException
Sets the client identifier for this connection.

The preferred way to assign a JMS client's client identifier is for it to be configured in a client-specific ConnectionFactory object and transparently assigned to the Connection object it creates. Alternatively, a client can set a connection's client identifier using a provider-specific value. The facility to set a connection's client identifier explicitly is not a mechanism for overriding the identifier that has been administratively configured. It is provided for the case where no administratively specified identifier exists. If one does exist, an attempt to change it by setting it must throw an IllegalStateException. If a client sets the client identifier explicitly, it must do so immediately after it creates the connection and before any other action on the connection is taken. After this point, setting the client identifier is a programming error that should throw an IllegalStateException.

The purpose of the client identifier is to associate a connection and its objects with a state maintained on behalf of the client by a provider. The only such state identified by the JMS API is that required to support durable subscriptions.

If another connection with the same clientID is already running when this method is called, the JMS provider should detect the duplicate ID and throw an InvalidClientIDException.

Parameters:
  clientID the unique client identifier
Throws:
  JMSEException - if the JMS provider fails to set the client ID for this connection due to some internal error.
  InvalidClientIDException - if the JMS client specifies an invalid or duplicate client ID.
  IllegalStateException - if the JMS client attempts to set a connection's client ID at the wrong time or when it has been administratively configured.

```

Figure 1.6: Javadocs for the `Connection.setClientID` method in JAVA JMS

As illustrated by the highlighted line in Fig. 1.6, which presents the documentation of the `setClientID` method from the JAVA Messaging Service (JMS), such details can be “lost” deep within the detailed elaborate narrative that characterizes the documentation of many API functions.

```

queueConnectionFactory = new ActiveMQConnectionFactory(BROKER_ADDRESS);
queueConnection = queueConnectionFactory.createQueueConnection();
queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
queue = queueSession.createQueue(queueName);

```

Figure 1.7: Code excerpt for creating a queue in JAVA JMS

As illustrated by the earlier example from SWING, however, even if the directives are clearly visible in the documentation, there are no guarantees that this documentation would be read at the time of creating the call or at any subsequent reading of the invoking code. Such situations can be aggravated by the significant fan-out of many methods, which simply presents developers with too many operations

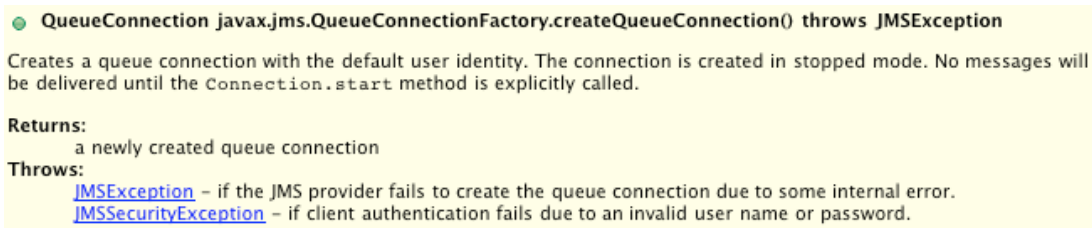


Figure 1.8: Javadocs for the `QueueConnectionFactory.createQueueConnection` method in JAVA JMS

to explore under everyday time constraints. For example, the code excerpt of Fig. 1.7 contains four calls involved in creating and initializing a queue in JMS. As shown by one of the studies in this dissertation, developers are more likely to explore calls like the complicated-looking call to setup a connection on the third line than the seemingly-trivial factory method in the second line. As a result, they may miss an important directive in the latter (Fig. 1.8) instructing them to use `start` to actually deliver messages.

These problems are even more likely in polymorphic situations where an overriding version of a method conveys directives that are not present in the overridden version. If a developer receives a reference to the supertype, he may not be aware of the existence of a possible dynamic subtype, of the overriding method, or of the different documentation. The *JavaDoc* hover offered by the IDE to investigate the documentation of invocation target will only present that of the overridden version from the static type. Though such situations often constitute a deprecated *conformance violation* [59], they are not uncommon.

We can therefore define the main problem that is specifically addressed by this dissertation as follows: **The neighbor knowledge awareness problem for source code:** In source code where “directives” may be associated or embedded in the documentation of a function, how do we increase the prospects that a visitor to a function that invokes it becomes aware of the directive?

The second major contribution of this dissertation (following our findings about design), is in demonstrating the severity and potential prevalence of this problem. I will present results from a lab study showing that developers do face difficulties in becoming aware of important directives in invocation targets. In other words, I will show that the probability of delocalized awareness, as described for the general problem, is far from 1 in the case of source code.

These findings highlight a serious communication breakdown between functionality providers and its consumers. One significant consequences for providers is that they must realize that merely documenting usage rules does not sufficiently increase the likelihood that their functions would be used correctly. I argue that it is necessary to come up with additional ways of increasing awareness to mitigate this breakdown. One possible solution, described next, is the third major contribution of this dissertation.

1.4 Approaches to the neighbor knowledge awareness problem in code

While the authors of a method’s documentation can structure the text to make directives more salient, such efforts have no impact if the developers who invoke this method do not read its documentation.

The third major contribution of this dissertation is in presenting a solution to the neighbor knowledge awareness problem in source code. My approach “pushes” the directives into the context of calling code. It does so by presenting cues on calls in the source code to make developers aware that the targets of these calls have associated directives. This approach may be generalizable to other instances of the neighbor awareness problem.

```

80     try {
81         queueConnectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");
82         queueConnection = queueConnectionFactory.createQueueConnection();
83         queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
84         queue = queueSession.createQueue(queueName);
85         queueConnection.start();

```

Figure 1.9: Code excerpt for creating a queue in JAVA JMS, with *eMoose* decorations

I implemented this approach for JAVA developers using the *Eclipse* IDE as the primary feature of our *eMoose* tool. My implementation decorates the call with a surrounding box and adds an icon on that line, as illustrated in Fig. 1.9 for the code fragment of Fig. 1.7. This should draw the user’s attention to the call and offer some indication of the availability of knowledge, while drawing attention away from other calls which do not contain directives.

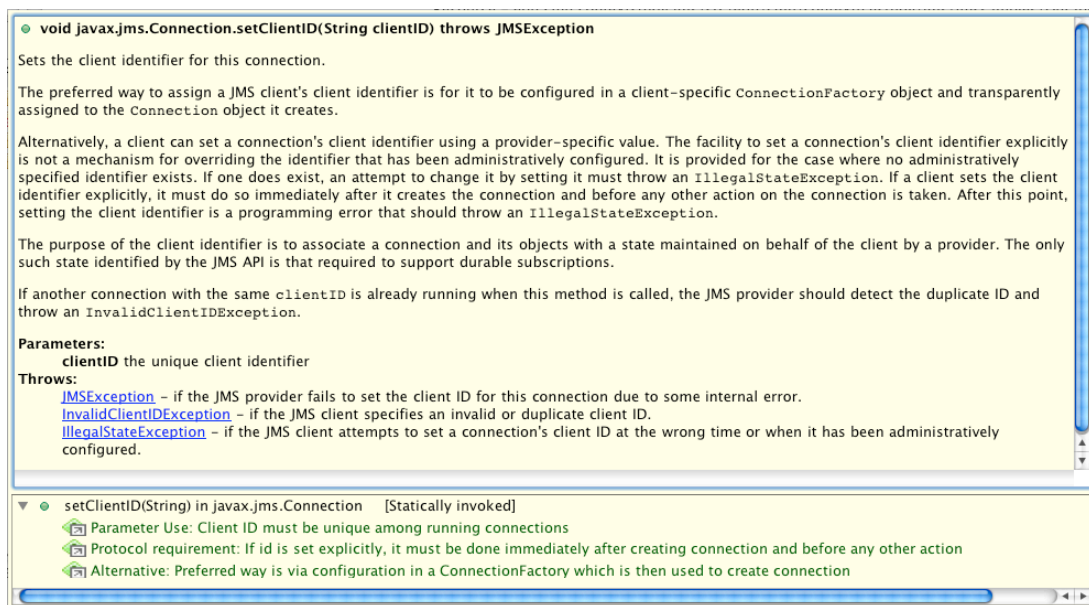


Figure 1.10: Javadocs for the `Connection.setClientID` method in JAVA JMS with *eMoose* additions

When the developer chooses to explore a decorated method, there is still a risk that the directives would be lost within the verbose text. To this end, my tool augments the *JavaDoc* hover in *Eclipse* with a lower pane that explicitly lists directives, as can be seen in Fig. 1.10. In the case of polymorphic code, the lower pane presents not only the directives corresponding to the documentation of the overridden version, presented in the upper pane, but also those associated with overriding versions in potential dynamic types.

My approach carries three major risks. First, there is no guarantee that the decorations would indeed attract readers to examine the documentation of the corresponding methods. Second, if the decorations are too effective, then developers may spend too much effort investigating documentations even if the knowledge they convey is not related to the developers’ current goals. This might render the use of the tool impractical and may eventually desensitize its users to the decorations. Third, the approach depends on the ability and willingness of API authors or users to tag directives in the documentation in an efficient and consistent manner.

A significant portion of this dissertation will attempt to address these concerns. It will present results from a lab study showing a significant impact for *eMoose* on certain types of tasks without an overbearing distraction due to the prevalence of decorations. A detailed analysis of records from the study reveals

much about the factors behind this effect. In addition, I present and analyze results from a lab study aimed at observing how different individuals identify directives in the same APIs.

It is important to emphasize the difference between this approach and the extensive work on automated conformance checkers [8,7,57]. There are many ongoing attempts to make *design by contract* [65] practical via formalisms for specifying usage contracts for functions and APIs, and static (or dynamic) tools for automatically checking conformance. While such tools can be extremely useful, they depend on the creation of formal specifications by function authors. The required investment may be too high, and some programmers may not have the skill for such accurate specifications. In addition, specifications cannot address the problem of making clients aware of certain details that do not constitute a usage violation, such as performance issues. Therefore, there is a need for an alternative mechanism for those directives that are not going to be captured formally. I believe that it is more likely that authors would be willing to tag existing text in natural-text documentation rather than create formal specifications, even if the benefit is limited to an increase in client awareness rather than automated conformance checking.

1.5 Thesis Statements

The primary focus of this dissertation is on the *neighbor knowledge awareness* problem for source code, which we repeat here as a reminder:

The neighbor knowledge awareness problem for source code: In source code where “directives” may be associated or embedded in the documentation of a function, how do we increase the prospects that a visitor to a function that invokes it becomes aware of the directive?

The statement of the thesis is thus:

Thesis Statement: When developers are examining code in the IDE, then by decorating calls to targets that have associated directives and making these salient when the target documentation is inspected, we can increase the likelihood that developers become aware of this information and avoid or mitigate errors and omissions without being significantly distracted.

1.6 Dissertation Organization

This dissertation is organized as follows:

Chapters 2 and 3 describe our two studies of collaborative software design. These studies helped identify the neighbor knowledge awareness problem for diagrams.

The dissertation is primarily concerned with the knowledge awareness problem for source code. To motivate the need for directive awareness, Chap. 4 presents a taxonomy of directive types, along with detailed examples of potential problems from our survey of APIs.

The *eMoose* tool, which implements our approach for “pushing directives”, is described in Chap. 5.

Chap. 6 presents our comparative lab study of directive awareness in source code, which shows that developers face difficulties in becoming aware of important directives, and that *eMoose* may increase this awareness.

To begin exploring whether collections of directives tagged by an API author or user could be useful to other users, Chap. 7 describes a small study which investigates how different individuals identify directives in the same API.

Finally, Chap. 8 describes our conclusions and avenues for further research.

Chapter 2

Studying the Environment in Collaborative Software Design

My doctoral research began with an exploration of collaborative software design work that takes place in physically collocated settings. The original goal was to identify opportunities for supporting such work with electronic tools, and in particular, for facilitating design in physically-distributed settings.

To this end, I conducted two studies of designers participating in design exercises at an academic conference. In the initial study, presented in this chapter based on [25], I gathered photographs and focused on characteristics of the physical work environment and activities, such as the use of drawing surfaces. In a subsequent study, presented in the next chapter and based on [28], I captured video records and focused on the notations and representations used in design.

While the focus of this dissertation has eventually shifted away from design towards the knowledge awareness problem in code, these studies have lead me to identify that problem and to recognize the importance of tracing temporal information links. Specifically, my findings in the first study emphasized the central role that canvases (the sheets on which designs are drawn) play in the design process. I saw evidence of the impact of their physical location on the team's work, and of the important of gaze in determining the point of focus. However, I also saw possible instances of the knowledge awareness problems.

The following two chapters present the two studies and the results identified at the time and also discuss findings relevant to my work on the knowledge awareness problem.

2.1 Background and related work

The design and architecture of a complex software system has significant implications for its functionality, cost, and reliability; a significant portion is therefore typically done upfront. While individual developers perform many design activities, the design of common modules, larger systems, and components is typically a highly collaborative process which involves many stakeholders. Design may be the most collaborative part of the development process [40].

Design tasks generally make heavy use of external representations to support problem-solving and collaboration and to capture the current state of the design [86]. This also applies to the domain of software design, in which diagrams go through a lifecycle from transient artifacts used to understand and come up with a design to archived documents serving communication and documentation purposes [16]. However, the choice of representation depends on many factors, including intended use, individual or col-

laborative design, organizational background, development paradigm, and the design problem specifics.

2.1.1 Representing completed designs

As with other design domains, well-defined and accepted representations and notations, rendered aesthetically and with precision, are important for clearly and unambiguously expressing finalized software designs [16, 23, 89]. In the object-oriented paradigm, the *Unified Modeling Language* [68] has gained widespread academic and industrial acceptance as a standard representation for completed designs that serve documentation or implementation planning purposes.

UML consists of 13 diagram types, which cover many of the structural, dynamic, and functional aspects of a system. These notably include *class diagrams* (CDs), *sequence diagrams* (SDs), and *use-case diagrams* (UCDs). UML offers precise notations for accurate specifications as well as extension mechanisms. This comprehensiveness enables compliant UML models to be used as early implementation artifacts, as blueprints for implementation, or as basis for automatic code generation [22]. A common criticism of UML is that it lacks fixed semantics and conventions [85, 12], potentially leading to defects due to misinterpretations. Other critics, focused on creative modeling, argue that UML is too strict and suggested that investment in creating complete models is not cost-effective [3, 67, 37].

2.1.2 Representing early designs

While much attention has focused on the representations used for documenting completed designs, little is known about how the representations are used in earlier design phases to come up with initial designs and if the same notations are applicable. One obvious difference about these phases is that design teams tend to sketch, often using physical mediums such as whiteboards. Sketching allows designers to effectively focus on the problem [3] and encourages experimentation with the design [14, 71, 58].

Most of our knowledge on early diagrams in industrial settings comes from Cherubini's study of diagram use at Microsoft [16]. He found that while many diagrams created in design collaborations were transient, some were immediately captured for subsequent use or were captured after being recreated in later collaborations. As their importance became clear, they iteratively became more organized and aesthetically pleasing and were often captured electronically. The representations used in these diagrams were a mix of informal visual conventions based primarily on box-and-arrow notation with only limited adoption of standards such as UML. However, the studied population used a variety of development paradigms and may not have been proficient in OOD or UML, and the factors leading to the representational choices were not studied.

The few observations that do focus on collaborative OOD work are primarily reported by researchers involved in constructing sketch-based design tools [23, 22, 14, 88, 89] and take place in specialized settings. In these observations, the end product is typically a design documented in syntactically correct UML, often in electronic form. Initially, freehand sketches are used to represent the problem domain, while limited UML diagrams, incomplete in content and syntax, are used for early designs of the solution [67]. Damm et al. [23, 22] further argued that these diagrams are then incrementally evolved into complete and compliant UML models with noncompliant elements removed. Overall, these studies give the impression that divergence from UML is accidental or unwanted.

Departures from UML or other standard notations might appear counterproductive when the goal is to produce an implementation-ready model or documentation. However, since representation choices have significant impact on design, constraining the early stages may have unexpected effects. Clearly, there are different potential uses and outcomes for diagrams created in design collaborations, and both organizational practices and prior knowledge of the intended use of a diagram likely have an impact on

the representational choices. Since the goal is to understand the needs of early design, I chose to conduct this study in settings which afford relative freedom from such potentially confounding constraints.

2.1.3 Design support tools

Most of the existing tool support for collaborative OOD is similar to that provided for individual designers, consisting mainly of functionality for creating complete models in UML. One approach aimed at distributed teams is to offer distributed groupware versions of the familiar single-user desktop CASE tools [11, 64]. In collocated collaborations, however, designers need large drawing surfaces [30], which were traditionally only available in the form of whiteboards or paper. As large display technologies became available, attempts were made to provide OOD specific support over electronic whiteboards [38, 14, 22]. These tools focused on the automated conversion of sketched shapes and handwriting, which are the natural mode of interaction with a whiteboard, into notational primitives and even UML models.

Of particular interest is Damm et al.'s *Knight* tool [22] which, based upon the observations described above, offers a guided mode that facilitates the evolution of these artifacts, through several levels of restriction, to complete UML models representing implementation-ready designs. Whenever artifacts violate UML specifications due to nonstandard notation or incompleteness, their tool forces the offending notations to be changed or removed. Other tools allow a layer of uninterpreted "freehand" sketches to coexist with a layer of structured UML.

The interactive nature of electronic whiteboards raises the question of how to best support collaborative design, and what, if any, support should be provided for rough, intermediate forms. Note that although these technologies received much attention, their availability is extremely limited, and most practitioners have no supporting tools when involved in collocated design collaborations. While some research focuses on supporting distributed design, such settings are plagued by many additional problems and are outside the scope of the current work.

2.2 Study overview

2.2.1 Background

Physically-distributed software development is practiced in many organizations due to globalization, outsourcing, increased telecommuting, and the open-source movement. While most research and tools are typically focused on coding activity, [74, 43, 24] or on asynchronous collaboration (e.g., via Wikis [2, 4]), there is significant need to allow a physical distribution of synchronous (real-time) design collaborations.

I argue that high-level software design typically necessitates some synchronous collaboration between multiple individuals. Even in large projects with rigorous use of documentation, mailing lists and other asynchronous forms of communication, there is usually a small group of individuals which conducts real-time collaboration throughout the lifetime of the project.

At present, these collaborations almost always take the form of a face-to-face *software design meeting* (SWDM), whose outputs are typically informal diagrams, notes, and verbal agreements. A physical distribution of such meetings involves challenges not present in business meetings or in asynchronous collaboration. Whereas in business meetings there is typically one speaker and even more typically a single point of visual focus (such as a presentation), visual activities in design can be concurrently carried out concurrently by multiple developers at different locations in space. Awareness and common grounding then become an important issue, as visual context is not always shared [51].

Design meetings are a forum where many critical decisions are made in real time. Many of these

decisions are informal mutual understandings, which participants later incorporate into their respective components or propagate to their teams. To ensure the preservation of these decisions, it is necessary to capture them in an unambiguous and persistent form. While decision preservation is a significant challenge for colocated meetings, it may be even greater in distributed meetings where each participant may have a significantly different perception of the experience.

2.2.2 Goals

Distributed collaborative design is not common. I therefore chose to perform an observational study of colocated design in an attempt to identify features that may be challenging to translate to distributed settings. With these findings, I hoped to find novel ways to support distributed design meetings. In this first study, my goal was to understand the “physical” facets of design collaborations and in particular the use of paper and physical space. To do so, I made use of photographic evidence gathered from several teams during the 2004 OOPSLA *DesignFest* event in Vancouver.

While my findings identified potential caveats in the transition to distributed settings, they also revealed many deficiencies and caveats in the current practices of colocated teams. These caveats may be indicative of potential knowledge preservation problems, and they caused me to subsequently shift the focus of my second study.

2.2.3 Settings

The *DesignFest* event

My studies of collaborative OOD took place at the the annual *DesignFest* events of the *ACM conference on Object Oriented Programming Systems Languages and Applications* (OOPSLA).

In this popular conference event, experienced designers select one of several given design problems and are randomly assigned to one of several teams that will work on that problem. They are given a short document describing the problem, constraints on the solution, and important use cases and scenarios. Depending on the session, teams then spend 3 to 6 hours coming up with an appropriate design to solve that problem. The stated goal of this event is to “learn more about design by doing it and to sharpen design skills by working on a real problem with others in the field” [29].

One notable characteristic of the *DesignFest* event is the relative freedom given to the teams. While teams are not expected to produce working systems, they are encouraged, both in the given documentation and verbally during the session, to prepare materials for presentation to others at the social event at the end of the conference and for a possible web archive. However, they are not given explicit requirements for the representations and quality of these materials. Similarly, the given documentation merely suggests a simple process outline, stepping from introductions to planning, discussion, sketching, and resolution of disagreements. It also encourages teams to elect a moderator and a recorder, but I have rarely seen these roles used in practice.

Note also that drawing activities at *DesignFest* take place primarily over physical mediums, which offer greater freedom than electronic tools [9]. The organizers provide all participating teams with at least one flipchart and several posterboards as well as notepads, sticky-notes, tacks, and pens. Powerstrips and wireless internet were available, but the few participants who used laptops did so mostly for unrelated activities. Only a few used CASE tools available on their laptop to try and digitize their teams’ sketches.

Reasons for selecting these settings

I chose to use the *DesignFest* settings for three reasons: First, participants do not have a history of working together and are collaborating outside an organizational context that prescribes design methods, notations, and processes. I believe this simplifies the interpretation of my results and enables generalization about common practices. Had I observed teams in an organizational context with a work history, it would be hard to disentangle behavior that is merely prescribed by organizational practices or shared team habits from natural behaviors that directly support the immediate task of collaborative design.

Second, since my focus is on the OO paradigm and the use of UML in particular, I wanted to minimize the risk that limited familiarity with these techniques would factor into the choice of representations. The OOPSLA conference attracts experienced designers well-versed in these techniques, which adds validity to the findings. Third, since design problems in *DesignFest* are based on real-world projects, I can observe multiple teams working on the same problem and reproduce the nonproprietary designs. Although I could probably find other settings that are better in any one of these considerations, *DesignFest* seems to score adequately on all three, and, in my opinion, deserves the attention of more researchers.

2.2.4 Methods, and subjects

My initial study aimed to understand the “physical” facets of design collaborations, and in particular the use of paper and physical space. It made use of photographic evidence gathered from two teams during the 2004 *DesignFest* event in Vancouver. The first team in the study was doing a “half-day” session, while the other was doing a “full-day” session on a different conference day. As I did not have access to a video camera, I tried to capture as many snapshots as possible to allow the process to be reconstructed. In particular, I focused on gatherings of subjects, trying to preserve a record of who was involved, in what locations, and what artifacts they were touching or gesturing at. Frequently I also took close-up photographs of the whiteboards and of the diagrams and sticky notes used by the subjects, both when posted in the public spaces and while they were still working on them. I later examined all these photos on a timeline in a digital photo organization software.

The half-day group consisted of four developers and two educators, while the full-day group consisted of five developers. The developers all had at least 4 years of experience as software engineers or architects in the industry; the educators had served as teaching faculty members for more than ten years. In accordance with the *DesignFest* rules, each group appointed a “moderator” and a “recorder”. Only the half-day group made use of the moderator role, and neither group used its recorder.

Both teams were working on the “case management problem”. The system is a general object-oriented framework for use in developing case-management applications for industries like insurance, finance and healthcare.

Both sessions were held in a large hotel banquet hall, and attendees were seated around large circular tables. Each team was provided with a single flipchart and a few posterboards for hanging materials; hotel notepads were provided to each attendee. The moderator of the half-day session also brought stacks of stickynotes for the use of his group. As we shall see, physical settings and resources had a significant impact on how teams performed their task

Even though both teams worked on the same problem, each team used different processes, interaction styles, tools, and notations. At the guidance of its moderator, members of the half-day group first individually brainstormed entity ideas on sticky notes, proceeded to arrange them on the poster board, and then worked as a team on creating class diagrams. The full-day group, on the other hand, brainstormed use-cases as a team. It divided them into two groups, separating to work on them, and then

regathered to discuss their results. With the session drawing to an end, they split again to create complete and finalized versions of their initial diagrams.

2.3 Results - use of canvases

Software design activities involve and produce numerous and complex visual artifacts [16], which can range from simple textual entities to longer documents and to complex diagrams. We refer to the surfaces on which visual artifacts are rendered as “canvases”, after the metaphor used in many UI toolkits. In *DesignFest* events, designers work primarily with pen-and-paper, so canvases range from large flipchart sheets to standard notebook paper and even sticky notes.

2.3.1 Canvas containment

Physical canvases offer flexibility that was not possible in computer-based graphic work on early computers [13, p. 195]. Even today, most computer-based modeling tools use a single simple canvas as a metaphor for the drawing canvas.

One advantage of physical canvases is that they can easily be moved about and placed inside other canvases, creating a containment hierarchy. For example, a sticky note can be placed on a sheet of paper (Fig. 2.1(a)), which can be posted on a posterboard (Fig.2.1(b)). As we shall see, designers interact heavily with this hierarchy of canvases, suggesting that this is an important feature for computer-based design support software.

An important property of this containment hierarchy is that it is fluid, as canvases are moved around over time. The same canvas can not only contain different “children” at different times, but may also itself be contained by different “parents” at different times. This may present challenges to the preservation of knowledge and the interpretation as artifacts, as canvases may end up far from the context in which they were originally generated.

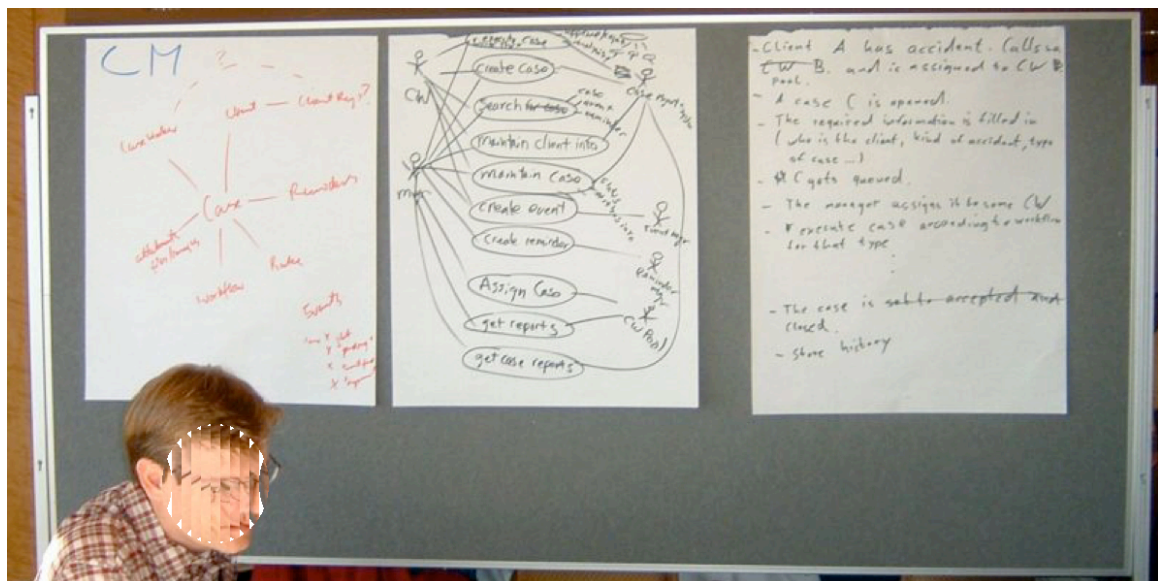
2.3.2 Canvas types

While most canvases serve merely as containers for actual content and for other canvases, it appears that certain types of canvases have special symbolic meanings or properties that hint at their expected use. For example, sticky notes are small and prompt people to fill them with concise and limited text, typically limited to a word or two. Their color helps distinguish them from notepads and reminds users of the adhesive material which hints that they should be attached to other canvases. Similarly, large flipchart sheets “invite” complex diagram or embedded smaller canvases; because of their size, we recognize that they are intended for hanging on a board. The rough surface of a posterboard indicates that it should not be drawn upon, and that sticky notes should not be attached to it. Rather, it suggests the use of pins to attach other materials, such as paper sheets from the flipchart.

Note that even though the type of a canvas suggests how it should be used, it might occasionally be used differently. For example, Fig. 2.2(a) shows a member of the half-day team using a small stickynote for a complex diagram. Fig. 2.2(b), on the other hand, shows a member of the full-day team using notepaper for brainstorming in the way that one would use a sticky: note how only a small portion of each sheet is used. Such misuse can result from a sudden change in plans or simply because a more appropriate canvas is unavailable. The full-day team, for example, did not have stickynotes or any smaller notepads.

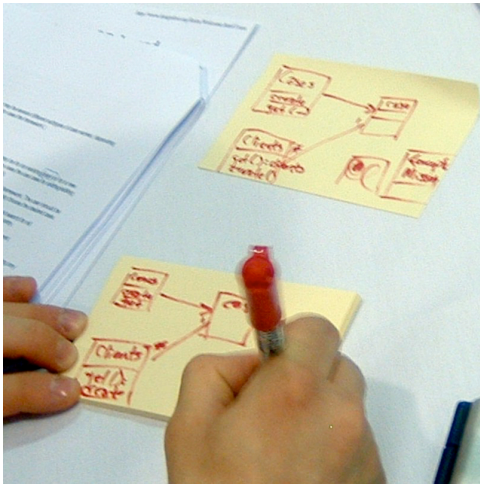


(a) Sticky notes attached to a flipchart sheet



(b) Flipchart sheets attached to a posterboard

Figure 2.1: Photos showing canvases containing other canvases



(a) Using sticky notes as notepad sheets



(b) Using notepad sheets as sticky notes

Figure 2.2: Inappropriate use of specific canvas types

An inappropriate canvas type selection can be confusing and wasteful and may demand a recreation of the canvas. This could be a benefit to design-support software in which types could be changed.

We note though, that in some cases the misuse of a canvas type could be intentional, perhaps owing to individual style or to a limitation of the expected content type. For example, while a sticky note typically contains a single idea in text form, one may express it as a diagram.

2.3.3 Dealing with canvas size limits

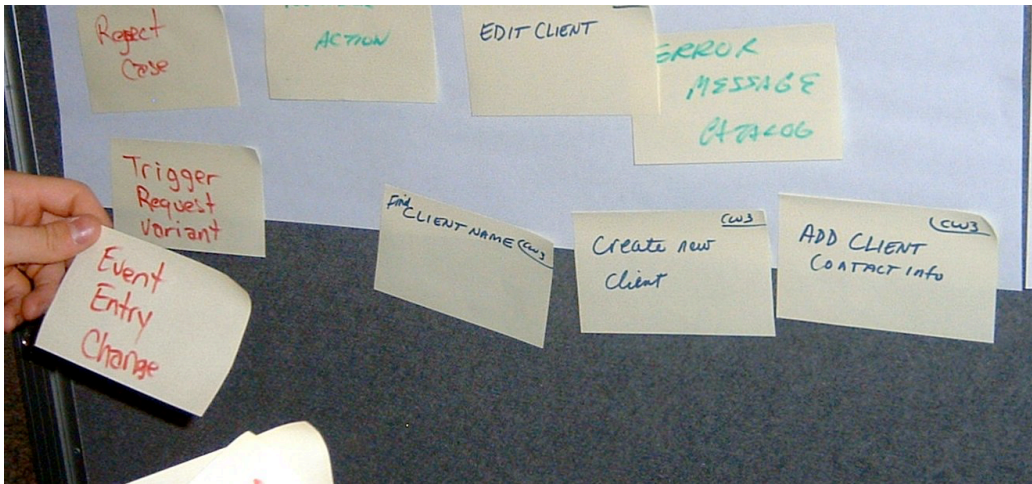
A major drawback of physical canvases is that their dimensions are fixed, limiting the amount, style, and layout of contents that they can accommodate. This forces designers to plan ahead, requires effort to be spent on organization, and reduces the flexibility of the canvas use.

When space in a particular canvas was imminently running out, some designers tried to force excessive amounts of contents into the available area. For example, Figs. 2.3(a)-2.3(c) show how additional sticky-notes had been forced into a limited space by hanging them outside the canvas boundaries or by letting them invade into other logical regions. Figs. 2.4(a) and 2.4(b) show complex diagrams squeezed into the available space, rendering them confusing and unreadable. I suspect that this behavior significantly affects the choice of content that designers make, and that they may omit details in such situations.

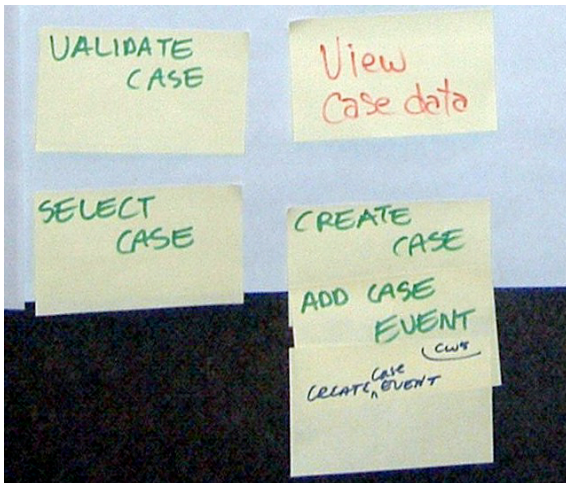
2.3.4 Rescaling

Designers often appeared to choose the kind of canvas to use based on the expected size of the contents and the intended audience. For example, small notepads were convenient for individual notes but difficult to share with others, whereas larger sheets are easier to share but are bulky and space consuming. As a result, when a previously personal artifact had to be shared or collaborated on, it was often first reproduced on a larger scale.

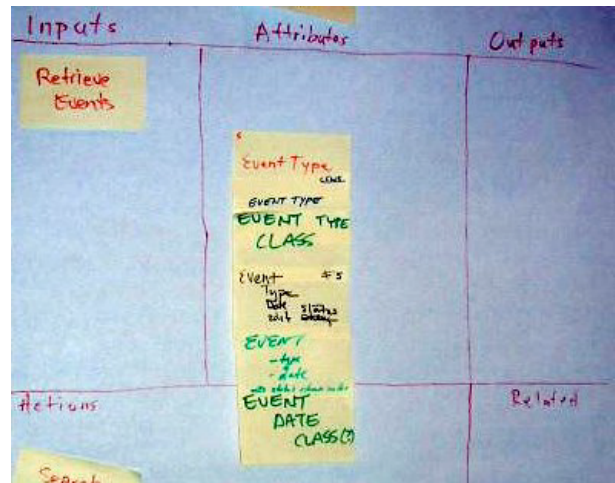
While this reproduction is tedious and could have been eliminated with an electronic drawing medium, the rescaling process has an important side effect: Recreating the artifact into a larger form is often not simply a matter of duplicating the original but rather is often an opportunity to rethink, improve, and polish.



(a)

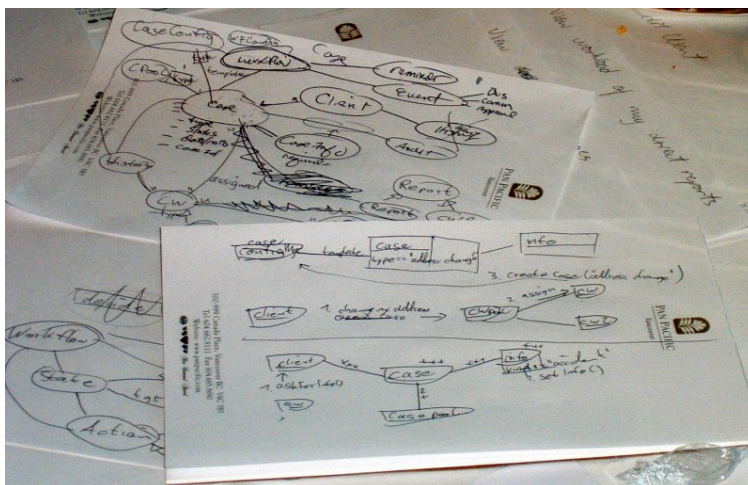


(b)



(c)

Figure 2.3: Photos showing how designers force too many sticky-notes into a limited container



(a)



(b)

Figure 2.4: Forcing excessive contents into limited canvas space



(a) The rescaling of a diagram is an opportunity to clean it up



(b) Rescaling is often a group activity

Figure 2.5: Rescaling a diagram

For example, Fig. 2.5(a) shows a designer copying a complex object diagram from a cluttered note into a large sheet of paper. In doing so, he is elaborating each entity, listing its properties and drawing it in straight lines instead of freehand curves. Meanwhile, others are examining both drawings, offering suggestions and approving each element as it is copied (Fig. 2.5(b)).

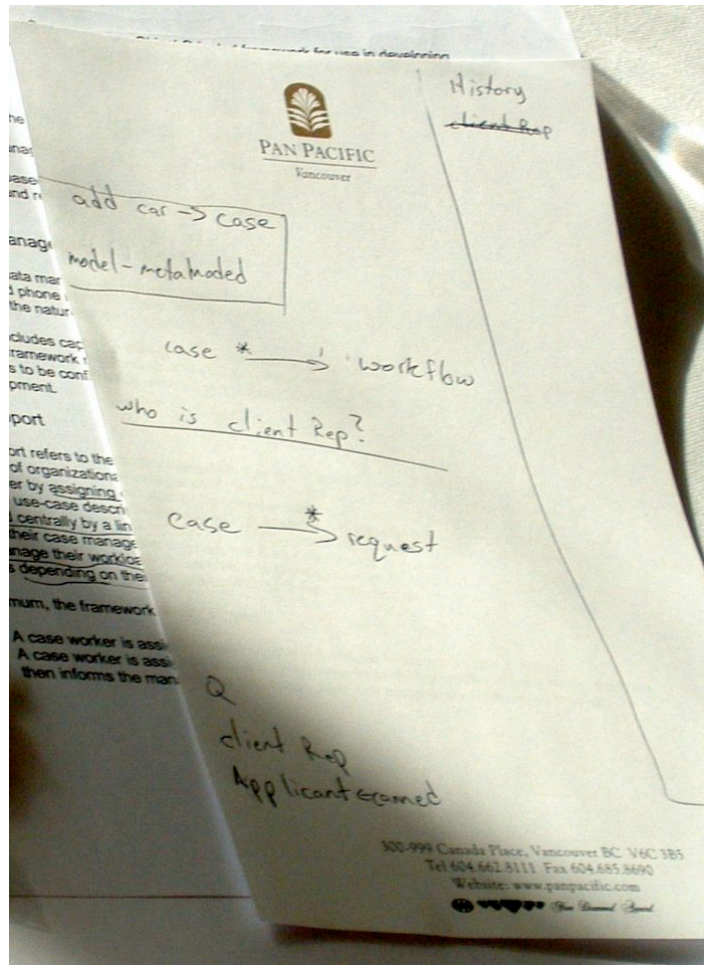
2.3.5 Partitioning and merging canvases

A canvas representing a single entity is often partitioned in order to allow properties and other information to be organized into categories. For example, Fig. 2.3(c) shows the partition of a canvas representing the *event* abstraction of the case management system; no sub-canvas was intended to stand on its own. CRC cards [6] are another example of this behavior. Occasionally, a certain partition is only relevant in the context of another, as in Fig. 2.6(a), which shows an area cordoned off for notes.

Sometimes, a partitioned canvas contains independent but somewhat related artifacts. Such a partition can serve to contrast design alternatives, as in Fig. 2.6(b). In other cases, however, it seems that this partitioning is an alternative to artifact organizations that are difficult when to achieve with multiple physical canvases. For example, in the lower sheet of Fig. 2.4(a), the two diagrams are only weakly related (by involving the *case abstraction*). While it make sense to separate them into two canvases, it would be difficult to maintain the close connection between them. An electronic tool, on the other hand, could achieve this with linking.

Note, though, that unrelated artifacts might occasionally be placed on the same canvas to avoid wasting resources and space and to reduce clutter.

In some cases, designers may want to merge multiple canvases into one. For example, participants in the half-day session were limited to small fixed-sized sheets upon which sticky-notes could be laid out. When they ran out of space on the first sheet (Fig. 2.7(a)), they numbered it and started using a blank sheet (Fig. 2.7(b)). When it ran out as well, they used a third sheet, and eventually placed the three of them in sequence on a posterboard (Fig. 2.7(c)), creating a single entity. Merging canvases with physical paper is tricky, but electronic tools could compensate.



(a) Keeping short notes



(b) Contrasting design alternatives

Figure 2.6: Examples of canvas partitioning



(a) Part 1



(b) Part 2



(c) Merged

Figure 2.7: Examples of canvas merging

2.3.6 External documents

Although content was usually rendered on a previously blank canvas, there were cases when an existing pre-printed document was further annotated or drawn upon. For example, most participants made initial annotations on the requirements document provided by the organizers. In some cases they even drew small diagrams next to specific requirements rather than draw them on separate sheets of papers where they would be more difficult to access associatively.

2.3.7 Modifications and deletions

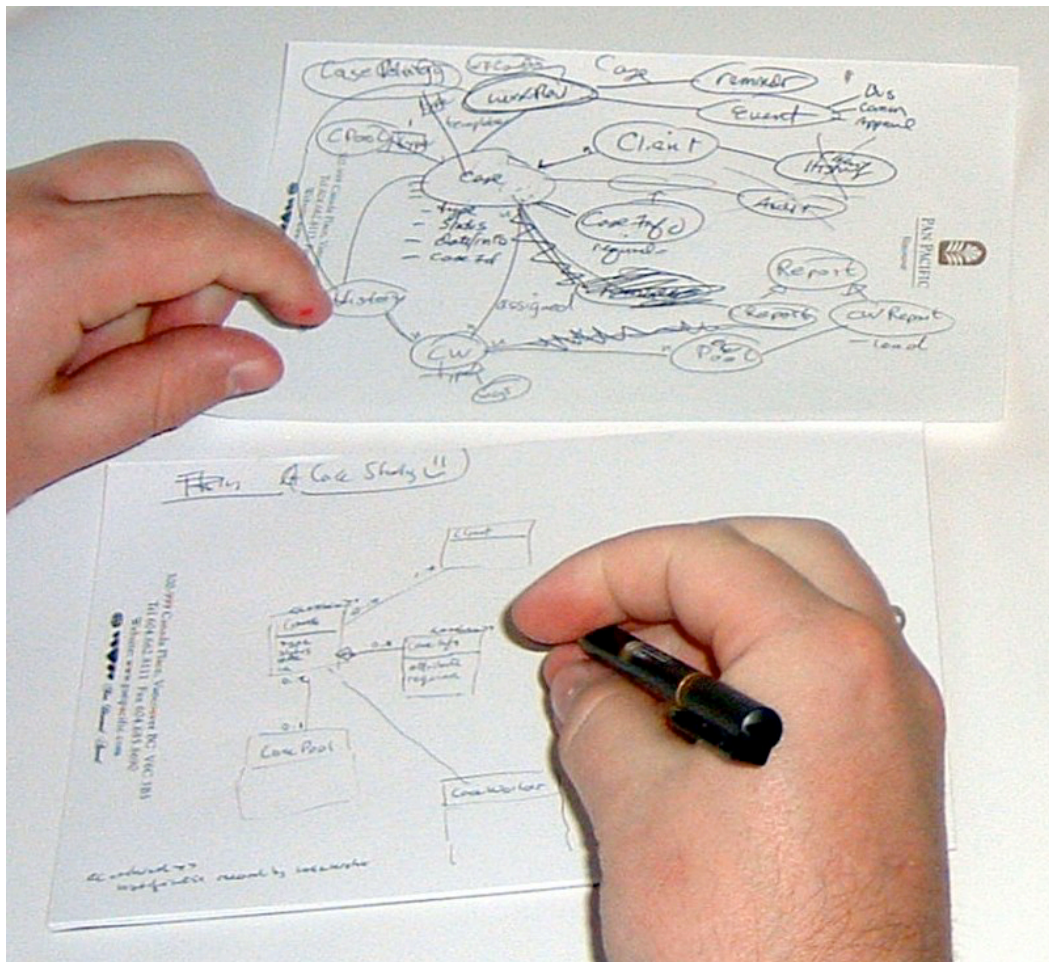


Figure 2.8: Cleanup

Physical canvases differ in how their contents can be changed or erased. Unlike dry-erase boards, contents on paper are the most difficult to change. As we can see in the upper diagram of Fig. 2.8, contents sketched in ink can only be removed in an untidy way that does not fully recover the lost space. As more and more items are erased, the document becomes too cluttered. Eventually, it becomes unfit for preservation or presentation and must be redrawn, although this is an opportunity for rethinking.

Another problem with physical mediums is that changes to one element often require manual changes to other elements. For example, if a specific element is “surgically” removed, all overlapping or connecting objects, such as fragments of connectors and text, will also have to be redrawn or removed. This further encourages a redraw. I suspect that the apprehension of the visual impact and the expense of

the potential redraw may sometimes discourage developers from making modifications, leading them to leave imperfections in the design.

Versioning and rollbacks

One of the interesting phenomena I encountered is that canvases are typically used and updated in discrete bursts of activity rather than continuously. An individual working on a diagram often does so in cycles, alternating between thinking or consulting other material and drawing or making modifications to the canvas. These cycles are much longer when several people are working together, partially because of the additional time required for discussion. Larger teams often abandoned canvases for a while, working on another before coming back.

For example, Figures 2.9(a) to 2.9(d) show a progression of changes to an artifact. Fig. 2.9(b) shows how the original rectangle is expanded into a diagram in a continuous work interrupted by frequent discussions and modifications. In the transition to Fig. 2.9(c), a change in ink color indicates that the new artifact at the bottom has been added as a separate burst. Finally, we see in Fig. 2.9(d) that a comment in blue has been added to the existing list of comments in black. The implication from this mode of work is that periods of quiescence with respect to a particular canvas or artifact may implicitly represent a version.

2.3.8 Rapid access

Even though the abundance of artifacts created during SWDMs prevents more than a few from being in focus at any given time, a certain portion could still be physically positioned for rapid access. For example, every *DesignFest* team was furnished with two or three poster boards, each capable of displaying up to three sheets of paper from the flipchart. Once a specific sheet was located on the board, its contents could be read effortlessly. To locate a particular canvas or artifact, however, people employ a variety of techniques.

The most efficient way to locate an item appears to be based on memorizing and recalling its spatial location. This location could be relative to the environment (e.g., “next to the window”), or relative to other artifacts (e.g., “on the sheet right below the class diagram”). The problem is that memory is unreliable, and that the canvas or the reference point could be moved.

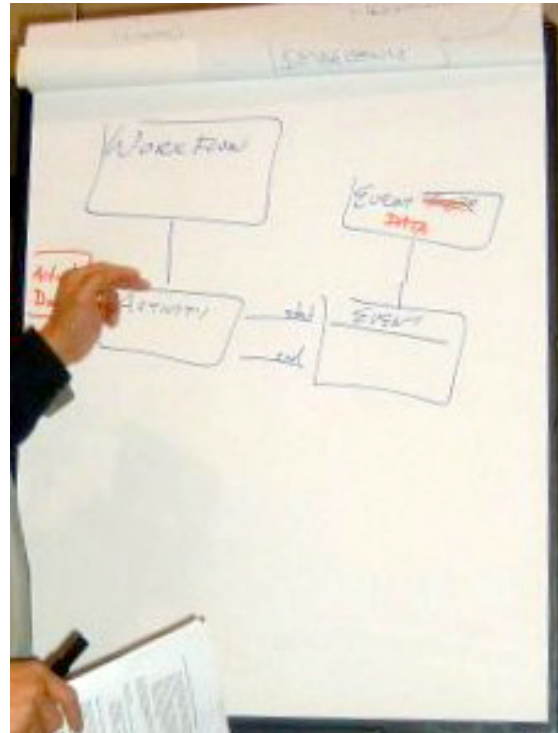
If many artifacts are graphical rather than textual in nature, their shapes can be identified in a rapid sweep over the entire workspace. A quick glance is often enough to locate a UML sequence diagram or a “spider shaped diagram”. The potential of this method appears to decrease as the number of artifacts increase and the visual differences between them decrease. This is particularly a problem if there are multiple versions of the same canvas.

Note that some teams titled their canvases. Titles may be an effective way to spot a canvas but they require some reading effort and must be distinct.

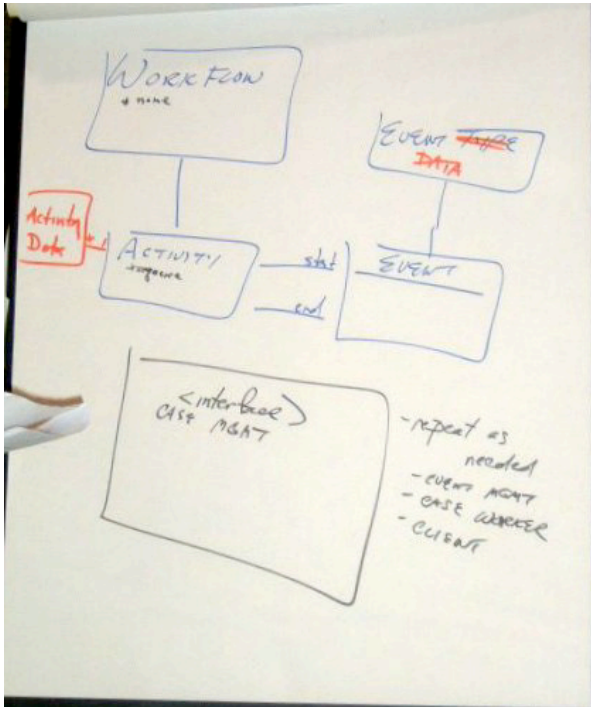
Unfortunately, the space available for rapid access in the manner described above is limited by the available physical space and the effective access speed. Important canvases, such as to-do lists, will not be removed. Others, however, may be removed in a least-recently-used manner to make space for others. In the physical world, it is difficult to organize and find space for the removed sheets. Many teams, for example, placed these sheets on the floor (Fig., 2.10(a)), where they were more difficult to access later (Fig. 2.10(b)). As we can see in Fig. 2.10(c), participants individually faced similar problems when maintaining large numbers of canvases.



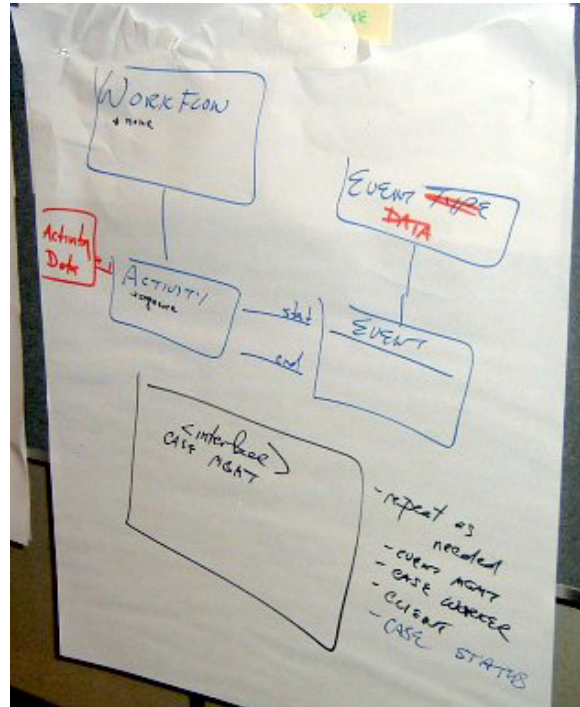
(a) Initial diagram in blue



(b) Changes in red



(c) New elements in black



(d) New comment in blue

Figure 2.9: A canvas is updated in bursts, implicitly creating versions

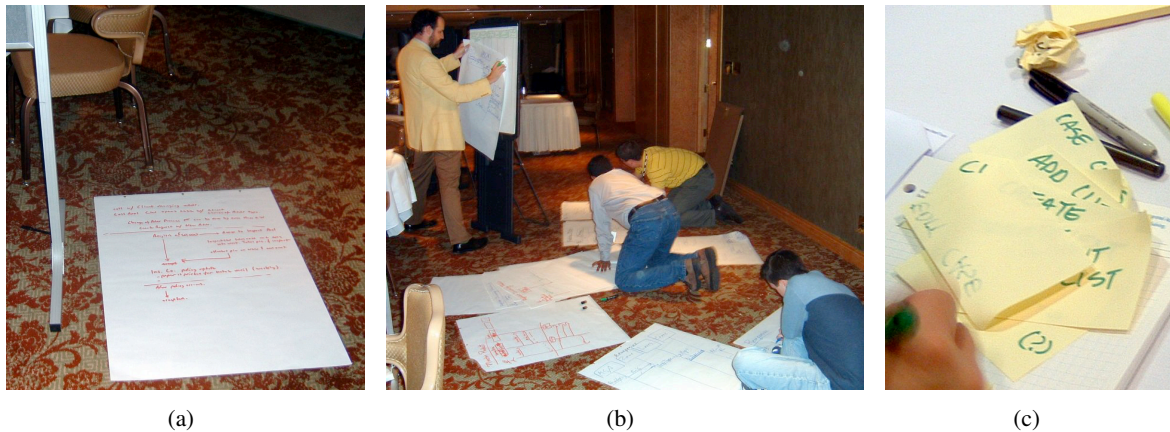


Figure 2.10: Canvases moved to secondary storage

2.4 Results - Team Structure and focus

My results so far have focused on canvases and their manipulation. We now turn to the collaborative facet of SWDMs. We begin by exploring the designers' division into subteams and then describe how they maintain their focus on nearby and remote artifacts and how they draw the focus of others. We then discuss the problem of locating the current focus of the group after diverging from it.

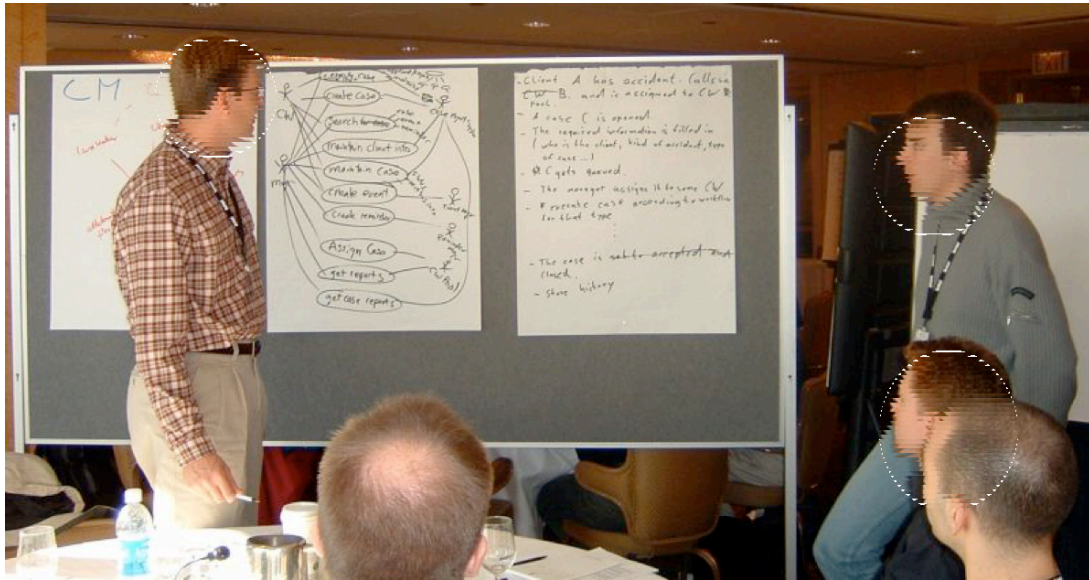
Division into teams

In the course of the DesignFest sessions, team members repeatedly split into subteams, regathered, and split into different subteams; occasionally they even worked individually. Much of this behavior was ad-hoc: Rather than having clearly defined roles and memberships throughout the session, subteams simply coalesced around specific tasks and artifacts and later dissipated without ceremony. As a result, every designer may have a different and partial understanding of the design, and may lack awareness of design decisions made by individuals outside the subgroup. This affects after-the-fact grounding, as certain concepts may be fully familiar to some individuals and utterly unfamiliar to others.

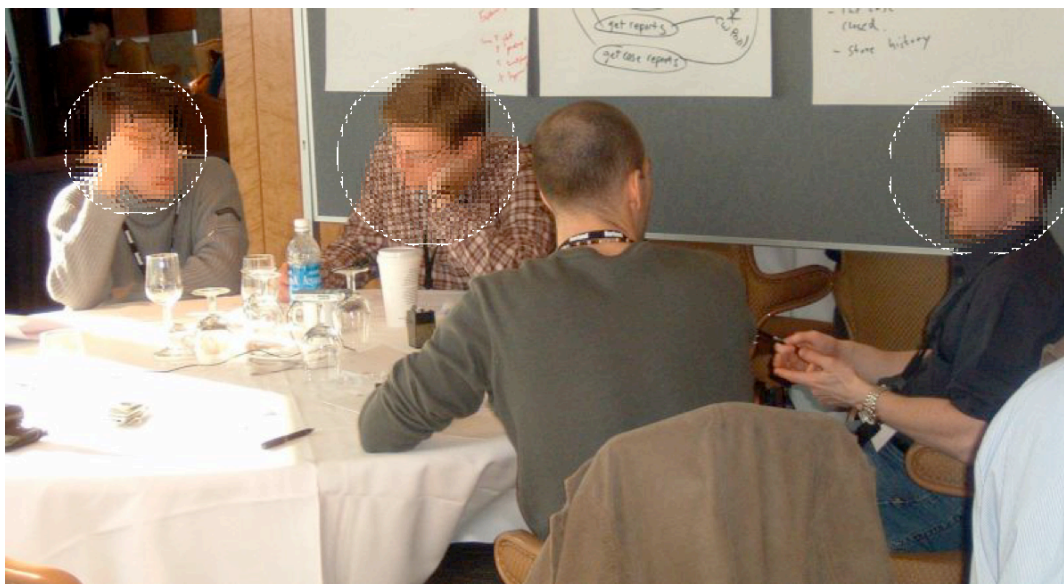
Since most of the participants did not know each other beforehand, it is interesting to investigate which factors determined the structure of teams. While social or cultural familiarity as well as interest or skill obviously played some role, the physical layout of the meeting area had a surprisingly significant effect. The random seating order around the round table shaped the first division into subteams, perhaps because of the convenience as well as the newly-gained familiarity between neighbors.

Furthermore, when gaps existed in the physical layout, contiguous chains of individuals tended to become teams. For example, the initial seating order of the full-day group was random. Seated from left to right were: *L*, *P*, *B*, *J*, and *D*. To avoid obstructing the flipchart and posterboard, *L* and *P* shifted left, leaving a gap. Although *L* occasionally moved across this gap to see the board (Fig. 2.11(a)), this gap dictated the later separation of *P* and *L* into a separate team (Fig. 2.11(b)). As the session progressed, however, both formed teams with other participants.

Spatial location also played an important part in the organization of the half-day group. Since the positions of its poster-boards and flipchart formed a wide angle, it was difficult to work on one board while maintaining awareness of the other (e.g., Fig. 2.12(a)). As a result, ad-hoc interaction was often limited to people working on the same area of the same board. Surprisingly, even though people moved about, they tended to work more with posterboards which were physically closer to their original seats.

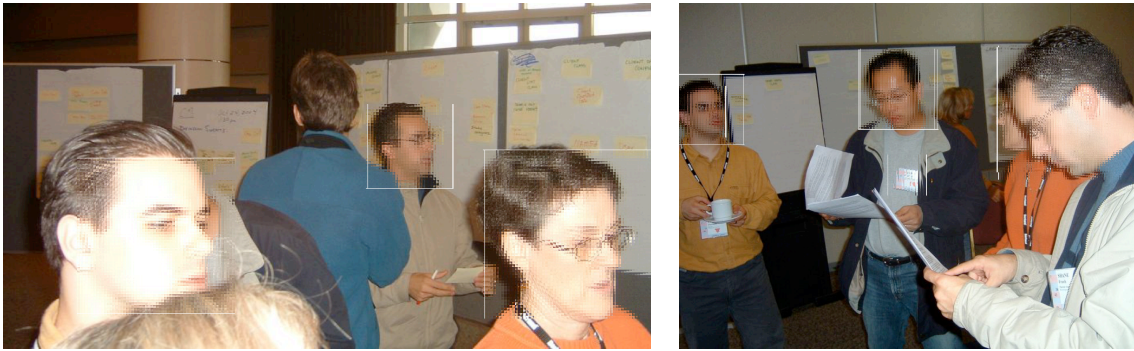


(a)



(b)

Figure 2.11: Team formation in the full-day group



(a) Ad-hoc teams formed in front of specific canvases

(b) Clumps of interactions attracted additional participants

Figure 2.12: Team formation in the half-day group

Ad-hoc interactions sometimes occurred when a person who was not part of any team was drawn to the activity of another team and joined it. For example, Fig. 2.12(b) shows one participant returning from a coffee break and joining a team that was discussing an artifact.

It is also important to note that team members tended to maintain a peripheral awareness of the activities of other teams. Awareness is important, because when teams were physically separated, they exchanged less information with one another and their composition changed less frequently. For example, due to lack of space one team in the full-day group used a second table hidden by a posterboard. To compensate for the loss of awareness, members of each team occasionally ventured into the space of the other to check on its progress, affecting the performance of both teams.

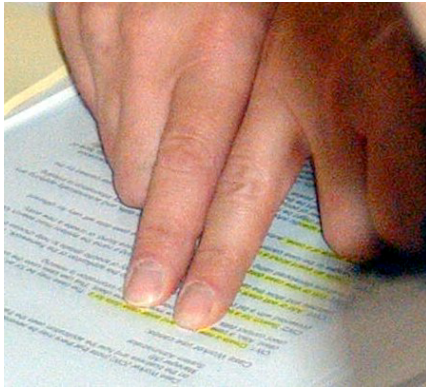
A significant problem regarding awareness arises when several people are working from one physical location, such as a conference room, in a distributed meeting. It might be difficult for those in other locations to keep track of all the parallel activity going on in that location from a single viewpoint. One approach would be to use automated means to monitor, digitize, and provide after-the-fact record of the activity [45].

2.4.1 Maintaining individual focus on artifacts

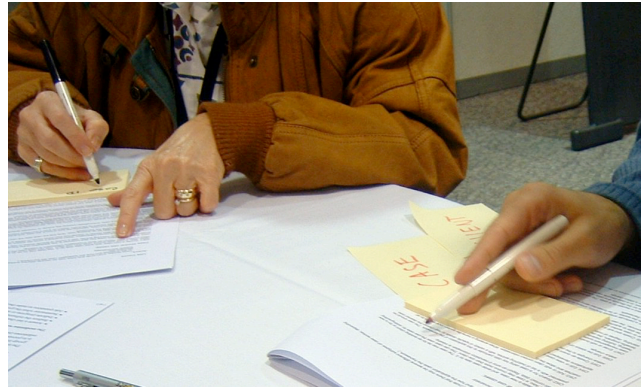
As discussed in the previous section, designers constantly shift between working with the entire team, working with a smaller subteam, and working individually. While they were working with others, I often observed their focus shifting away from their team for short periods of time. A person might, for example, review some notes or change some diagram, as can be seen in Fig. 2.12(a) where he turns around to examine an old artifact. Yet even when his attention is not on his team, he maintains awareness of its activities and is able to locate its current focus.

The complexity of the task and the distractions of group work cause individuals to intermittently lose their focus on an artifact. To maintain it, an individual who studies a nearby canvas will often use physical means, such as placing a finger or a pen at immediate proximity or in actual contact with the canvas surface (Figs. 2.13(a) and 2.13(b)). In fact, I observed designers moving towards a distant canvas in order to maintain physical contact, even if they could see it clearly from afar.

The problem is further aggravated when trying to concurrently maintain focus on two nearby items. Such a scenario is in fact very common. For example, figs. 2.8, 2.14(a), and 2.14(b) respectively show a designer creating a new diagram based on another diagram, a requirements document, and personal notes. Figs. 2.5(a), 2.3(a), and 2.9(a) show designers trying to maintain focus on small and large canvases at the

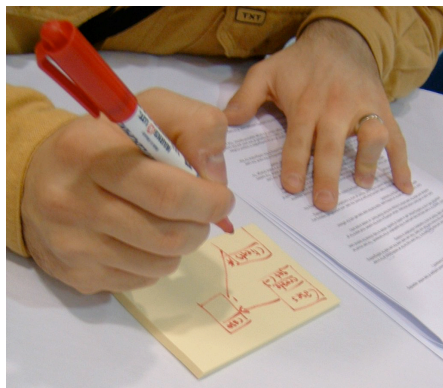


(a)



(b)

Figure 2.13: Maintaining personal focus on one item



(a)



(b)

Figure 2.14: Maintaining personal focus on multiple items

same time.

2.4.2 Drawing the focus of others to an artifact



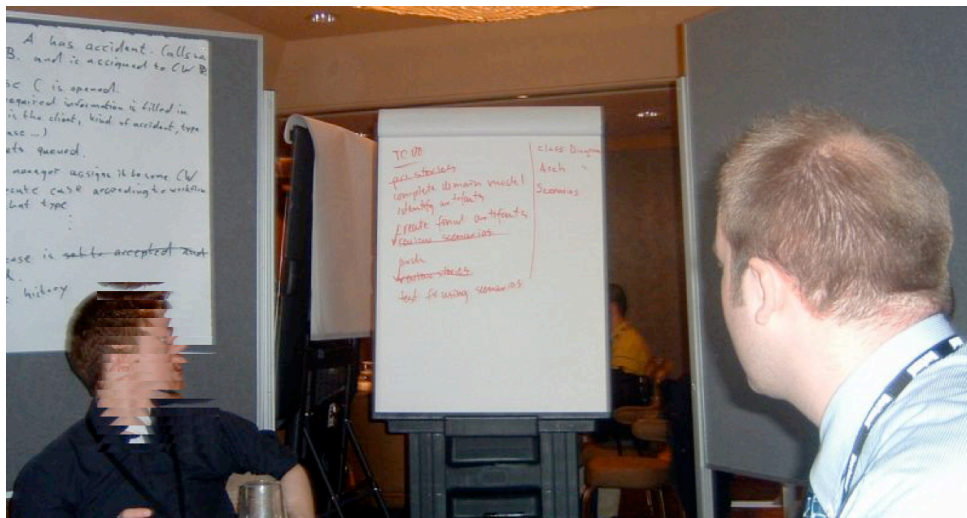
Figure 2.15: Gesturing to another person about an item at close proximity

When designers are working at close proximity to the canvas and to each other, they often use similar means to point at specific objects (Fig. 2.15(a), 2.15(b)). Since they often think in metaphors [39], they might use special gestures to indicate certain behaviors such as components that “talk” or “search for each other” (Fig. 2.15(c)). Occasionally, designers “violate” each other’s “personal space” by pointing to an artifact within that space (e.g., Fig. 2.15(d))

A related problem is of pointing in documents for which multiple copies or base documents exist. For example, Fig. 2.12(b) shows several people trying to locate the same spot in their respective copies of the requirements document.

2.4.3 Inferring peer focus

Even while breaking themselves from the focus of the team, designers continue to maintain peripheral awareness of its activities and are able to rapidly rejoin its focus. Consider, for example, the person standing with his back to the group in Fig. 2.12(a). When he turns around, he will quickly find the artifact everyone is looking at, even if nobody is specifically pointing at it.



(a)



(b)

Figure 2.16: Inferring focus from gaze

Consider Figs. 2.16(a), 2.16(b), and note how it is possible to follow or intersect the gaze of others to determine the artifacts they are looking at. In fact, it seems that we can do this almost instantaneously in our daily lives, and even identify those who are looking at other objects and ignore them. The ability to follow the gaze of others has been shown to improve performance in different tasks (e.g., [77]).

2.5 Results - Notations

Although this study focused on the physical environment and the use of canvases, I found the notational choices made by subjects to be noteworthy. Subjects seemed to initially use free-form notations with ad-hoc semantics to capture ideas while brainstorming. Some of the generated artifacts were text-based or were based on the organization of other sheets of paper. Subjects also used certain notations from UML, such as boxes for classes, but the resulting diagrams were far from compliance with the standard. Relatively-compliant UML notation was only used at a much later stage in the design, after the teams have fleshed out most of the design and were creating materials for documentation and presentation. This interesting behavior inspired my decision to focus the second study on the notational choices used by designers. I was not able to use the materials from this first study for this purpose because there were not enough quality close-up photographs of the artifacts, and as there was no video or sound track to help understand how they evolved.

2.6 Discussion and tool implications - Use of canvases

We now turn to analyzing the results described above and focusing on limitations of the physical environment and the implications for electronic tools. I will address how tools can support collocated and distributed design, and will pay special attention to the problem of preserving design knowledge.

2.6.1 Support for hierarchy of canvases

My observations in this study highlight the central role that canvases play in the design process. While design can be carried out well on a single whiteboard, it appears that access to physical canvases of differing sizes and types enriches the design process and offers developers additional flexibility.

Designers in my study frequently made use of the ability to place canvases inside one another and to move them from location to location. This was particularly useful in early brainstorming stages, as one could capture a variety of ideas or entities on separate canvases without having to worry at all about classifying them or organizing them. We have also seen that some canvases are more than mere containers for information - they have types and they convey a role.

Even today, the ability to utilize a variety of nested containers is not present in most design support tools or even in most mainstream drawing tools. Formal modeling tools typically follow UML conventions of one diagram per sheet and offer a standard drawing canvas on which only the corresponding UML primitives could typically be placed. Collaborative design support tools are somewhat more flexible in what can be placed on the canvas but tend to stick to the whiteboard metaphor of providing a single general drawing surface on which primitives and freeform drawings are placed.

I argue that tools that intend to support free-form software design and seek to offer maximal flexibility must move away from a simple whiteboard metaphor and into a canvas-based metaphor. In doing so, they must go beyond the capabilities of drawing tools, which use a single canvas, and beyond most diagramming tools, which support groupings of primitives on a single canvas. Rather, canvases can be placed inside the main canvas, and each should potentially be capable of supporting the inclusion of any content or any other canvas. While some tools (e.g., [48]) support a single level of embedding, my findings suggest that recursive embedding should be permitted.

The location of each canvas in the design space and in relation to other canvases and the hierarchy should be easy and natural to manipulate. If an electronic whiteboard or other touch-sensitive medium is used, manipulation via touch and perhaps using real-world analogues should be supported [48]. We

note that supporting a hierarchy of canvases in synchronous collaborations is far from trivial, as there are challenges even with supporting standard grouping operations [44] when multiple users may operate on the same entities.

We also note that a recent tool called *Calico* [61] takes an important step in the direction outlined above. It introduces the notion of “scraps”, which are essentially groups of elements on the primary drawing surface that have been “lifted” off that surface and can now be manipulated as an independent unit. Unlike standard groups, however, each scrap functions as a drawing surface that can accept new content. Scraps can also be placed or grouped on top of other scraps, an important step towards our proposed containment hierarchy.

2.6.2 Knowledge preservation and canvas hierarchy changes

While the ability to modify the containment hierarchy over time offers a great degree of flexibility, it also presents a serious challenge to the interpretation of the resulting artifacts. Over the course of the design process, each canvas can appear in multiple locations and with different “parents” and “children”. I argue that intermediate locations may convey important design knowledge that would be lost if one relied only on the state in the final diagram.

For example, if a sticky note belonged at some point to a group that was assigned a specific property or semantics, that property may still be valid even after the note has been moved. Similarly, other information such as time of creation and the identity of the original author may also be lost. More generally, I argue that the location of a canvas in proximity to other canvases affects its interpretation and the interpretation of its contents.

Tools for supporting design should therefore not only preserve the final state of the artifacts and the canvas hierarchy but also a complete history of past states. Such a complete history is important for design since any past decision may be revisited at some point in the future, so earlier states may be occasionally traced when seeking answers to specific questions. In addition, as we later describe, such a history may be necessary to index conversations and activities.

2.6.3 Organizing canvases

All observed teams generated a large number of artifacts on a large number of canvases, including numerous large top-level canvases. Work on these canvases was not sequential and independent: canvases on which most of the immediate work has concluded continued to play an important role. Some were subsequently modified and updated, and many canvases influenced work on other canvases.

Nevertheless, the limitations of the physical design space and likely also of human ability forced designers to use a small working set of canvases which were easily accessible and place everything else in secondary storage. Teams struggled to maintain a working set of visible artifacts within the limited working space and invested significant effort in finding and retrieving diagrams after they were removed from that set. Teams also tried to increase the physical locality of related information, placing related information adjacently or embedding canvases within one another. The costs associated with physical mediums prevented them from doing so in all applicable situations, especially for short references, and they often needed to switch attention between different areas of the workspace.

The main advantage of the physical environment is that various cues, including direction, light, and elements in the room helped designers be grounded in their surroundings and have reference points for quickly recalling the locations of items.

Large electronic sketching surfaces offer the potential for alleviating these problems by creating

a virtual drawing space that is much greater and more flexible than any physical canvas [30]. With appropriate interaction techniques, they can help teams rapidly identify artifacts in this space and bring them into physical proximity within the limited physical viewport. Multiscale interfaces [36] can help overcome the inherent resolution limitations of these displays. However, in an electronic tool that has a limited physical viewport, such as an electronic whiteboard, there are limits to the number of canvases that can be concurrently viewed. In more immersive environments, reference points may be lacking.

The approach to material organization and recall in electronic tools must therefore be different. In environments where keyboards are used, one alternative is to use of hot-keys or “virtual spaces” to access specific elements in the working set and to use a simple search mechanism to search the secondary storage.

Another option is to rely on the location of elements within a larger virtual canvas and on their apparent shapes, such as showing an overview or map of the design area. An interesting variation of this approach, which combines an outline view with the use of virtual spaces, is taken by *Calico* [61]. The tool organizes canvases on a grid, supporting both an outline view of the entire workspace and very convenient switching between adjacent canvases in the grid. One can imagine how this could be extended to support multiple viewports so that one canvas can be referenced while working on another.

In larger projects that involve many design sessions, the total number of canvases may be too great to track with simple means. I believe that techniques borrowed from the organization of digital images, such as as thumbnails and tags, may be useful for organizing the diagrams and facilitating recollections. As the design process lengthens, access based on timelines becomes particularly valuable.

Although my focus here has been on supporting organization with electronic tools, the greatest implication of my findings is that since canvases are passively read while discussions take place or other artifacts are being edited or created, it may be necessary to preserve that information over time.

2.6.4 Canvase sizes, splitting, and merging

We have seen that physical canvases, especially ones based on pen-and-paper, have two significant drawbacks: They tend to be rigid in their sizes, and they cannot be modified without residual clutter that eventually requires a redraw.

Electronic tools are very adept at automating these tasks but may have several caveats. First, automated changes that affect the layout of nearby objects may disrupt specific intentions of the designers. Second, the mundane tasks involved in cleaning and recreating diagrams are often an opportunity to rethink and improve the design, and these would be eliminated with manual redraws. Third, while automatic tools can facilitate the process of splitting and merging canvases or changing their containment hierarchy, these activities often result in the loss of important contextual information.

2.6.5 Private and public space

A common behavior among subjects in my study was to initially create artifacts in the private space. These artifacts were later discussed with others and then recreated on the main canvas at a larger scale, or the other way around.

Electronic tools can help designers avoid the mundane work of recreating an artifact in the shared space. Certain design support tools (e.g., [89]) support private spaces and public space, and allow easy migration.

Since many materials in the shared design artifacts have their roots in the private space, support for preserving earlier states must reach all the way into this private work.

2.6.6 Canvas types

Support for specific canvas types can easily be added to the model of canvas containment hierarchy. By restricting certain attributes of a canvas to specific values, such as size or color, it is possible to create prototypes or primitives. In fact, the meeting moderator could guide participants towards a certain modeling style by setting up a palette of recommended canvas primitives. The moderator could then indicate a shift in the design phase by changing this palette.

2.7 Discussion - Drawing activities and focus

We now turn to discussing how individuals and teams draw and how they maintain their focus.

2.7.1 Drawing activities and versions

I frequently observed that visual content was rendered in bursts with long periods of quiescence. While this may typically result from the incremental nature of the design, I believe that another major factor is the perceived cost of reversing transient changes. Designers frequently needed to capture brainstormed ideas or illustrate examples and proposals in visual ways. However, the costs of creating such illustrations over the existing diagram and then erasing them is very high. Instead, participants frequently spent much time considering or debating changes before actually making them.

As part of this behavior, participants often “wrote in the air” with their pens to avoid committing early to “permanent” changes. This behavior was particularly common in the context of short interactions such as answering a particular inquiry. All such gestures are lost with physical surfaces, along with the knowledge that they carry. This is particularly problematic if the intended drawings conveyed design options that were considered but rejected as they carry significant rationale about the options that were eventually chosen [60].

One major advantage of electronic tools is the ability to easily roll back unwanted recent changes, which may encourage designers to actually draw or write. The second advantage is that all rolled back changes can be preserved for future reference. The checkpoints for such reversals can be explicit, but can also be implicitly based on periods of inactivity. However, suitable mechanisms would be needed to distinguish temporary illustrations that need to be erased from the unrelated concurrent activities of other individuals and subteams.

Since we now turn to discussing focus, it is important to mention that it may be possible to help individual developers and subteams remain oriented on recent changes by making these changes more salient. For example, we could paint using a luminescent virtual ink color which “dries up” into the regular ink color as time passes. This will also help others see where the most recent activity took place if they wish to join it or to avoid interfering with ongoing work.

2.7.2 Individual focus

Designers in my study attempted to maintain orientation and focus on artifacts via direct touch. While this was most common while reading long texts on preprinted sheets, it also occasionally occurred when reading specific elements that were attached to public canvases. In a transition to electronic tools, it is important to continue supporting this behavior without triggering unexpected interactions with touch-sensitive devices.

2.7.3 Attracting peer focus to an artifact

In collaborative work, designers frequently had to draw the attention of their peers to specific artifacts and items within them. They typically first drew the attention of specific peers vocally, and then they helped them focus on a specific area by gesturing or gazing. Other peers were later drawn to the activity and followed the gaze to find the point of focus.

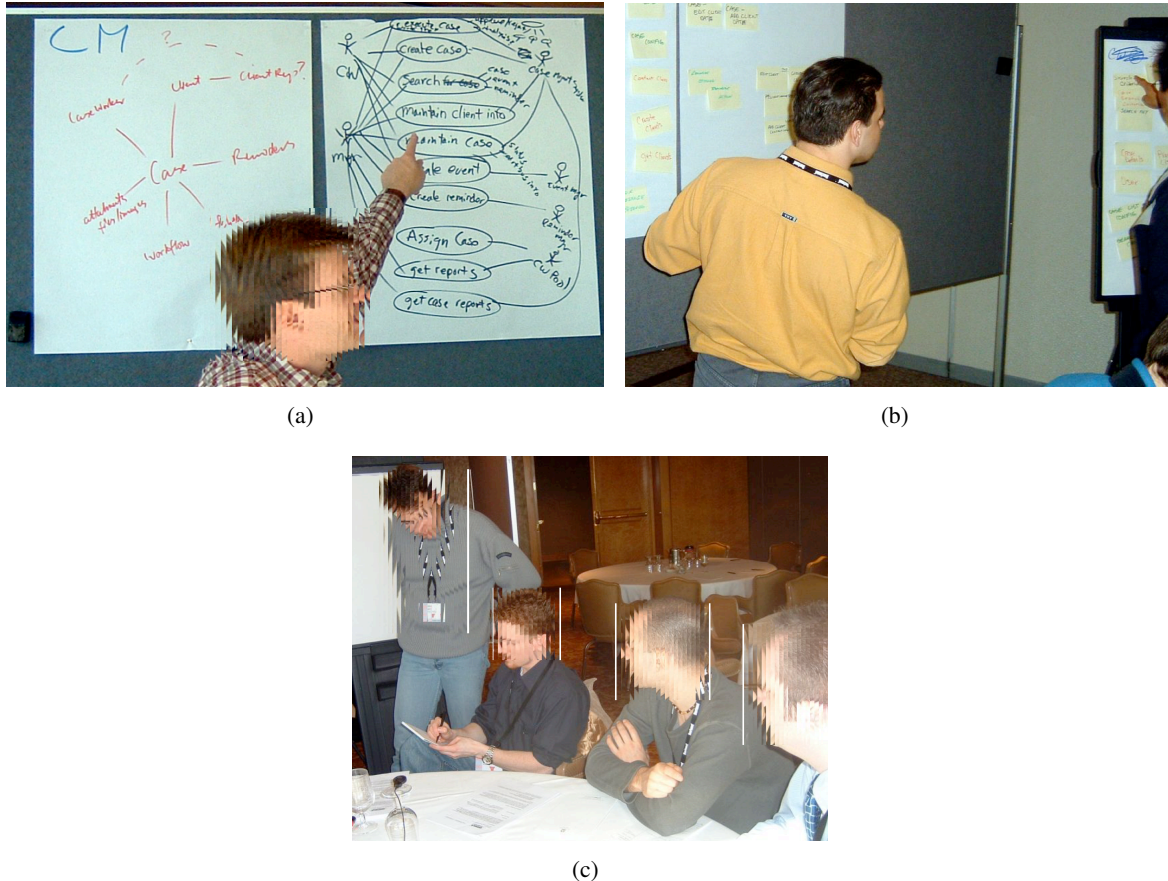


Figure 2.17: Pointing at a remote artifact

Note that identifying the target of a gesture or gaze in physical settings may be difficult when the canvas is particularly large or small (Fig. 2.17(a)), when it is far away (Fig. 2.17(b)) or when it is obstructed by others (Fig. 2.17(c)).

While distance and size limitations can be overcome in electronic tools, the ability to gesture or infer gaze is often not available. To compensate, the tool must first offer a way to start a new collaboration session with a peer around a specific artifact, as was done for code in *Jazz* [43, 15]. Second, for existing collaborations there must be a way to draw the peer's attention to an artifact without causing complete disorientation. Third, other participants must be aware of the focus and be able to quickly join it.

One possible way to provide awareness of focus regions is by highlighting viewports in the shared space. I believe, however, that simply drawing a box to represent the viewport, as done by some shared editors is distracting, especially when structured diagrams are viewed. Instead, I suggest that when one user is examining an area that is currently being viewed by another, a "spot light" would be projected to represent the peer's viewport. The area would therefore be significantly illuminated, with a stronger intensity towards the center and gradually decreasing intensity towards the edges. The main advantage of

this approach is that illumination is additive. Therefore, if many viewports are overlapping, that region will be illuminated at high intensity, helping other users identify and join it faster. In addition, it is easy to identify slight shifts and gaps in awareness.

Note that in some cases the gestures used to attract attention or refer to other artifacts carried special meanings. For example, a gesture was used to indicate that there is a dependency or flow between two artifacts. Some analogues may be necessary in an electronic environment.

Finally, turning to knowledge preservation, note that since exposure to existing materials may affect the interpretation of conversations of artifacts, it is important to try and preserve a record of the focus of all individuals over time.

2.8 Threats to validity

2.8.1 Impact of settings

Observational studies of software development in the literature take place in the lab, in academic settings, or in industrial ones. Research in artificial settings enables the collection of data which could be difficult to gather otherwise. Of course, these advantages involve a tradeoff, as artificial settings raise the question of whether the results could be generalized. Clearly, additional research in industrial settings is required to establish that the observations I made here hold more generally. All settings are unique in certain respects, and replication in multiple organizations may be necessary. Nevertheless, let us briefly mention the primary ways in which *DesignFest* differs from industrial settings and estimate their potential impact on generalization.

First, design takes place in isolation from other, overlapping, development phases; there is no prior requirement gathering, though in some sessions the *DesignFest* organizers played the role of clients and spent time with each team. More importantly, the produced designs are not subsequently implemented. However, *DesignFest* problems are based on real-world specifications, and most participants appeared passionate and serious in their work. It is thus likely that the proposed designs, or at least the initial high-level sketches, would be similar to those created by industrial teams early in the design phase.

This brings us to the second problem of the extreme time pressure compared to industrial settings. While more research is necessary, I believe that my data reliably captures how representations are selected when teams first face the problem, an issue for which limited prior knowledge exists. The impact of the time limits is likely to be primarily on how details are fleshed out, documented, and reviewed; these issues have been studied more extensively by others.

Third, *DesignFest* participants receive no material compensation for participation and are not held accountable for their work; the motivation is primarily to learn and interact with respected peers. In that sense, it resembles open-source settings, where nonmaterial compensation is key to contribution. This similarity also applies to the diverse background of the participants, as industrial teams typically have shared prior experiences and practices. However, I explicitly tried to minimize such organizational bias.

Beyond the threats to validity imposed by the settings, one main limitation of this study is that it relies on a small sample of two groups. It may therefore be difficult to generalize to design in general, and there are no guarantees that similar phenomena would be seen in additional observations. For this reason, the follow up study samples a larger number of groups.

A second, related limitation is that I was the only individual to examine and analyze the data. There is therefore a risk that important details were missed or that the focus is misplaced. Naturally, the analysis and conclusions are also subjective. Nevertheless, the importance of this study is in highlighting certain

issues that may affect design and the use of the resulting artifacts, and therefore in helping focus the data collection and analysis of the follow up study. In particular, it led me to focus on the artifact contents and on the preservation of knowledge.

Chapter 3

Studying the Notations and Representations in Collaborative Software Design

The previous chapter described my first study, which was conducted at OOPSLA 2004. It relied solely on photographic evidence and my recollection of the events, and focused primarily on the physical drawing environment. My initial intention was to aid design by developing tools to support the physical drawing activities. However, when I contrasted the photographic evidence with my recollections of the session, I began to suspect that there is a much more fundamental need to support the drawn contents themselves, both at the time of the session and in the long run following their creation. This requires an understanding of the notational choices made by designers.

3.1 Study overview

3.1.1 Background

As previously discussed, software design is a highly visual activity where diagrams are used for brainstorming, grounding, and communicating ideas and decisions [16]. Various notations have been proposed for expressing designs in the object-oriented paradigm, and one of them, the *Unified Modeling Language* (UML) [68], has become a widely recognized standard. Many software packages and CASE tools enable designers to create UML models for documentation and implementation purposes. Recent efforts make UML modeling functionality available to teams using electronic whiteboards. Such tools are either multiuser extensions of CASE tools (e.g., [38]), or tools that help transform freehand drawings that include UML constructs into full UML diagrams (e.g., [14, 22]). Some also offer support for the systematic capture of decisions and rationale [11].

However, casual observation and the results of my first study suggests that OOD teams do not fully utilize these standards. Instead, they tend to follow an erratic process in which they generate a plethora of diagrams and fragments. These visual artifacts are not only less aesthetic and complete than those created electronically, but most importantly, also frequently diverge in their notation from UML, in some cases substantially.

While this behavior can be dismissed as an attempt to cut corners or of a need for agile design notations [3], it highlights important issues for the object-oriented paradigm that has received little attention. First, while we have powerful notations, formalisms, and tools for expressing finalized designs, we know

surprisingly little about the processes and representations by which teams create these designs in the first place and whether the same tools are applicable in these early stages. Second, while we know that designers often sketch in alternate notations, the origins, semantics, and roles of these notations remain unclear. Third, we know little about the impact of sketches created in these notations, the connections between them, and whether or not they can be straightforwardly used for documentation or implementation or transformed into compliance with standard formalisms.

These issues have many significant implications. For example, it is not clear that UML, which is primarily a specification and documentation standard, can also effectively be used as a design notation in early phases and in collaborative settings. Should efforts be invested in modifying UML or in coming up with a different notation? Can UML-based design tools be augmented to effectively support early collaborative design?

Despite the abundance of literature on sketching and collaboration in general, only a few works [23, 22, 14, 88, 89] address the unique settings of collaborative OOD, and these primarily focus on the eventual automatic transformation of sketches into complete and aesthetic implementation- and archival-quality UML models. They appear to treat the creation of these finalized models as the primary motivation behind drawing activities and tend to consider the early sketches as premature or peripheral artifacts that will evolve in due time; digressions from UML are treated as accidental and are deprecated.

While support for transforming and completing these models into a standard notation is likely valuable for use in later development stages [16], I suspect that this approach does not “tell the entire story” about OOD representations, nor does it support all the unique needs of design teams. Rather, there may be important underlying reasons behind the representational choices teams make which have significant impact on how teams work in the object-oriented paradigm and on how practices and tools can support them.

3.1.2 Goals

In this second study of collaborative design, I seek answers to several questions:

1. What kinds of representations are constructed by design teams as the design evolves, and how do they diverge from UML?
2. Why are such nonstandard representations constructed and what purpose do they serve?
3. How are these representations used throughout the design process?
4. Can a better understanding of these representations suggest new practices and forms of tool support?

To answer these questions, I conducted a second observational study at the OOPSLA *DesignFest* event. This study used video recordings and focused on representation and diagram use.

My intention in this study was not to comprehensively catalogue or explain all the improvised notations used in collaborative OOD or to offer quantitative information about representational choices, although these are interesting avenues for subsequent work. Instead, the presentation here is focused on typical cases in which alternate notations are used, since these have little to do with the completeness of the model and are perhaps the most revealing about representational needs and choices. In addition, I focus on several related issues that have significant implications for supporting collaborative OOD but that have received limited attention: How heterogeneous information is represented, the characteristics of designs that spread over multiple diagrams, and how teams cope with this delocalization.

3.1.3 Methods

Collected data

The observations presented in this chapter are primarily based on extensive data collection in the OOPSLA 2005 session of *DesignFest* and always refer to it unless otherwise noted. A follow-up study was carried out at OOPSLA 2006 for validation purposes, but its results were not analyzed at the same depth and will only be discussed anecdotally. In this 2005 session, I made video recordings of the design sessions of several teams.

I chose video in order to capture as much context as possible, including a trace of all interactions, annotations, and references to artifacts. Because teams tend to post and refer to materials all around them [25], I used widescreen camcorders to capture more of the design area, one of them filming in high-definition. To enable us to evaluate the value and limitations of more available means of preserving design knowledge, I also made a separate audio recording and frequently took still photographs. These additional mediums also served as a redundancy for the material in the videos.

Due to the sensitive nature of video recordings, significant measures were taken to ensure participant consent. Prior to the conference, the *DesignFest* organizers sent pre-registered participants an email describing the study and asking for preliminary consent to different recording mediums. The large number of preregistered participants allowed the organizers to form initial groups based on consent without denying anyone their choice of design problem. Since the *DesignFest* event is also open to walk-in participants, potentially affecting group composition, I also verbally introduced the study to each of the consenting groups and obtained written consent. Note that all names in this chapter have been changed.

Design task and groups

The real-life design problems available that year were a medical information system for a pediatric therapy clinic, a system for a web-based image management and photo printing business, and a generic simulator for third-party controllers in industrial production lines. The first two problems are representative of typical information and eCommerce systems, while the last is an example of a more exploratory project in a different domain.

Since many consenting teams were working at the same time, my selection of observed teams attempted to obtain a blend of problems. I sampled only groups that had at least four participants at the beginning of the session and that were located in areas with limited acoustic interference from other teams. Depending on the session, teams worked between 3 and 6 hours excluding breaks, and I tried to film the entire session of each team. In one case, I switched from team D to team E to capture more drawing activity. Overall, I obtained over 20 hours of video footage from seven teams, summarized in Table 3.1. I also obtained still photographs of the work of other teams.

Group	Problem	Session	Length	Recorded
A	Simulator	Sun PM	3 hours	All
B	Image Shop	Sun AM	3 hours	All
C	Image Shop	Sun PM	3 hours	All
D	Medical System	Sun Full	6 hours	First 2 hours
E	Medical System	Sun Full	3 hours	Last hour
F	Medical System	Wed Full	5 hours	All
G	Medical System	Wed Full	5 hours	All

Table 3.1: Groups for which video footage was recorded

Data analysis

Before proceeding to analyze the data, all video footage was digitally transferred to a video editing software, allowing us to effectively browse hours of footage to trace the evolution of artifacts. The tapes were then fully transcribed as \LaTeX documents, allowing us to create versions limited to dialogue and versions with descriptions of additional activities.

My analysis of each team's work typically proceeded as follows. First, I studied photographs of the finished diagrams from the end of the session and tried to understand them without additional information, as potential consumers of these materials would have to. Next, I examined all the photographs taken throughout the session and used temporal cues to obtain a better understanding of the artifacts and their evolution. Only then I read through the transcripts and eventually watched the entire tape. Throughout this process, I made notes of relevant observations. Eventually I pooled and studied the observations from all teams, identified the examples which I present in this chapter, and returned to the materials to study them in depth.

3.2 Results: Alternate notations in individual artifacts

The presentation of the results begins with the representations used in the collaborative creation of individual artifacts.

3.2.1 Use of UML and divergences

As could be expected from the venue, UML was utilized by all observed teams. Specifically, all teams drew class diagrams, and several drew sequence diagrams, use-case diagrams, and even one package diagram. Perhaps due to the limitations imposed by the settings, the nine other diagram types were hardly used. In conformance with casual observations and prior works, I frequently observed departures from UML and proceeded to investigate them.

Previous researchers [23] suggested that divergences from UML are mostly accidental and that diagrams eventually comply with the standard. When I observed teams explicitly reproducing artifacts for presentation at the conference's final event, they indeed tried to create aesthetic and complete UML models. However, most of these attempts took place towards the end of the session, after the brunt of the creative activity was done, and then had a relaxed and visually distinct interaction style. Teams often split up, and subgroups worked on recreating diagrams on new canvases instead of modifying existing ones.

In the creative phases, however, I often observed situations where the departures from UML appeared to be intentional rather than accidental. Specifically, I saw artifacts which convey information that could have been expressed in a straightforward manner via UML, but were created in completely different notations or which violated significant principles of the standard. These cases are particularly interesting since they yield significant clues about the behavior and needs of design teams.

To help the reader distinguish the detailed objective descriptions of activities and artifacts from my subjective interpretation and analyses, the former are presented as *italicized* text. Also, the textual descriptions frequently refer to photographs of the artifacts.

3.2.2 Adapting to evolution

Certain UML diagram types convey multiple layers of information. Class diagrams, in particular, not only list the classes of the system (or its entities in the case of domain-modeling [3]), but also specify the members of each class, and the object-oriented (e.g., inheritance) and data cardinality (e.g., one-to-many) connections between classes. The notations of UML allow designers to add this information in any order while still maintaining the syntactic correctness of the diagram.

Nevertheless, I observed a number of cases where diagrams ended up with all these layers of information but expressed in an entirely different form. Teams tended to select diagram types opportunistically in order to address the issue at hand, and captured knowledge as it emerged from the problem-solving process even if it did not fit the current selection. In some cases, they began constructing a diagram of one type only to have it morph into another. On other occasions, the diagrams turned into a collection of fragments related only by their relevance to a particular design issue. I present three such examples in detail in order to convey the process by which these changes occur.

Example 1 - Structural domain model

The first example comes from team A, which was working on the problem of constructing simulators for third-party controllers of production lines.

*The team began its session by discussing assumptions, and then turned to exploring use-cases. Jack, standing by the flipchart, titled the blank canvas *Stories*. Then, somebody suggested that they first cover high-level domain entities, and others agreed. The team started brainstorming ideas, which Jack scattered on the titled canvas.*

Already, we have a discrepancy that may confuse future stakeholders: based on the title one may expect to see the names of use cases, but the diagram actually contains the names of entities. This is not a significant problem for team members who spent most of their time on entities and may not even recall the original title.

*Very soon, it turned out that the proposed entities could be related and questions were posed and discussed. Are *Node* and *Waypoint* merely synonyms? Are *Source* and *Destination* unique entities, and is there more than one of each? Appearing to make the decisions himself, Jack used parentheses to make *Waypoint* an alias of *Node*, and he then replicated *Source* and *Destination* under *Node*, with a small right-angled arrow to indicate the connection (Fig. 3.1).*

Improvising the arrow notation allowed Jack to capture the connection without disrupting his attempts to also capture the barrage of brainstormed ideas. However, the figure became inconsistent with class diagram notation and its semantics unclear. Jack did not clarify the semantics of the connection, and when another participant proposed a specialization relation, he did not hear it or simply ignored it. Nevertheless, he appears to have memorized this notation for subsequent idiomatic use.

Note, though, that even in the early stage of Fig. 3.1 the diagram contains a risky redundancy, as *source* and *destination* appear twice, once independently and once under *Node*. A user skimming the diagram is more likely to notice the independent *source* and *destination*, and may not realize the fact that these entities are connected to *Node* because the information is slightly delocalized.

*As the team continued to brainstorm, a notion was suggested for the inventory of items at a particular location. At first, it was captured via the *Count* property of *Node* and listed with the same right-angle arrow. A replica of *Count* with the arrow was then added under *Path*.*

*Further discussion of the notion of inventory lead to the realization that *Source*, *Destination*, and *Waypoints* are all special kinds of *Node*. The representation was changed, with *Waypoint* listed*

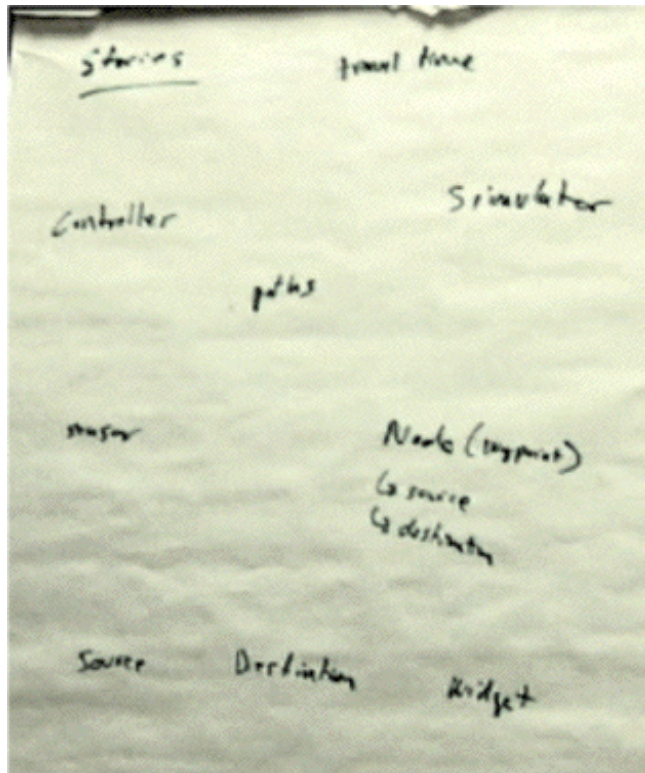


Figure 3.1: First step in domain model diagram of team A

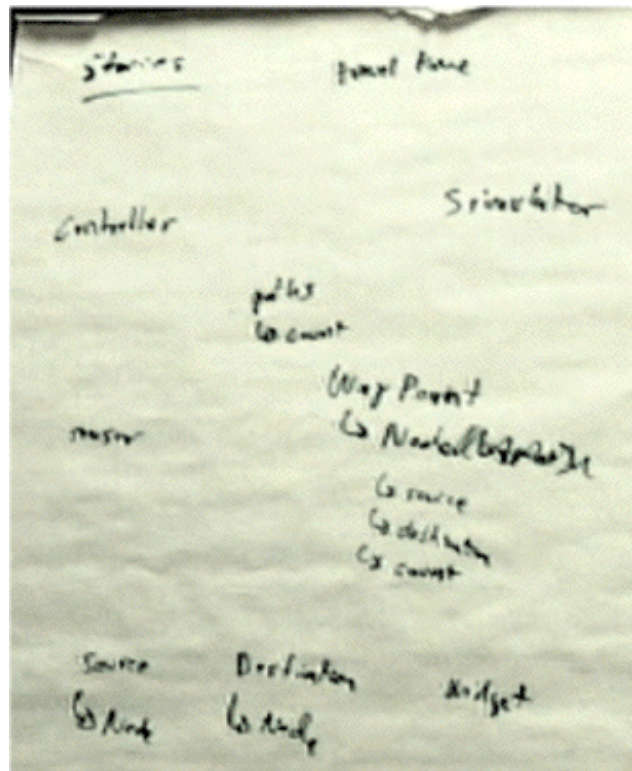


Figure 3.2: Second step in domain model diagram of team A

above Node, and Node listed under the independent Source and Destination entities (Fig. 3.2). It took a while until the redundancy of these entities under the original Node was fixed.

The diagram in Fig. 3.2, is now quite confusing. Source and destination appear twice, and the connection of each to Node appears twice. However, since the semantics of the arrow are not clear, it is possible that the vertical direction of the arrow carries semantics. In this case, the fact that Node appeared once under and once above the source or destination is misleading.

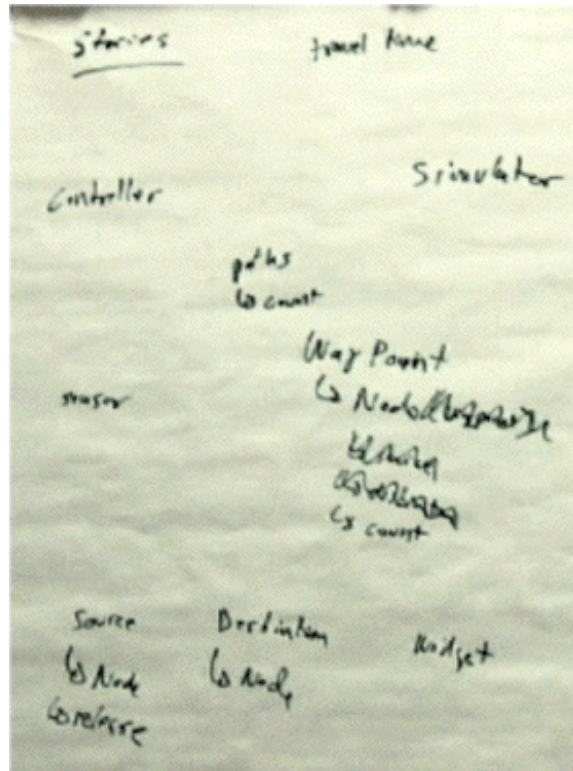


Figure 3.3: Adding methods to the domain model diagram of team A

Later on, after a Release entity was proposed for the Source and written down (Fig. 3.3), someone suggested writing the behaviors in a different color, allowing more methods to be added at various locations around the entities.

The resulting diagram, depicted in Fig. 3.4, conveys the same details as would a class diagram used for domain modeling [3]: candidate classes, properties, methods, and inheritance. However, from the point of view of “proper” UML modeling, its notations are ambiguous without the context of specific discussions and elements. For example, inheritance, which in class-diagrams appears as a direct line from a subclass to a superclass above it, appears here as an arrow from a superclass to the name of a subclass below it. The problem is not only with the direction but also with the replication of the superclass name. In addition, the exact same notation is used to represent a property of a class and is also similar to the notation for a method.

These problems may preclude this diagram from serving as a documentation artifact, at least without contextual information. For example, how would someone viewing this artifact infer that Count is a property of Node and not a superclass?

Nevertheless, this diagram appeared to facilitate brainstorming as a preliminary step to coming up with a solution. By writing all brainstormed nouns around the board, the team was able to avoid an early commitment to distinguishing classes, fields, and methods, which require different notations in

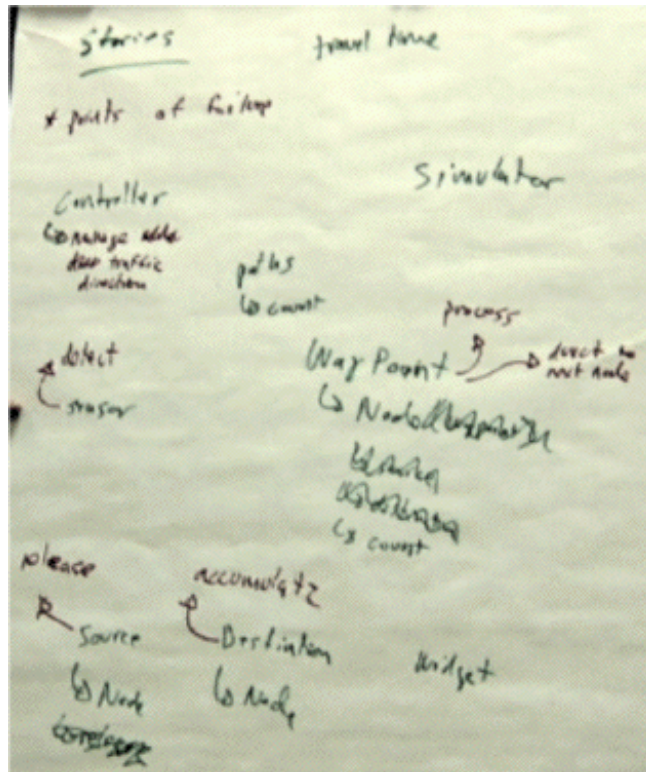


Figure 3.4: Final domain model diagram of team A

class-diagrams. Furthermore, keeping the related entities for each candidate class in close proximity, sometimes at the cost of replication, allowed the preservation of the diagram as primarily a catalog of entities rather than as an attempt to fully structure the domain.

Example 2 - structural solution model

In the second example, team E, which was working on the medical information system, arrived indirectly at a class diagram while retaining some inconsistent notations.

The team was trying to envision the usage model of its system and started drawing a sequence diagram that started with a user making a request. Since users interact with the system and provide it with additional information via web forms, there was soon a need to represent these forms. Craig, standing by the posterboard, placed a page that contained a rudimentary architecture diagram immediately to the left of the sequence diagram and partitioned it. In the remaining area, he started drawing rectangles for UI forms, writing the names of important actions inside them (Fig. 3.5). More interaction was specified in the sequence diagram, and additional web-forms were added.

By increasing the spatial proximity between the sequence diagram and the UI diagram of Fig. 3.5, the team was able to follow and manipulate the representations of two facets of the system at the same time. No evidence, however, remains to alert future stakeholders of the connection between the two diagrams. In addition, any decision or information associated with an entity in one diagram may not be visible to someone examining the corresponding entity in the other diagram.

As forms for treatment plan and treatment step were added, a one-to-many relationship was realized and captured (Fig. 3.6), followed by more cardinality connections (Fig. 3.7). These changes essentially turned the form collection into an entity-relation diagram. Then, the team realized that

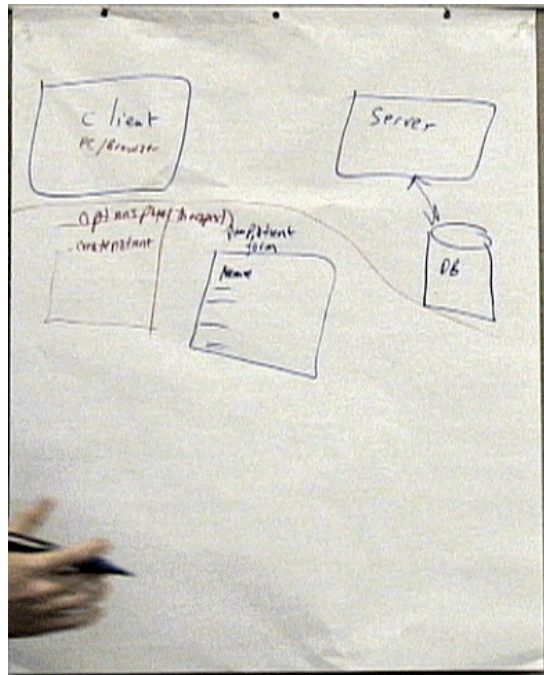


Figure 3.5: First step in the solution model diagram by team E

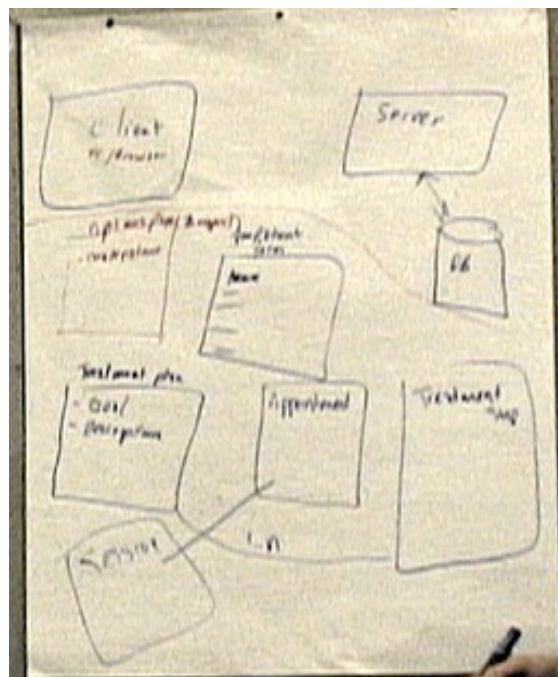


Figure 3.6: Solution model diagram by team E becomes an entity-relation diagram

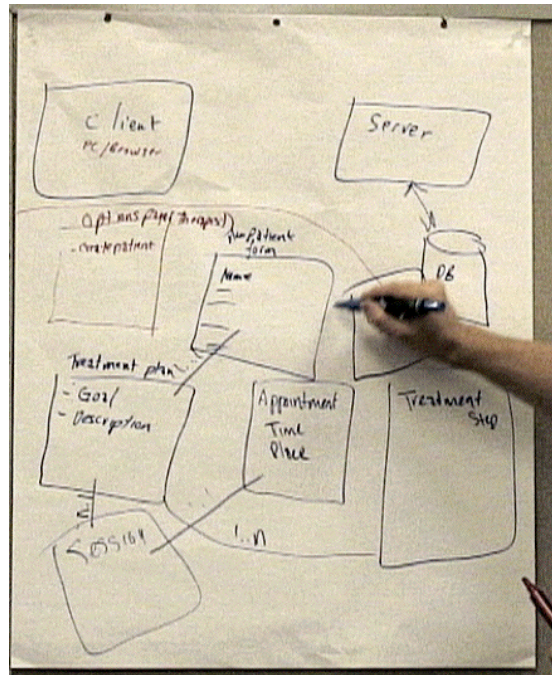


Figure 3.7: More relations are added to the solution model diagram by team E

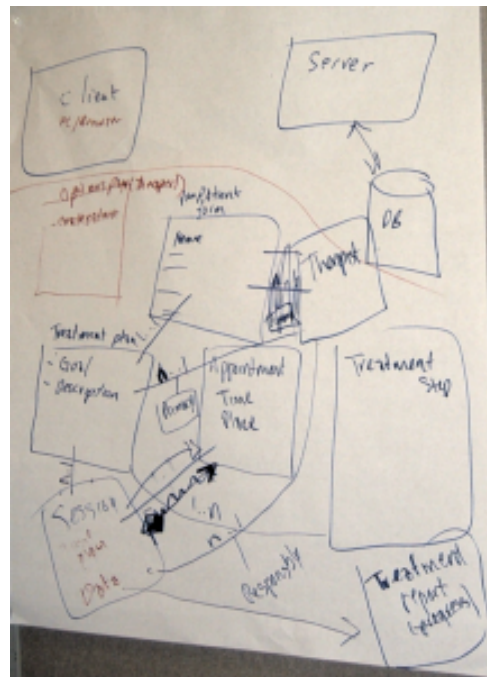


Figure 3.8: Final form of the solution model diagram by team E

a *Session* is a special kind of *Appointment*, and added a parallel inheritance arrow to represent it, turning the drawing into a class diagram. In addition, a line representing control- and data-flow was added from the *Session* to the generated *Treatment report*, thus diverging from class-diagram notation (Fig. 3.8).

Although the final version of this diagram resembles a compliant class diagram, it contains additional layers of information such as the contents of some web-forms. In addition, certain entities represent entry points into the sequence diagram, but this contextual information is not captured in writing and is lost. By combining all this information into one diagram, the team was able to treat multiple facets of each entity without replication.

Example 3 - behavioral domain model

Alternate representations evolve not only for structure but also for behavior and function.

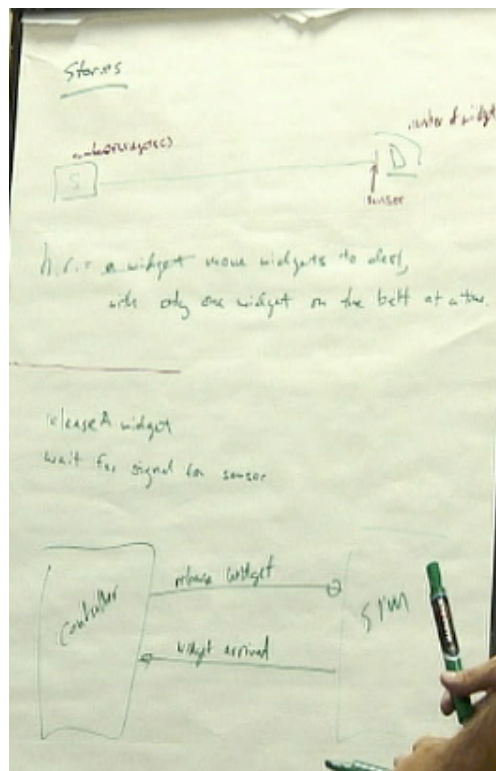


Figure 3.9: Early form of the functional domain model by team A

Team A finished the domain model of the first example and began to explore the interaction between the controller and the simulator with a simple scenario. On a new canvas, they created a small map of the simulated production line, consisting of one source and one destination connected by a single conveyor belt. To describe the behavior, Al wrote down a business rule, requiring a widget to arrive at the destination before the next one leaves the source (top of Fig. 3.9).

Now, Al wanted to describe the behavior of the controller and specified it in two textual steps (middle of Fig. 3.9). Jack approached the board and added a sequence diagram to represent the interactions between both sides of the system (bottom of Fig. 3.9). When they realized that the controller's behavior is continuous, Al added a third step, *send next one*, and created a flowchart arrow from it back to the first step. Someone then mentioned the need for an exit-condition, and Al turned the steps into pseudo-

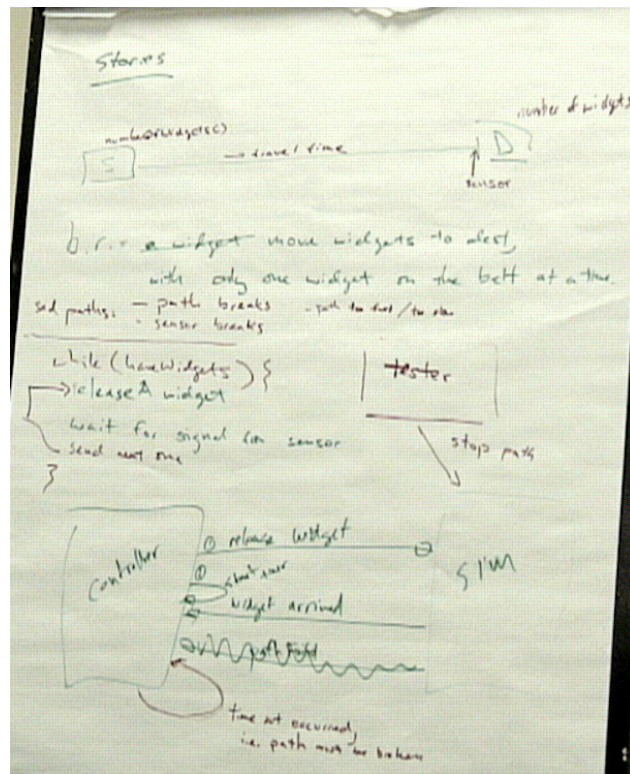


Figure 3.10: Final form of the functional domain model by team A

code by wrapping them within a `while` loop with curly braces (middle of Fig. 3.10). Later, the sequence diagram was modified to represent this change, and an external testing agent was added. (bottom of Fig. 3.10).

We see that differences in individual perspective can lead to representation choices that differ in their ability to cope with the evolution of the design. AI was focused on the controller and described its behavior in isolation, whereas Jack preferred a view of the entire system. Jack's choice to use a sequence diagram later helped cope with the introduction of an additional entity, the tester, into the system. However, such diagrams typically represent only a single execution path and could not be naturally extended to meet the evolving need to model iterative and conditional behavior.

The textual representation chosen by AI, on the other hand, was more malleable and was complemented by borrowing idioms from familiar representations. The series of steps seemed sufficient for sequential code, but iterative behavior demanded the representation of control flow, and the arrow notation of flow-charts was promptly borrowed. Flow-charts, however, bloat under complex control flows, and the more concise syntactic elements of a programming language were quickly used.

In this case, identifying the connections between the diagrams is relatively straightforward since they are collocated on the same canvas. It is relatively easy for a person to look at all three at the same time. If the diagrams were created in an electronic framework that uses distinct canvases or on multiple physical canvases, there would be nothing to indicate to a casual observer that more information is present in another diagram or that all diagrams should be examined together.

3.2.3 Diverging from UML to work with custom levels of structure

Everyday experience tells us that the hand-drawn diagrams created early in the design process typically appear less aesthetic and organized than their finalized versions, especially electronic ones. In some cases the difference is purely aesthetic and the diagram can be rearranged and re-rendered without actually changing the structure [14]. Other sources [22] suggest that earlier models are simply less complete and omit content and connections outside the current focus; the missing details are incrementally added later. While I witnessed many examples of less aesthetic diagrams and of less complete ones, I also observed early diagrams which appeared visually different because of a choice to use a different level of structure.

Starting with unstructured representations

Notational standards such as UML typically define several diagram types, each with its own primitives. Software tools that support them often require a user to specify the diagram type in advance and only allow the use of the appropriate primitives. However, as we have seen in the previous examples, an early commitment to a diagram type is not always practical. In fact, teams may initially work at such a simplistic level that no diagram type is appropriate, since it would incur significant costs and may constrain them with irrelevant structural details.

The most typical example of this behavior occurred as teams began working on a new canvas and started brainstorming a homogenous set of entities. For example, if team A was forced to use a class diagram when constructing the domain model of Fig. 3.8, the cost of surrounding each new entity in a box might have been marginal. The real cost would have come from constraining the ability to modify and experiment as certain entities turned out not to be classes but rather properties or methods.

It appears that when teams expected their design to evolve, they tended to prefer simpler representations that imposed less constraints. Often, they chose to use text in the form of lists or scattered elements. As the design evolved, relations and annotations were gradually added. In a few cases, the teams used this opportunity to complete the diagram towards UML. For instance, team F took separate diagrams conveying a list of actors and a list of activities, placed them adjacently, and connected them to form a use-case diagram, as can be seen in Fig. 3.11.

In most cases, however, teams continued to diverge from UML. They represented the added information by repeatedly using improvised notations, as in the case of team A, or borrowed idioms from other notational standards, such as the inheritance arrows and relation cardinality annotations from UML class diagrams. The repeated use constrained the representation and implicitly added some structure to the diagram. This structure might eventually help in the interpretation of the diagram even if the notations are unclear, as one could likely do with team A's domain model from Fig. 3.8.

When reuse was limited, and in particular when certain notations had only a single instance, we were often left with less structured and consistent representations that were more difficult to interpret. For example, Fig. 3.12 depicts a diagram by team F, which was working on the medical system. The diagram primarily lists entities, but it also conveys several unclear relations. The relations between `role` and the several descriptive entities to its immediate right use the same notation and could thus perhaps be interpreted as examples or instances of the same concept even before we consider the actual text of these entities to verify this assumption. On the other hand, the relationship between `patient record` and `treatment plan` is unique in this diagram and therefore cannot be interpreted with confidence without contextual knowledge from the actual session.

Even if some notations are reused, however, the benefits can be offset if the emergent structure becomes too complex. Consider Fig. 3.13, which conveys a diagram of objects from team G, who were also working on the same problem. In this diagram, it appears that the team initially brainstormed entities

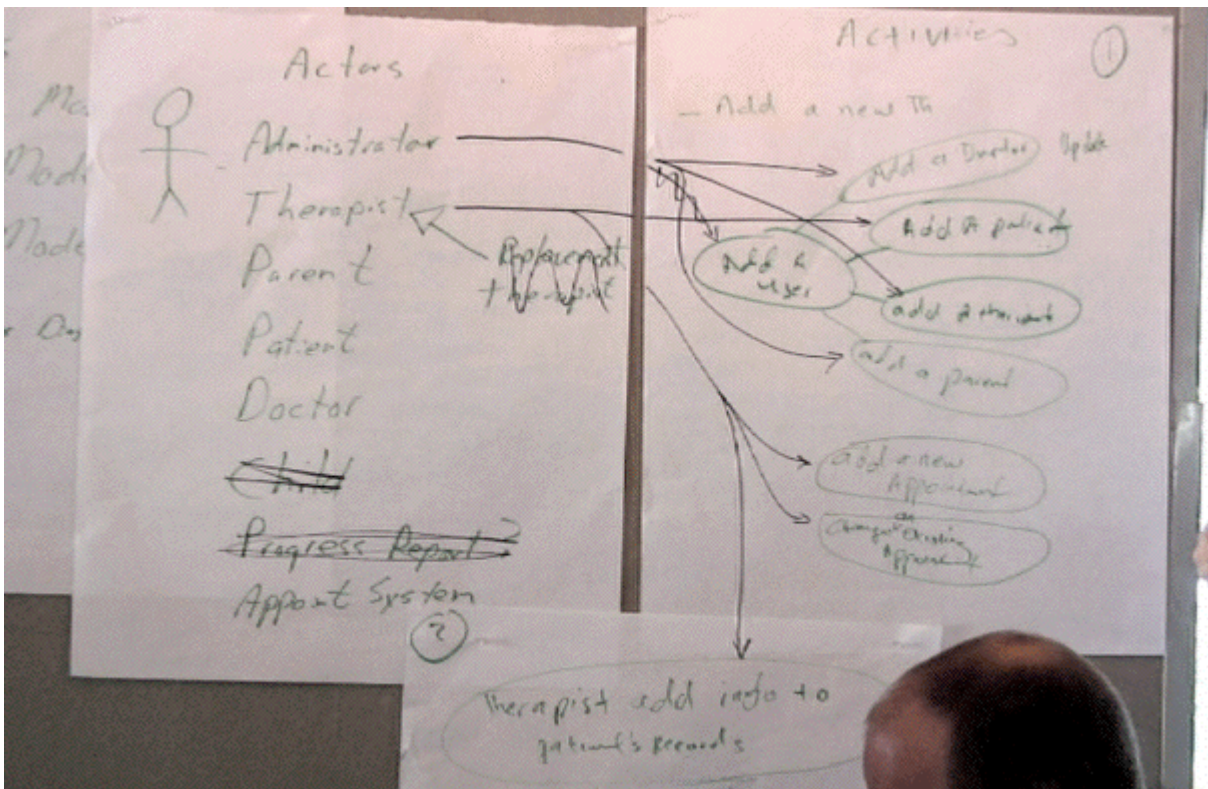


Figure 3.11: A use case diagram composed by team F

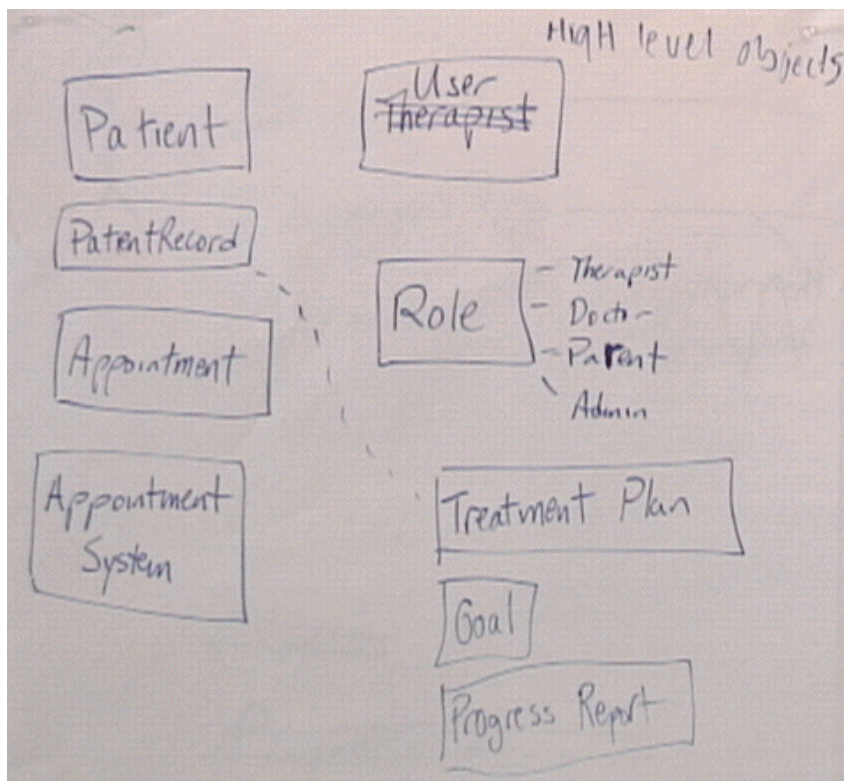


Figure 3.12: Connector notations in an object diagram of team F

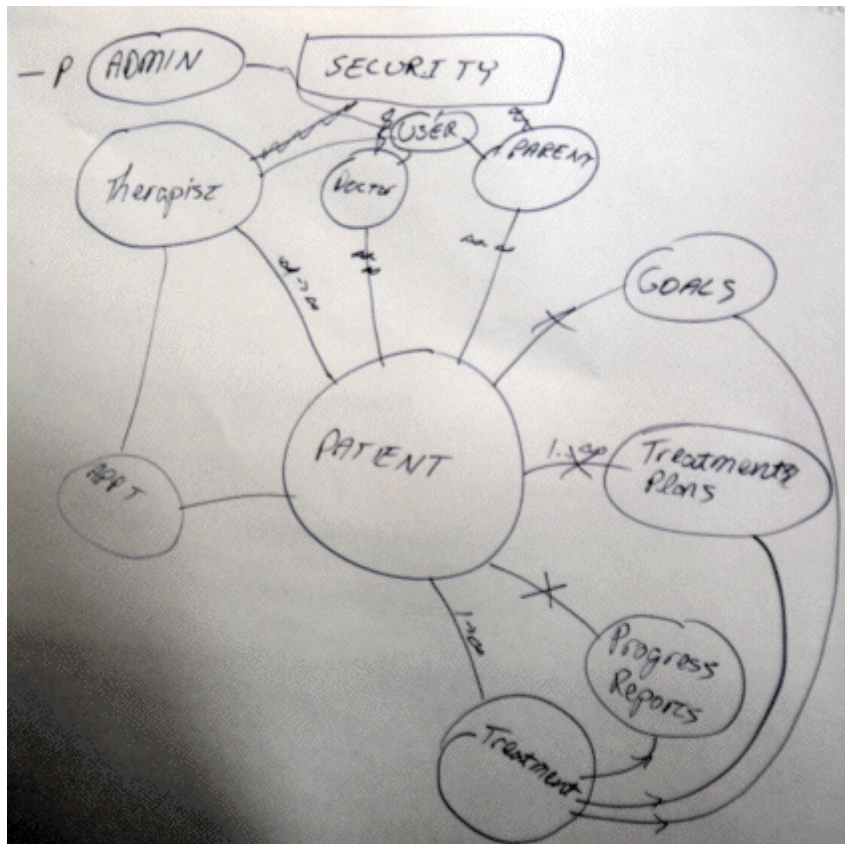


Figure 3.13: Connector notations in an object diagram of team G

Working with more structure

Based on casual observations, one may expect that freeform design sketches will always have equal or less structure than models adhering to standard notations. After all, these notations are often designed to accommodate several layers of complex information. While this may generally be true for UML class diagrams and sequence diagrams, use-case diagrams revealed an opposite phenomenon, where seemingly improvised non-conforming notations actually added structure rather than omitted it.

At its core, a use-case diagram (UCD) is a bipartite graph that matches a set of actors with a set of use-cases,¹ thus offering a straightforward way to represent this many-to-many relationship. However, to identify the UCs associated with particular actors, or the actors associated with particular UCs, one must trace all outgoing edges. This incurs significant cognitive effort since the UCDs for large or dense relations tend to become cluttered [25].

While all but one team (A) in my study explicitly listed actors and UCs or activities, only three of them (C, F, G) drew actual UCDs. All other teams, as well as team C after it had already created a standard UCD, preferred to connect the actors and UCs using the more structured tabular, textual, and numerical forms depicted in Fig. 3.16.

It appears that for the two information system problems, teams tended to partition the set of use cases by an associated primary actor. This partition effectively created a one-to-many relationship that could receive little benefit from the bipartite representation and yet could become difficult to follow. Teams thus preferred to represent this partition in structured textual representations, which are more organized and carry lower cognitive demands but cannot be used for many-to-many relationships.

Use-case diagrams are particularly susceptible to replacement by more structured representations because their inherent structure is so limited. In contrast, class- and sequence- diagrams are very structured, and are sometimes replaced by alternate notations, mostly in the opposite direction, of less structure.

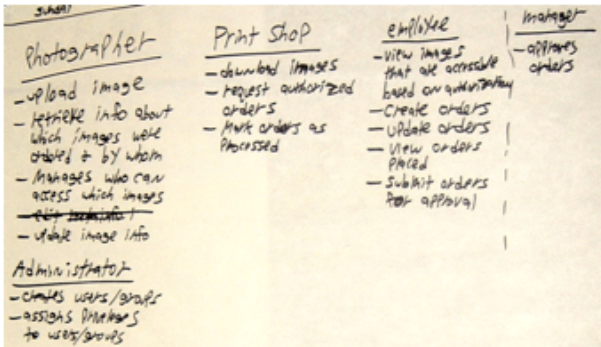
Nevertheless, even these diagrams types may occasionally be replaced by more structured representation. For example, flowcharts, pseudo-code and actual source code are more structured than sequence diagrams because they inherently support complex conditions, control flow, and reuse behavior that is not natural to sequences; this may explain their use in the earlier example of the behavioral model. Similarly, class diagrams are limited in describing the contents of each individual class since they simply present a list of members. We have occasionally seen teams devoting a separate diagram to the contents of a specific class, giving them more freedom in specifying and manipulating its members. Some teams even created bins within the representation of a class into which members were sorted, effectively creating another level of abstraction within the class.

Finally, note that the representation of Fig. 3.16(d), while appearing the most aesthetic, also presents the greatest challenge due to delocalization. A reader examining the left canvas without seeing the right canvas may not be aware of the fact that the roles and activities are related. The arrows in the other diagrams make this clearer but at the cost of reduced aesthetics and organization.

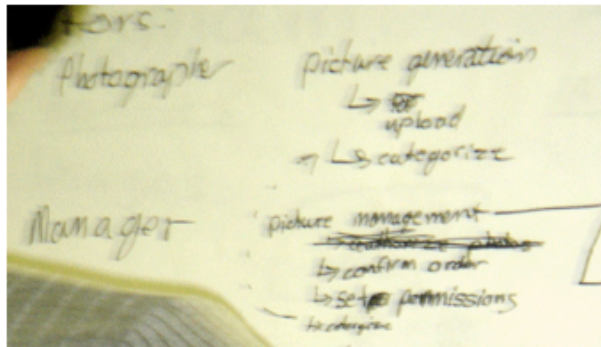
3.2.4 Summary – Collaboration on individual artifacts

The common thread of the observations presented so far is that the design process is structured by the team's unfolding understanding of their problem and solution rather than by the formalism they use for the solution. This allows the team to capture the results of their problem-solving process in whatever order that it happens to take and tackle issues in the order they choose, often using sketches as a short-term memory. They strive to minimize cognitive load by structuring data and increasing the locality of

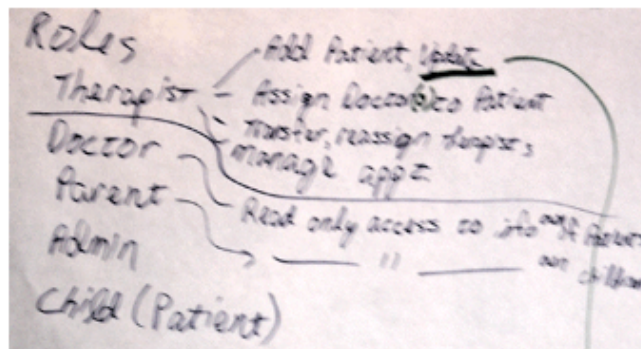
¹Another layer of information connects use-cases to represent extensions and dependencies, but I rarely saw its use.



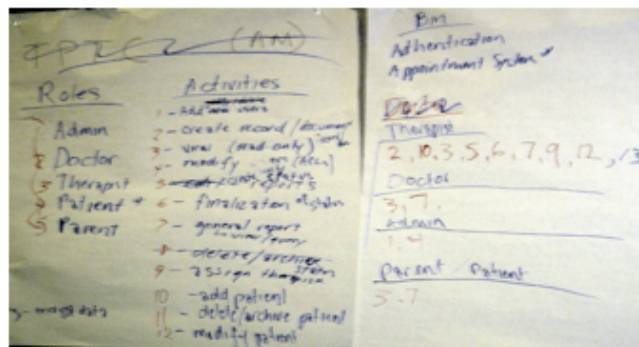
(a) Team B



(b) Team C



(c) Team G



(d) Team E

Figure 3.16: Alternative use case diagrams

relevant information. Forcing them to focus on specific notations and diagram types would clash with the way they work.

More importantly, to effectively address ad-hoc needs, teams seem willing to create artifacts with certain inconsistencies and ambiguous notations at the cost of making them incomprehensible to outsiders. Interpreting these representations would require familiarity with the context of their creation, unlike UML or other documentation-oriented formalisms.

Also note that even when working with few canvases, designers frequently have to use multiple diagrams and representations. If these representations are physically remote, a reader examining one canvas may not be aware of information in the related diagrams on the other canvases.

3.3 Results: Representing heterogeneous information

In the previous section we focused primarily on individual diagrams that diverge from UML notation or use alternate representations. Individual diagrams, however, tell only part of the story. As the teams' understanding of the problem and solution evolves, their knowledge and designs consists of different types of information. To fully understand how designers represent these designs, we must investigate how they deal with this heterogeneity.

3.3.1 Using independent diagrams

Since software designs cover multiple facets like structure and behavior, two-dimensional notational standards like UML typically consist of multiple diagram types [37]. The UML standard dictates that each diagram will use the notation of exactly one diagram type and restricts the use of foreign annotations. These restrictions are usually followed in formal design documents, and most CASE tools enforce them by providing a separate drawing canvas for each diagram, and offering only the drawing primitives of the chosen diagram type.

DesignFest teams, however, use physical mediums and are not forced to produce a formal design document. They are therefore free to violate this restriction. Nevertheless, it appears that initially they did try to adhere to the standard,

The ability to increase the spatial proximity between diagrams, which is relatively straightforward with paper, appears to aid teams in coping with the heterogeneous nature of the design. For example, in the earlier example of team E which was working on the medical system (Fig. 3.5), they had to consider the parts of the user interface visible to the user as well as the activities behind the scenes. This led them to work concurrently on the sequence diagram and the UI diagram, which were placed on separate but adjacent canvases.

When smaller diagrams were required and the use of separate large canvases for each diagram was impractical, teams appeared to relax the restriction a little, allowing diagrams of different types to reside on the same canvas. For example, we have seen team E using the area below the architectural diagram to create the UI diagram in order to conserve space.

Teams may also intentionally bring multiple small diagrams together onto the same canvas in order to increase locality. In the earlier example of team A's behavioral model (Fig. 3.9), the exploration of a scenario resulted in multiple artifacts. While each artifact serves a different role, it must be interpreted in the context of the others and kept consistent with them, thus benefiting from increased locality. A similar multiplicity, involving a sequence diagram and a more complex map of the production line, appeared when the team later explored a more elaborate scenario.

3.3.2 Combining diagrams

While teams generally tried to keep diagrams of different types separate, they occasionally violated UML practices by combining information from different diagram types into a single artifact. This typically involved combining behavioral and structural information.

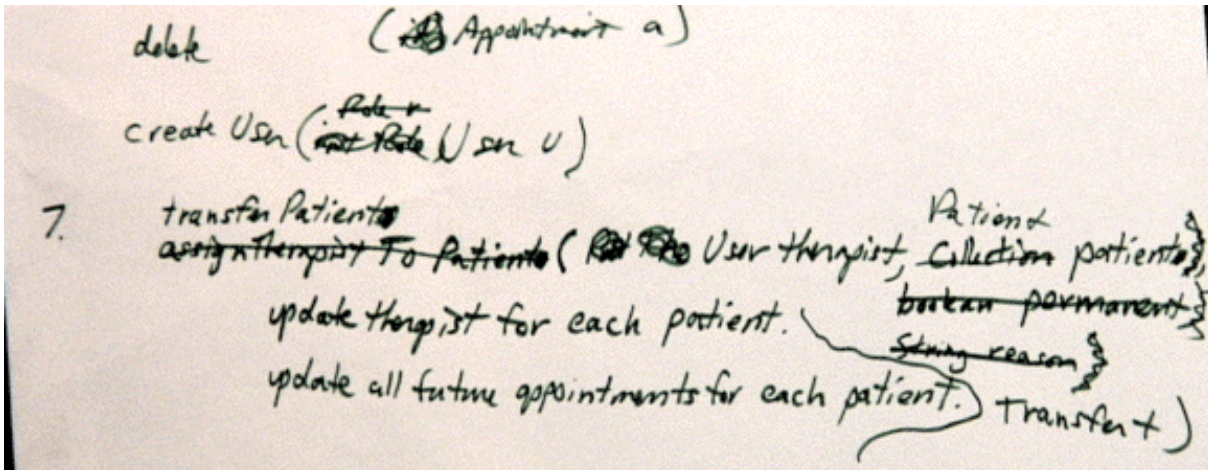


Figure 3.17: Implementations steps included within in a class diagram

For example, team F devoted an entire canvas to the methods of a single business object (Fig. 3.17) in a manner similar to a class diagram, but in addition to listing the method signatures, they also specified implementation details which have no place in such diagrams according to the standard.

A similar phenomenon was observed in the 2006 validation study, when a team working on a tournament management software textually elaborated the steps of a use case and added class-diagram elements to represent related entities (Fig. 3.18). In both cases, the added information is closely coupled to the context of the primary diagram type.

In addition, I frequently observed teams augmenting UML class diagrams with indications of control- and data- flows. For example, in the earlier example of the evolving class diagram by team E from Fig. 3.8, the team created an output entity for a `Treatment report` and added a line from the `Session` class to represent the flow of data used to create it. Similarly, the three outgoing edges from the `Treatment` object in the data model created by team G from Fig. 3.13 appear to represent some form of control or data flow.

One team in the 2005 study used class diagram notation to represent the system architecture and the interactions within it, as depicted in Fig. 3.19.

Other teams introduced external systems and storage mediums into standard class diagram. For example, Team A was creating a class diagram and discussing how the `Simulator` class could be configured. They added a configuration file and a class for loading it, and then they used arrows to model the flow of data from the file to the simulator via the loader class (Fig. 3.20(a)). This hybridization of class structure and data flow is apparently not accidental, as it was repeated in a subsequent finalized version of the diagram (Fig. 3.20(b)).

By combining diagrams or introducing foreign elements, teams increase the locality of information while reducing the clutter, effort, and redundancy that stems from the use of independent diagrams. While these are tangible benefits, this behavior appears to have a more fundamental motivation that is rooted in how designers think about the system and its components. Early design discussions typically revolved around objects rather than classes and referred to their structural and behavioral properties in

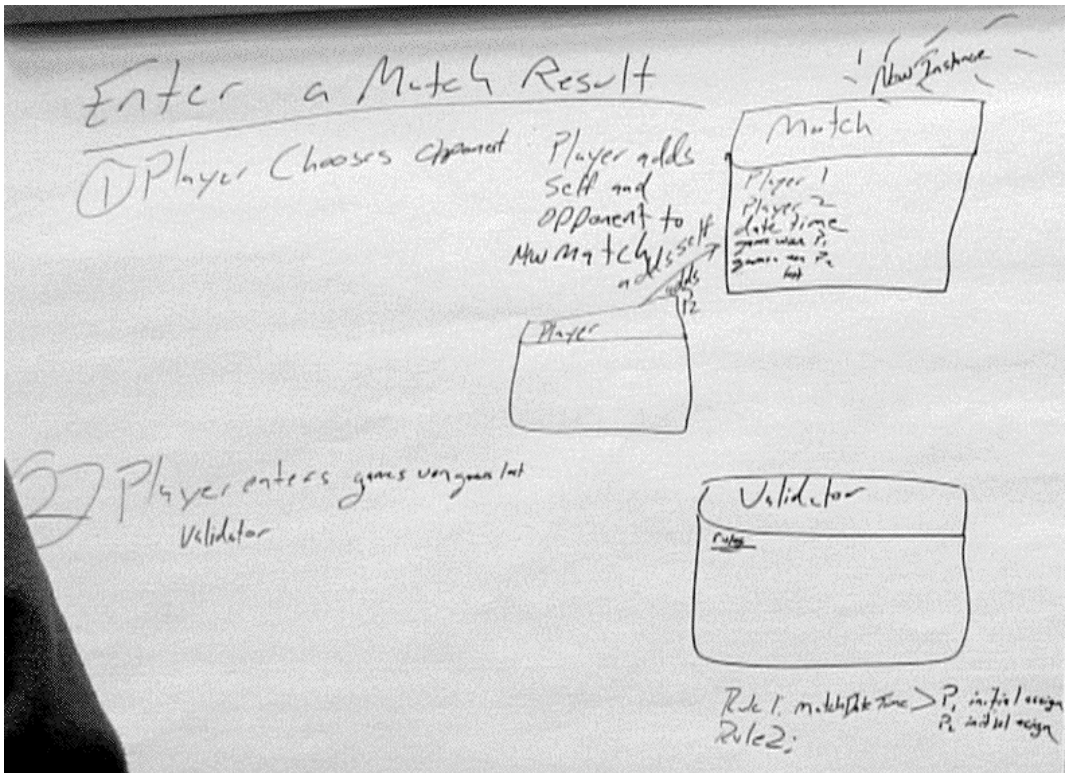


Figure 3.18: A class diagram embedded among use-case steps

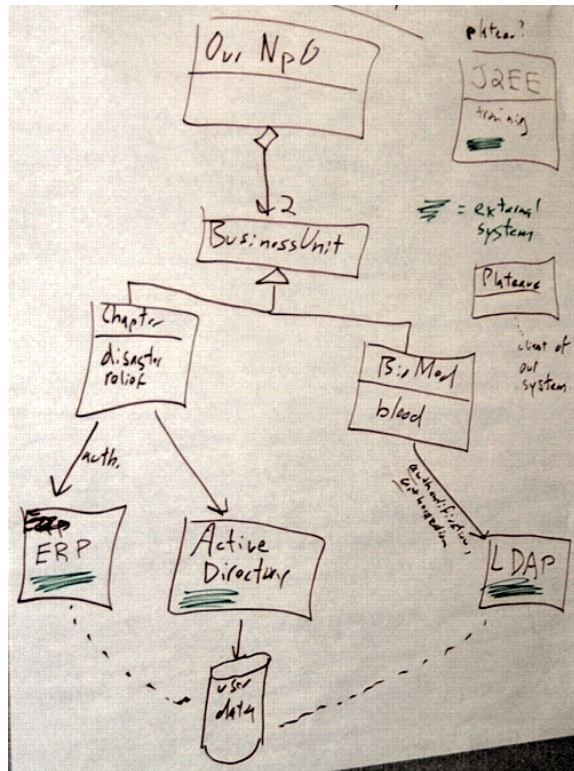
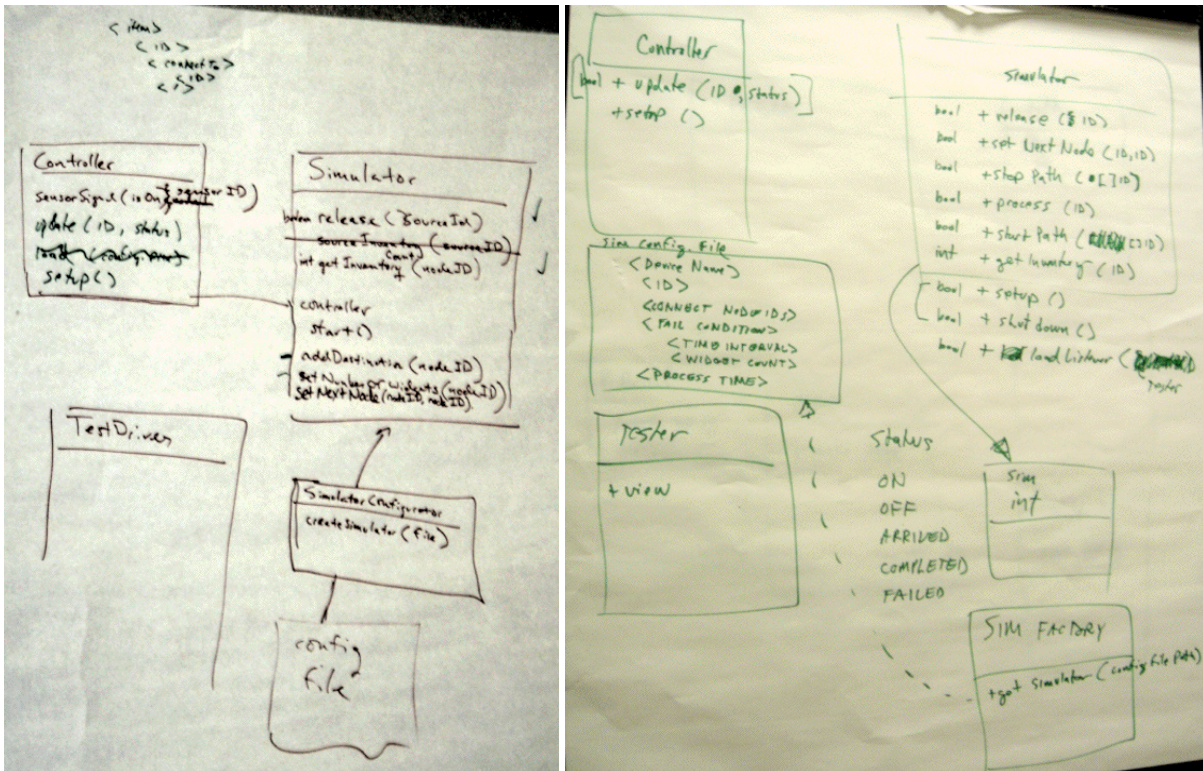


Figure 3.19: Architecture presented in class diagram form



(a) Initial diagram

(b) Later diagram

Figure 3.20: External elements in class diagrams by team A

ways that cannot easily be disentangled. While notations that dictate a separation of these facets offer ways to accurately model the details of each facet without interference from the other, this has limited benefit in these early stages, while the costs of forcing a disentanglement may be significant.

Impact on knowledge awareness

The problem with combined diagrams is that while convenient for modeling, they carry the requirement of eventually being disentangled for presentation and preservation purposes. Once separated, several problems may ensue: First, because each artifact is created in the context of the other, there may be certain dependencies between them and each may individually be hard to interpret. Second, to save effort, the designers may avoid the duplication of information, leaving gaps in one artifact that are readily filled by the other; once separated, each artifact could be incomplete. Third, there is a difficulty keeping the separate artifacts consistent as they continue to evolve. Future maintainers making changes to one diagram may not be aware of the past connection and thus of the need to change the other.

I also argue that some teams may be apprehensive about combining diagrams in this way, but that the above examples demonstrate that designs are naturally multifaceted. Thus, even if a team designs using multiple separate canvases, the implicit connection between each diagram exists and must be preserved.

3.3.3 Introducing peripheral information into diagrams

The tendency to increase the locality of information in the diagram is also evident from the ad-hoc integration of concrete examples, details, and instantiations into the diagram, rather than placement in

external documents or use of annotation standards.

The UML standard, especially as implemented in CASE tools, accommodates peripheral information only in specially marked “notes” that lie in proximity to- or in direct connection with- diagram elements; these constitute a semantically- and visually- separate annotation layer. The last example shows that in collaborative work, the boundaries are not as clear, at least visually: the configuration file with its proposed format in Fig. 3.20(b) looks at first like an integral part of the diagram.

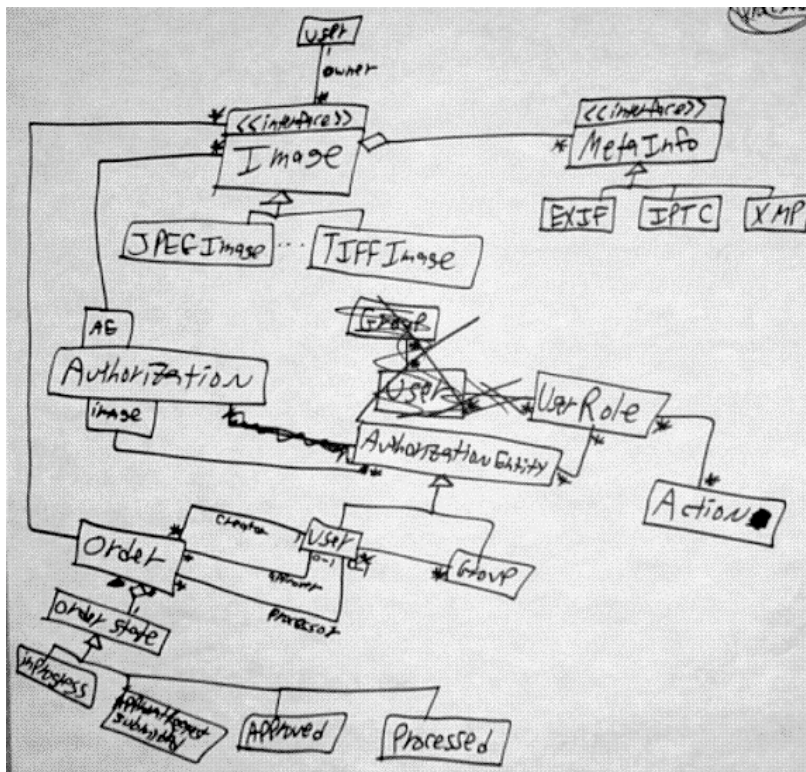


Figure 3.21: Examples added using subclass notation in class diagram by team B

Similarly, as team B was working on the image shop problem, they integrated examples of images, metaInfo types, and order states into the class diagram (Fig. 3.21), even though it appears that these were not intended to eventually become classes.

Concrete examples, not always textual, were often added in an ad-hoc manner while discussing a recent idea or addition. For instance, during the initial discussion about the configuration files, a member of team A wrote a concrete example of its format on the top of the page (Fig. 3.20(a)).

Similarly, in response to questions about security, a member of team C drew a small incomplete class diagram (Fig. 3.22). As more questions were raised about the data relations, he added a sample record next to it.

While examples were typically placed on the same canvas as the entities they refer to, we saw limited spatial proximity or explicit connections between them within the canvas. For instance, the data format appearing at the top of Fig. 3.20(a) is visibly remote from the configuration file at the bottom of the figure. In fact, external observers might consider it unrelated to the diagram or relate it to the methods of the controller as both use a distinct color. It appears that explicit proximity or connection, as practiced by UML, was not necessary since the example was given in a specific context. When creating documentation-oriented artifacts, teams appeared to explicitly capture these associations, as is evident by the relocation of the format into the configuration file (Fig. 3.20(b)).

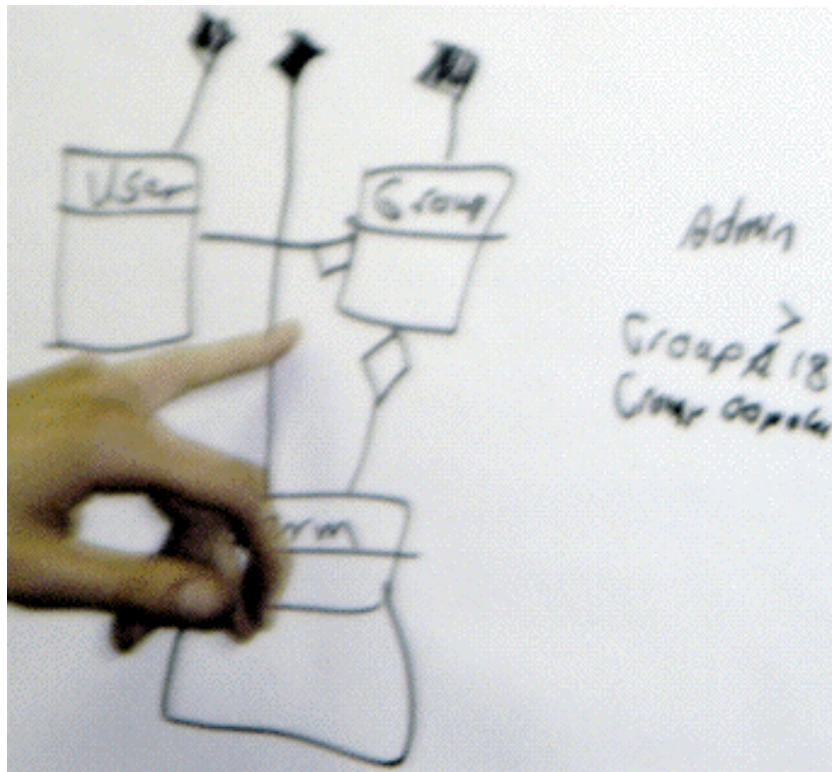


Figure 3.22: Sample record added to class diagram by team C

Note that important context-sensitive design information is not always fully captured in writing. It may sometimes be captured as more of a placeholder or reminder of other information and be difficult to interpret without the contextual information. For example, only meeting participants will recognize that the annotations on the methods of the simulator in Fig. 3.20(a) convey that some methods are private while others correspond to a certain scenario. Similarly, the meaning of the crosses or the arrows on the edges in the earlier example of Fig. 3.13 is unclear but likely denote some concept discussed in the session.

I believe that the inclusion of examples within a diagram is often done to ensure awareness. Designers may suspect that if the example is kept separate, there would be nothing to lead someone examining the design diagram to suspect that an example or additional information is available.

Finally, note that as we have seen in the previous study, many notions and ideas in the course of discussion are not expressed with permanent visual markings but rather created with hand movements or even sketched in the air or with a capped pen over the board. All evidence of these gestures is lost in the final diagram, even though they potentially convey important annotations and examples. Designers often appeared to use this mode of communication to avoid cluttering the paper, so it is possible that interaction would have been different over a dry-erase whiteboard or an electronic medium. Nevertheless, this behavior demonstrates that pertinent design information may not be explicitly captured in an artifact but rather expressed in its context.

3.4 Results: Dependencies between diagrams

The results presented so far show that even when teams are given freedom in their choice of representation, their designs are still dispersed over multiple diagrams. In this section we focus on the dependencies

between these diagrams, how they change over time, and how teams cope with them.

3.4.1 Diagram evolution across canvases

In examining the representations of individual artifacts, we saw diagrams evolve and change their type or focus in response to ad-hoc design needs. As the diagram becomes more visually dense, however, it can no longer evolve “in place”. Especially when using physical mediums, larger segments cannot be manipulated with ease. Meanwhile, striking out contents is detrimental to the diagram’s aesthetics and may overload short-term memory if material is recreated. Thus, and as we have seen in the first study, a single design or final diagram can sometimes evolve over several versions, each on a different canvas.

The model of continuous evolution towards complete UML described in the literature [23,88,14] implies a monotonically increasing shift towards completeness in content and notation. When the evolution is not in-place, this model implies that each new version should convey at least the same information as its predecessor (except for intentional revisions), thus rendering all previous versions redundant. However, reproducing the contents requires a significant mental effort, which designers are likely to try and minimize. An issue of significant concern, therefore, is whether they would perform this mental task and copy all details or spread the design over multiple incomplete versions, potentially leading to a situation where the final version might not stand on its own.

Example: Evolution of data model

Let us consider one such situation, occurring as team E continued working on the class diagram for its medical information system (Fig. 3.8). As a result of the process by which the diagram evolved, it had become quite cluttered, with key entities, such as `Patient`, still represented as forms rather than as classes.

Craig described the diagram as “a mess” and suggested that they clean it up; someone suggested summarizing and Craig agreed, saying they should also capture the relations. They proceeded to create a new diagram with a clearer and more spacious layout. It began as an entity-relation diagram but became a class diagram as inheritance was added once again. In creating the new diagram, the team fell into a pattern, depicted in Fig. 3.23: Craig would turn his body or walk towards the original diagram and identify an important entity or relation. A discussion would ensue, followed by rendering a version on the new diagram, after which the pattern repeated itself.

The final version of the diagram, depicted on the left canvas of Fig. 3.24, conforms with prior research which suggests that artifacts are often recreated with equivalent content to improve aesthetics or to migrate into an electronic tool [22,16], or with less content to highlight specific details [23]. However, the first study suggested that the mundane activity of recreating artifacts serves not only aesthetic purposes, but also offers a chance to inspect, reconsider, and improve past decisions, resulting in different content. What, then, is the relation between the old and new diagrams in the work of team E, and to what degree are aesthetics the primary difference between them?

As cleanup progressed with elements transferred to the new diagram, certain decisions were revisited or expanded. For example, at the end of a long discussion, the one-to-many relation of `Responsibility` between `Therapist` and `Session` from the original diagram became a many-to-many relation between `Therapist` and `Treatment plan` in the new one and received several associated properties. Afterwards, the team created a relation between `Plan` and `Appointment` to indicate that a plan can consist of both appointments and sessions rather than solely of the more specific sessions. Later, the flow arrow from `Session` to `Treatment report` was replaced by a one-to-one relation. A discussion on a reporting infrastructure ensued, and though never finished, a general `Query` class was added to represent

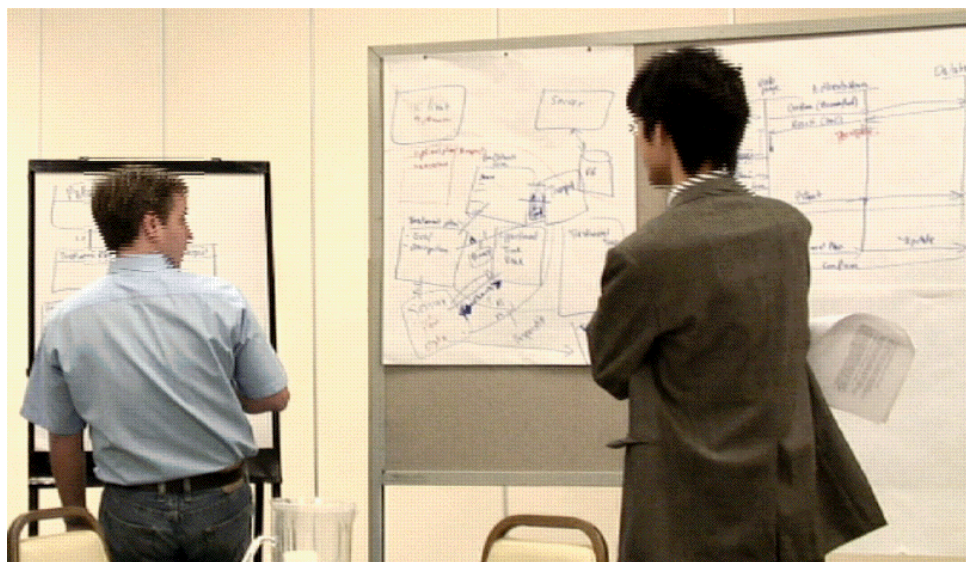


Figure 3.23: Team E recreating the data model

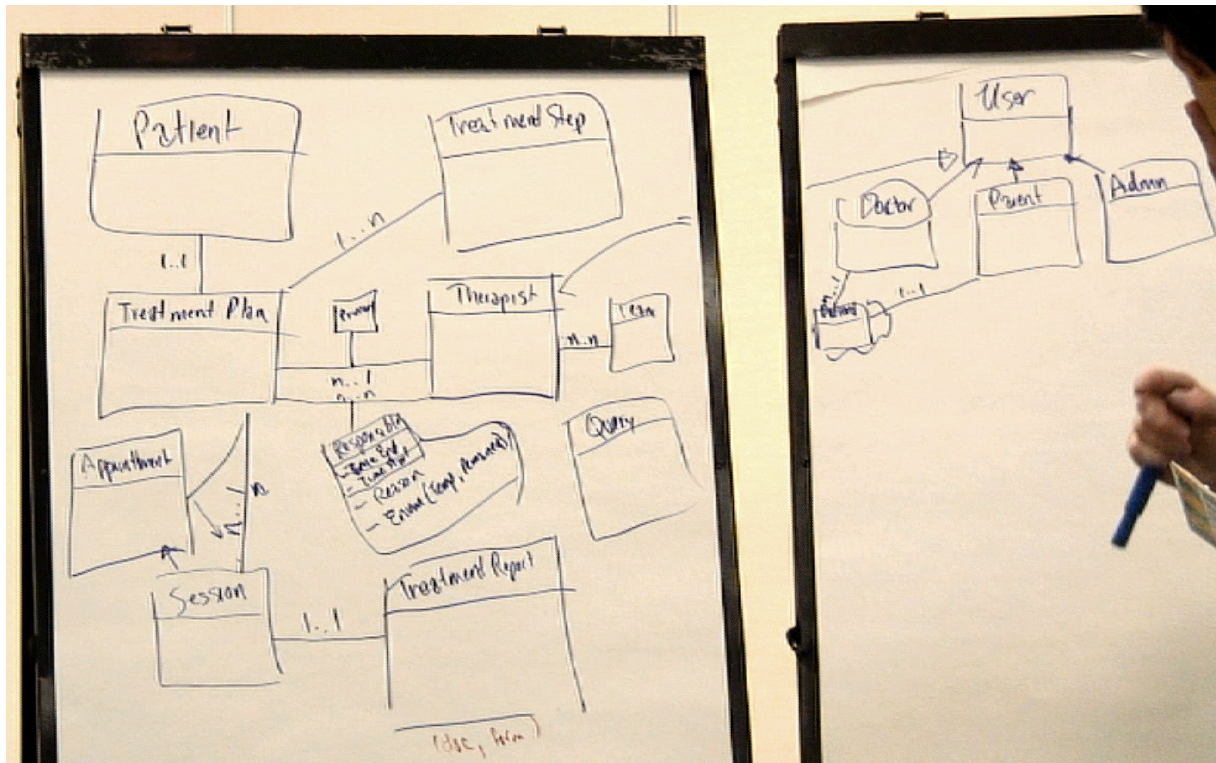


Figure 3.24: Revised data model by team E

this functionality.

This example offers anecdotal evidence that designers may conserve effort by avoiding the replication of some pertinent information, such as the fields of the `Treatment plan` class, to the new diagram. Grasping the complete design thus requires aggregating information from both versions, which is made difficult by the differences between them. It is also just one of the many situations I encountered in which substantial and often conflicting differences exist between the earlier version of the artifact and the revised one. The difference may not be noticed without careful comparison of both diagrams. Determining the final decision would require knowledge of which diagram was newer, and determining the rationale behind them involves recalling the discussions at the time of the transition. A greater problem, perhaps, is that a reader examining one version may not be aware of the existence of other versions.

Evolving through multiple revisions

Many diagrams evolved through more than two revisions. In general, relatively early after teams began working on a new version, they seemed to determine the magnitude of differences from the previous version. When it appeared that only minimal changes would be necessary, they tended to split into smaller groups and produce an aesthetic and complete finalized version. Otherwise, the team remained cohesive, and less attention was paid to completeness and aesthetics, perhaps in anticipation of yet another revision. Nevertheless, the early determination was not always accurate, resulting in significant decisions being made at the subgroup or individual level.

3.4.2 Noncontiguous artifact evolution

The previous example demonstrated a localized and contiguous evolution from one version of the artifact of focus to another version of the same artifact. Earlier, we have seen interweaving of work on multiple diagrams which evolve together, as when team E was creating the data model of Fig. 3.5 while working on the sequence diagram, or when team A was working on the behavioral model using multiple representations (Fig. 3.9). In these situations, there was still a locality in time and space and no additional artifacts were used or changed in the interim.

Such locality, however, is not always the case since, as we have seen, ad-hoc needs often divert discussions in different directions. Timelines constructed for the observed sessions show that teams tended to work for a while on one primary artifact and then shifted their attention to another. An artifact may be abandoned at some point, only to be recalled at a later point and be discussed, referenced, copied, or continued, on the same canvas or on a different one.

Team A was working on the class diagram of Fig. 3.20(a), which was still located on the flipchart. In the course of one long discussion, the team's assumptions came up, and the focus changed to a list of assumptions posted earlier on a different posterboard. After adding assumptions, they came back to the class diagram for another ten minutes, after which they flipped the chart to an earlier unrelated diagram, and worked on it for a while, occasionally also working on the assumptions list. After returning to the class diagram, they began making the changes in green (controller methods, data format, and markings near simulator methods). They posted the page and turned to working on scenarios for an hour. Only towards the end of the session did they begin creating the revised version of Fig. 3.20(b).

This example shows that individual diagrams do not evolve in a vacuum. Instead, each change takes place in the context of the current state of the design at that time, which could be captured in artifacts that share no obvious connections. For example, certain design decisions were made in the context of the original assumptions list, while others were made in the context of the new ones. The artifacts do not capture these temporal dependencies, potentially presenting a significant challenge for the eventual interpretation of the meeting products.

3.4.3 Dependencies on multiple diagrams

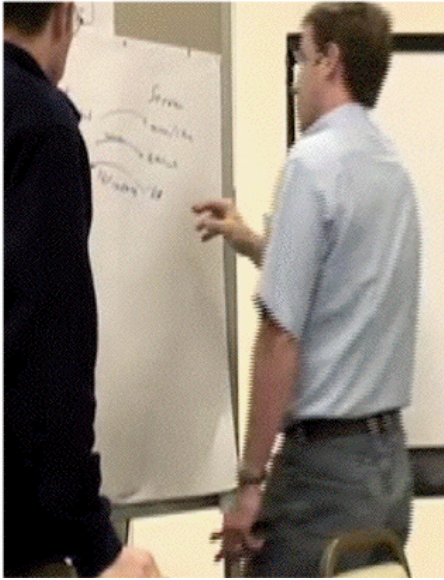
The network of dependencies between artifacts is further complicated by the flow of design information from multiple sources, typically earlier diagrams, into a single “sink”, often the current artifact of focus.

Having finished discussing reports, the E team turned to the relations between the entities representing actors in the system. They referred to a small diagram, created very early in the session, which described inheritance relations among `USER` entities, including ones that did not appear in the original diagram of Fig. 3.8. These entities and relations were then copied into a separate page, and data cardinality connections were added. As can be seen in Fig. 3.24, this diagram was then placed adjacently to the right of the main diagram and a connection was made, effectively integrating the new part into the main diagram.

The additions to the class diagram are compliant with UML, but much of the design rationale behind them lies in the context of the differences from the original hierarchy and their integration into the larger diagram.

Another example of how multiple artifacts are used, integrated, and discussed, occurs towards the end of the E team's session:

With time running out, the team decided to capture the high-level architecture of their system, which they initially abandoned very early in the session. In constructing this new diagram, they had prolonged



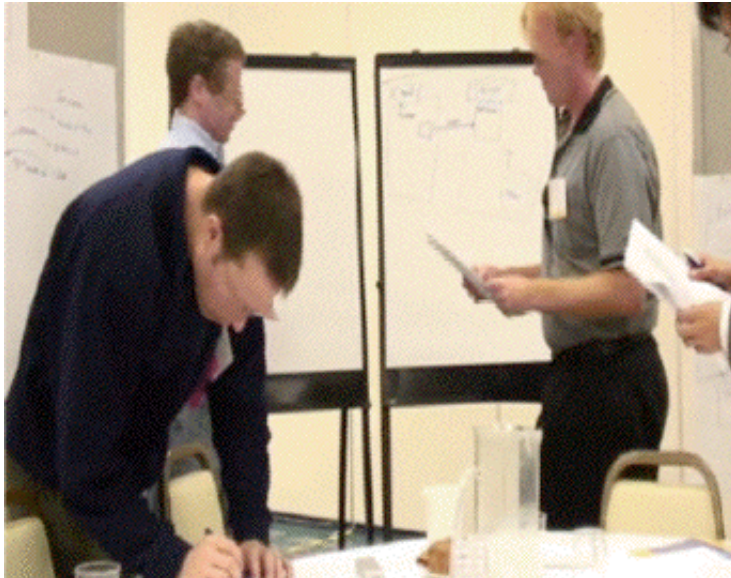
(a)



(b)



(c)



(d)

Figure 3.25: Team E building an architectural model

discussions and referred to several artifacts, shown in Fig. 3.25, including: (a) a simple model of client and server responsibilities, (b) the sequence diagram, (c) the architectural sketch on the same canvas as the discarded original model, and (d) the problem-specification document with its notes. The resulting diagram adds little beyond the original sketch, but many architectural decisions were made verbally in the context of the referenced artifacts but simply not written down as time ran out.

The above examples show that even though individuals maintain much design knowledge in their heads and could implicitly apply it to the current diagram, they instead explicitly reference it in existing materials. A likely explanation for this phenomenon is the need to ground the knowledge and offer context to the upcoming discussion while ensuring that that it is shared by all participants. In addition, since I frequently saw individuals examining artifacts before speaking, it is possible that peer pressure leads designers to “check their sources” before contributing.

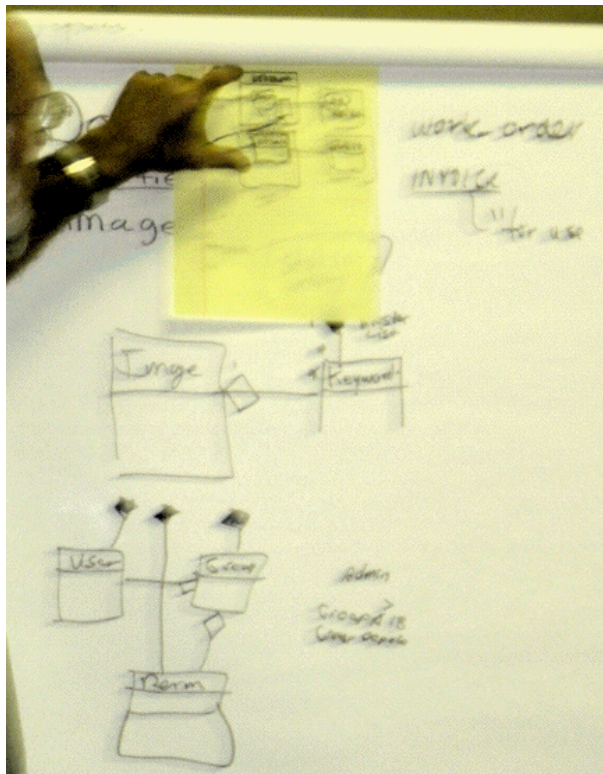
3.4.4 Coping with multiple artifacts

The dependencies between design artifacts are a known challenge to all designers but may be particularly problematic for collaborative teams working with physical mediums. The need to make artifacts visible to everyone requires the use of large shared drawing spaces and results in the creation of large scale artifacts. The team’s workspace is usually limited, allowing them to keep only a limited working subset visible at all times [25]. Even the artifacts in this workspace are typically spread around the design environment, preventing designers from seeing all of them at once.

For these reasons, when teams needed to continuously focus on multiple artifacts, they tried to increase the physical locality. If the diagrams were not on the same canvas, then they were moved around and placed next to each other. For instance, I observed team E place the UI and sequence diagrams next to one another, as elements in the former were entry points into the latter. Similarly, I observed team F place lists of actors and activities next to one another and compose them into a use-case diagram (Fig. 3.11).

When the need to focus on multiple artifacts was transient, diagrams were often held in proximity without being physically attached. This was especially common when recreating a new version of a diagram from an earlier model or a personal note (Fig. 3.26), after which the original was often removed from the working set. There was limited need to affix the diagrams because the old version could always be brought back if necessary (e.g., Fig. 3.25(c)). There are situations, however, where diagrams were held in proximity in the course of conveying ideas. For example, when team F worked on a scenario involving user actions, a designer would often hold up a sketch of a web form and a sequence diagram, demonstrate a user action on some widget, and then continue to follow the sequence of actions in the sequence diagram until the output was produced back on the form. In these situations, no permanent evidence was left of the connection between the artifacts.

As we have seen, the content of one artifact can affect the discussion or content of another, even if they do not share mutual entities. While individuals may possess the relevant knowledge in their minds, they appear to reexamine the artifacts before speaking and later refer to them explicitly. Unfortunately, the effort and time involved in moving and arranging artifacts around the workspace prohibited the establishment of physical proximity in many of these cases and forced teams to repeatedly switch their attention between different parts of the design area. This was particularly evident for shorter references to materials posted around the work area and especially when individuals, rather than the entire team, were considering the materials or referring to their personal notes or the given documentation. The only physical manifestation of these public references were gestures and pointing, while the only clue of private references was typically a glance or a change in head position.



(a) Team C



(b) Team F

Figure 3.26: Holding diagrams to increase locality

3.5 Discussion

3.5.1 Overview and contributions

My observations suggest that in choosing representations, teams make deliberate and intentional choices to diverge from standard notations or borrow idioms from other notations. While often benefiting from lower physical and cognitive efforts, teams make these choices primarily in response to the immediate needs arising from their evolving understanding of the problem and solution. Improvised freehand notations are not only easier or faster to use but also offer a degree of freedom in selecting structure and interpretation that may better fit and evolve with the design than a fixed notation. Similarly, information is dispersed and laid out in ways that not only increase the locality of related entities but that also facilitate a more natural representation of complex heterogeneous information.

While these representational choices allow teams to focus on creativity, the resulting artifacts and the relations between them may be less intuitive, exact, and complete. Teams try to compensate for this by using additional but transient communication mediums, such as voice and gestures, and by relying on memory and contextual cues. Other stakeholders, however, will face significant difficulties in subsequently interpreting and using the resulting diagrams as documentation or implementation artifacts. In fact, knowledge of contextual and historical information of the experiences shared by the meeting's participants may be necessary even for diagrams that teams have explicitly prepared for presentation.

The importance of these findings is in improving our understanding of collaborative design meetings and in guiding the development of new tools. One contribution of my work is in demonstrating the priority given to creative activities and that some of the noncompliant artifacts, often treated as peripheral by existing tools and approaches, actually play a major role in problem-solving and in communicating about the design. Second, I show several typical ways in which collaborative design unfolds and ad-hoc needs lead away from standard notations. A third contribution is in highlighting the importance of contextual and historical information in the work of design teams and in its manifestation into the products, thus prompting efforts to preserve it. Such efforts may be more effective than attempts to support or perfect any notational standard since teams require a flexibility that may not be possible with a fixed representation.

3.5.2 Use of UML as an idiom

Although the presentation here is focused on divergences from UML, it is important to note that the representations of most artifacts are still based on that formalism and often comply with it. This is particularly evident in finalized versions created for presentation: UML is apparently considered appropriate for documenting and communicating designs.

Testimony to its communicative qualities is offered by the fact that some of its notations become idioms that are applied almost automatically, even outside the expected context. For example, the inheritance notation of class diagrams is so well recognized that we have frequently seen it used to indicate generalization and specialization relations among other entities, such as actors in use-case diagrams. The use of inheritance to convey examples in the class diagram of Fig. 3.21 is another possible example of this phenomenon. These idioms even make their way into architectural diagrams. For example, in a freeform architecture diagram created by another team working on the image shop problem (Fig. 3.19), we can see the UML notations for inheritance and aggregation used in a diagram that primarily conveys components and the interaction between them.

In fact, after examining the plethora of diagrams in three years of *DesignFest* events, I raise the possibility that the use of UML in the earlier and more creative stages of collaborative OOD, before

final diagrams are created, is not a real use of the standard or of a consistent subset. Rather, it appears to be a use of freeform notations that borrows and utilizes several well-recognized UML constructs as idioms. This interpretation may offer some explanation for the use of only a handful of UML diagram types and of a very limited subset of their available primitives, even though many of the participants were very experienced UML modelers.

When not using the idiomatic constructs, representations tended to devolve into the box-and-arrow diagrams observed in non-OO settings [16]. This presented only a limited difficulty during the meeting because interpretation was implied by the context of the conversation. However, it presented a problem for external readers who lack syntactic or contextual cues to interpret them.

It is important to clarify that I do not consider my findings about UML to be indicative of specific weaknesses of the UML standard compared to other potential notations. My observations are not merely the result of studying a particularly problematic formalism. While they highlight inadequacies of UML in supporting collaborative design, UML is primarily a specification- and documentation- oriented notational standard.

The important implication of my observations is that it is not clear that any current or future fixed standard with these goals would be flexible enough. The same factors and ad-hoc needs will likely lead designers to improvise around its restrictions as well. Thus, instead of trying to improve collaborative OOD by attempting to find a perfect formalism, perhaps an investment in tools that are independent of specific notations may be more rewarding.

3.5.3 Divergence from UML

Observed divergences

The collected photographic evidence from my observations at *DesignFest*, of which only some examples were shown here, contains numerous examples of representations used in collaborative OOD. Many of these representations significantly diverge from UML, and some consist of multiple diagrams with dependencies between them. However, my goal was not to merely confirm the casual observations and everyday knowledge of their existence, nor was it to try to catalogue them or to elicit quantitative information, which would have little use in these restricted and unique small settings. Rather, in initiating this study, I set out to understand the representations used in collaborative OOD, the reasons for their creation, and their implications for tools. In the preceding result sections, I presented representative examples of behaviors that, I believe, shed much light on these questions.

When I set out to explore the representations, I expected artifacts to primarily be examples of incomplete diagrams with accidental freehand notations that could eventually be evolved into complete UML models, as described in the literature [23]. I also expected each diagram to be relatively independent, which would facilitate the transition to UML. Perhaps because of the less restrictive settings of my study, the actual data revealed surprising results: While I did see many incomplete diagrams, the representations ended up being too varied to fall under any simple classification and often spanned multiple diagrams. Most importantly, they appeared deliberate rather than accidental.

I have seen many instances in which the divergent representation could not simply be dismissed as early forms of UML or as independent and unrelated freehand annotations. Instead, there appear to be situations in which teams create artifacts that convey the same data as proper UML diagrams but make use of improvised notations to quickly solve immediate problems or capture insights before they are lost. One path for the evolution of such representations occurred when teams began with a less structured representation and then introduced additional notations and structure. A second path is when they chose a representation that could offer more structure or cover more facets of the design than a standard UML

diagram.

In addition, rather than create the portfolio of visually independent diagrams frequently seen in design documents and CASE tool models, teams deliberately created a large, interdependent and seemingly disorganized array of artifacts. Some of these combined multiple diagrams, types, and annotations, while others depended on or overlapped with other artifacts. In addition, there is a complex network of dependencies, connections, and locality between the artifacts, many of them transient and context-sensitive and in some cases subtle. Some of these connections are spatial, involving the location of the artifacts at specific times, while others are temporal or contextual, involving their state or use at given times.

Ad-hoc choice of representation

Existing explanations for the divergence between design sketches and standards such as UML tend to follow two themes. First, early artifacts are mostly incomplete, and can evolve into conformance with the formalism with sufficient effort and guidance [22, 38]. Second, there are criticisms of the standard itself, such as its power of expression, elegance, level of restriction, or approach to organizing data. Thus, designers might “rebel” against it or at least adopt additional notations.

Based on repeated close studies of the videotaped evidence, I propose an explanation for the choice of representation, and the divergence in notation in particular, that is more fundamental and less dependent on the specifics of formalism and settings. Namely, I suggest that while teams are aware of the need to convey their design for future use, and will explicitly work towards that goal in later and visually distinct phases of the design, that is not their primary motivation and concern.

Rather, in the creative phases of the design, both the design process and the representations used to capture it are structured as an ad-hoc response to the team’s unfolding understanding of their problem and solution. The ad-hoc approach allows teams to capture the results of their problem-solving process in whatever order that it happens to take, and tackle issues in the order they choose, often using sketches as a short-term memory.

In respect to representation, ad-hoc choice serves several purposes. One obvious benefit is in allowing teams to minimize the costs in distraction and physical effort that arise from visual activity and in particular from adhering to a complex notational standard. Like previous researchers, I saw teams filtering out unrelated content [22] and skipping aesthetic polishing [14, 71]. I also saw them avoiding the menial chore of copying all content when creating a new version of a diagram, effectively spreading the design across several versions. In addition, I have frequently seen participants using gestures and “air-pens” rather than actually drawing in order to avoid the associated costs of writing and subsequently erasing materials. This was also evident when teams used small and portable canvases or sticky notes that could be temporarily attached to larger sheets and moved about [25].

A second related benefit is that it enables them to localize different types of information related to an issue, thus reducing physical clutter and memory load. Teams tended to increase the physical locality of relevant materials: they shuffled canvases around or placed related diagrams on the same canvas and even integrated different materials into the same diagram. However, this increase in locality is not only physical; it also extends to a reliance on individual and group memory, on the immediate context, and on alternate communication mediums like speech or gesturing. This locality acts as a substitute for the need for specific and well-defined references and notations.

A third, less obvious but more fundamental purpose of ad-hoc choice is that information could be represented at levels of completeness, abstraction, structure, and organization that are best adapted to the team’s current and anticipated needs. In some cases, the available representations could not match the level of structure at which the teams wanted to work. In other cases, by avoiding an early commitment to

a particular representation, teams were able to allow their representation to evolve in ways not possible with a fixed representation.

Limitations of sketch recognition

As described in Sec. 2.1.3, most existing efforts for supporting collaborative OOD focus on using electronic whiteboards as shared drawing spaces, typically using sketch recognition to identify primitives of the notation. These primarily focus on specific UML diagram types, though a recent tablet-based tool [38] supports customizable and domain-specific notations.

Although there is limited information on these tools' success in the field, my observations present two potential difficulties to real-world deployment. First, designers employ a range of improvised and generalized notations that may be difficult to distinguish and associate with the specific notations of a fixed standard with adequate confidence. Second, designers rely on contextual information and on alternate communication mediums such as gestures, which are not available to sketch recognizers, to complement and help interpret the improvised notations.

Thus, while such tools may be very useful when preparing the final UML diagrams for documentation, I argue that they may be significantly less useful in the earlier creative stages.

Impact on interpretation

As I have repeatedly seen throughout this chapter, the priority given to creative design and the attendance to ad-hoc needs comes at a cost: teams create an unedited collection of artifacts with certain dependencies, inconsistencies and ambiguous notations, that may be less comprehensible and useful to outsiders, thus lowering their potential as documentation or implementation artifacts. This poses a problem because sketches as well as archival-quality artifacts from design meetings are subsequently used by developers, who need to understand decisions made in the design meeting [16]. They also frequently need to understand the rationale behind these decisions [50], which is often not documented explicitly [55]. Furthermore, problems in interpreting UML diagrams have been implicated in some software defects [53].

Even some of the explicitly-recreated and finalized diagrams that I have seen will present a challenge to external observers due to ad-hoc annotations, foreign elements, and the differences between versions. For instance, a hypothetical external observer will be challenged to understand the annotations on the methods of Fig. 3.20(b) or where the status codes fit. Understanding the complete solution of team E, such as its data model of Fig. 3.24, will require him to locate the earlier versions, such as Fig. 3.8, identify the differences, and elicit some details from the original. Interpreting all these representations would require familiarity with the context of their creation.

3.5.4 Order of evolution

Throughout all observed sessions, it was evident that designs and artifacts do not evolve on a single continuous path of monotonically-increasing completeness. In response to ad-hoc needs, the designs and artifacts evolve on multiple paths that are interwoven and sometimes merged. Every artifact or idea may be abandoned, only to be recalled at a later point and be discussed, referenced, copied, or continued on the same canvas or on a different one. Each artifact or entity may be at a different level of maturity and completeness.

The interwoven and noncontiguous order of work presents several challenges. During the session, it

challenges the team members' memory, individually and as a group, and they spend much effort attempting to recall and recap past discussions and decisions. Physically, it also increases clutter and confusion as the team struggles to maintain a limited working set of diagrams, requiring them to search, move, and flip diagrams. It also depreciates the value of spatial cues which are essential to locating material [25]. The impact also persists after the session has ended since the loss of temporal order can present challenges to interpreting the final diagrams.

For instance, because team A made changes to its posted list of assumptions while working on the diagram of Fig. 3.20(a), a reader examining earlier artifacts might mistakenly assume that all listed assumptions were made at the same time at the beginning of the session and thus interpret those artifacts in this mistaken context.

Note that identifying the connections and interweavings between artifacts is also challenging. Access to artifacts can vary in length and impact, from longer references that involve copying, making changes, or even a return of focus and a continuation of discussion drawing, to short references, acknowledged only in glance and speech. Individual designers leverage the extreme collocation to maintain awareness of what others are doing and what they are focused on. As could be expected, we have seen coordination problems arise when teams split into subgroups [25].

The nonlinear path of the design evolution also exacerbates the problem of understanding the results, as it is difficult to interpret the resulting artifacts without understanding how and why they evolved from the originals. While the design rationale is rooted in the context of both versions of the artifact, it may only be captured in the discussion itself, and the difference between the artifacts may be the only hint that such a discussion ever took place. In addition, subtle references to artifacts and temporal clues that may have important impact are not captured.

3.5.5 Need for preserving context

Perhaps the most important contribution of my findings is in highlighting the critical role that contextual cues, memory, and alternate communication mediums play in the design process. This information, important during the session, is particularly crucial for a subsequent understanding of the designs and their rationale. At present, casual observation suggests that most teams do little to preserve this information, and at best preserve only the final diagrams after the meeting has ended. However, it is not always possible to foresee in advance what decisions will be revisited in the future and therefore what artifacts will be important. In some cases this will only be recognized after several reproductions [16].

For these reasons, I believe that such investment may often be preferable to attempts to create a final and aesthetic version of the artifacts during the meeting. Therefore, the "take home message" for all designers is to try and preserve as much of this information as possible. The question, of course, is how to best capture this information.

In my effort to understand design diagrams, I make use of various information sources that were available. In terms of visual information, perhaps the most useful resource, and also the one easiest to obtain, is the collection of photos taken during the session. These captured intermediate states of the diagrams and their spatial location in relation to other artifacts. They helped me understand the evolution of each diagram and of the design as a whole, as I could compare diagrams over time. In some cases, they revealed important material that was later erased. With the ubiquity of digital cameras, most designers should be able to take these photos even in spontaneous meetings.

Unfortunately, unless photos are taken frequently, important details may be lost, including concurrent work on multiple diagrams, short references, or examples that are quickly erased. In addition, photos cannot capture gestures, "drawing in the air", and the glances that establish reference. This information

is particularly important if a voice recording or transcript of the meeting is captured, since many verbal references rely on a specific visual context, such as what the designer has just looked at.

In my study, only the video stream captured enough of this evidence, due to its continuous nature. At present, video may be the most comprehensive means of capturing contextual details from collocated collaboration and may therefore be the safest approach for teams seeking to minimize the risk of losing critical information. However, although its effectiveness for requirements engineering has recently been demonstrated [18], the use of video in software design remains controversial.

One obvious concern is the feasibility of recording all collaborative situations. However, since the goal of the video is primarily to capture gestures and timing information and offer context to verbal interaction, even low-fidelity streams are sufficient. With decreasing storage costs and the availability of webcams and video-capable digital cameras, I believe that this footage can be captured inexpensively even in spontaneous meetings.

A more fundamental concern regarding video is the challenge of locating information within such an unstructured stream. In my experience, the random access and rapid skimming capabilities afforded by digital video are extremely helpful in pinpointing relevant sections efficiently without watching the entire stream. Video can thus be considered as a form of “insurance” in case specific information may be sought in the future.

In the future, advances in video analysis, or perhaps correlated data from instrumented electronic whiteboards, may provide structure to this stream and offer new possibilities such as the creation of useful summaries or aggregations of context. For example, one could imagine a tool that could help supply useful context for a confusing notation by visually summarizing the set of actions that occurred immediately prior to creation of the notation, or by highlighting anything edited in close temporal proximity to the use of the notation. Based on my results, it seems that finding a rich set of techniques for capturing and displaying contextual information is a potentially very rich research area.

I realize that some teams will not be able to use technological means to capture contextual information. To these designers I can only suggest that they remain constantly aware of the future interpretability and traceability of their work, and try and preserve information in the diagrams whenever possible. This does not have to mean an explicit and costly documentation effort; it could be as simple as annotating some artifacts in the short pauses before switching to another area.

The last two chapters were concerned with software design. In both of these chapters, I argued that there is a need to preserve a lot of knowledge, but that is only one side of the equation. There is a need to enable designers to retrieve it in appropriate context. For instance, past decisions about an entity could be relevant the next time that this entity is referenced. We now make the abrupt switch to the world of programming and code, on which the rest of this dissertation focuses. In this world, knowledge preservation is quite common, in the form of member documentation. However, as we shall see, recall still presents a significant challenge. The next chapter presents directives, elements of knowledge captured by the original developers who can significantly affect users of the resulting artifacts.

Chapter 4

Directives

Our discussion so far dealt with software design in the physical world, where a variety of connections exist between artifacts, and various forms of knowledge can be associated with entities. The focus of the rest of this dissertation, however, is on the more restrictive domain of software source code, and specifically on the connection between documentation clauses and methods.

This chapter elaborates and illustrates the notion of *directives*, clauses in method documentation that convey important yet often unexpected instructions to callers. Section 4.1 introduces this concept and suggests criteria for distinguishing these directives from other clauses in software documentation. Section 4.2 discusses related research on documentation and comments, and relates the concept of directives to these works. The rest of this chapter is devoted to presenting a taxonomy of the types of directives that I have encountered when I systematically surveyed the documentation of the JAVA standard library. The goal of this presentation is to investigate the breadth of issues of which method callers need to be aware, and further motivate our investigation of directive awareness. To this end, the narrative presents many examples.

4.1 What are directives?

4.1.1 The principle of least-surprise

In his now-famous presentation, “How to design a good API and why it matters” [10], Joshua Bloch, who designed many of JAVA’s core APIs, listed various design maxims. One of these is:

Obey the principle of least astonishment - every method should do the least surprising thing it could, given its name. If a method doesn’t do what users think it will, bugs will result.

As we shall see in this chapter, even in the standard JAVA library there are many methods that do not follow this principle.

While the *principle of least astonishment* is a well-recognized concept, it is a very subjective one. Since every user may have different expectations given a name, what is considered a surprise? While the concept of directives described in this dissertation is broader in its scope, its roots are tied to this principle and to the notion of surprise. As a result, there will not be a simple predicate that can be used to determine whether a certain clause that appears in a method’s documentation is a directive. However, I will present here general criteria for identifying potential directives, and in Chapter 7 will present some empirical data on whether individuals can reliably and consistently distinguish directives from other forms of documentation.

Prevalence of specifications

Almost every clause that appears in a *JavaDoc* is likely to be useful in at least some context to some particular audience. After all, if we assume that a clause is not erroneous, then it should at least be useful to a developer who: 1) knows nothing about the method and the API, 2) needs to know everything about the method for some systematic evaluation or review, 3) may not be particularly skilled or experienced.

However, most developers do not match this description. First, few people use an API without knowing anything about it or its domain. Second, a method's documentation is read because its name already seemed relevant to what the developer was seeking. Thus, the developer may have some knowledge of the domain and certain expectations. Third, most developers have some skill and prior experience in programming, and can use that experience when dealing with unfamiliar APIs. Finally, and perhaps most important, in most development and maintenance tasks, there is no need to know everything about the method, just what is necessary to use it correctly in the current context.

Here, however, lies the problem: Sun's guidelines for writing API documentation require that the *JavaDoc* for each method include a complete and detailed specification [1, 82]. Such specifications are by nature meant for an audience that is performing a systematic review, rather than a pragmatic developer aiming to learn the necessary minimum. The complete specifications thus obfuscate the important details. In my survey of APIs, I observed that these principles are followed religiously in major and highly visible APIs, and the effort is relaxed in more limited APIs and in less central areas of the major APIs. Paradoxically, these limited specifications be better-suited for the needs of most developers.

Nevertheless, since specifications are so detailed, I argue that the vast majority of clauses in the documentation of major APIs are "specification clauses" and thus less relevant for most developers in most scenarios.

4.1.2 Surprising clauses

At the other extreme from specifications lie clauses that are completely unexpected even to skilled developers who are relatively well versed in the API.

The most obvious form of such clauses are shortcomings of the method that are surprising because potential clients may be optimistic in their expectations of the completeness and correctness of the API. In small domain-specific or proprietary API, a method might only handle a subset of all legal inputs, often the ones that represent the most frequent or important scenarios. Alternatively, it may only be partially implemented, and not produce the expected output.

In the worst case, such behaviors are not documented and are found at runtime. More commonly, these limitations are documented, but not necessarily in the header. For example, it is not rare to see to-do comments left in the implementation of methods in proprietary APIs. In other cases, limitations and pitfalls may appear as comments within in the source code, close to where the problem occurs. The problem with such clauses is that they affect callers but are not visible to them unless they acquire and investigate the source code. Limitations appear less common in major APIs, perhaps because of the added scrutiny and history of revisions. In addition, APIs developed with test-driven methodology are typically fully-implemented in order to satisfy all the tests.

Even in major APIs, however, certain documented behaviors may surprise even seasoned developers who are familiar with the API. For example, the central `String` class in JAVA offers a `replace` method that takes two character sequences and replaces in the string on which it is invoked every occurrence of the first with the text of the second. The same class also offers a similarly-named `replaceAll` method with a similar signature. Based on the names and the similarity to common UI idioms, a developer might expect that `replace` does a single replacement while `replaceAll` does multiple replacements.

A developer who is not familiar with either and seeks to do replacements may choose to use the latter as it seems like the closest match to his goals.

Scrutiny of the documentation of this method in all JAVA versions up to and including 1.5, depicted in Fig. 4.1 shows that the method actually works with regular expressions, a worrying sign for very experienced developers. The documentation in version 1.6, which is depicted in Fig. 4.2, adds an important detail: “backslashes and dollar signs in the replacement string may cause the results to be different than if it were being treated as a literal replacement string”. As a result, the call `myString.replaceAll("USD", "$")` may produce unexpected results.

replaceAll

```
public String replaceAll(String regex,
                        String replacement)
```

Replaces each substring of this string that matches the given [regular expression](#) with the given replacement.

An invocation of this method of the form `str.replaceAll(regex, repl)` yields exactly the same result as the expression

```
Pattern.compile\(regex\).matcher\(str\).replaceAll\(repl\)
```

Parameters:

`regex` - the regular expression to which this string is to be matched

Returns:

The resulting `String`

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

Figure 4.1: Documentation of method `String.replaceAll()` in JAVA 5

replaceAll

```
public String replaceAll(String regex,
                        String replacement)
```

Replaces each substring of this string that matches the given [regular expression](#) with the given replacement.

An invocation of this method of the form `str.replaceAll(regex, repl)` yields exactly the same result as the expression

```
Pattern.compile\(regex\).matcher\(str\).replaceAll\(repl\)
```

Note that backslashes (\) and dollar signs (\$) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string; see [Matcher.replaceAll](#). Use [Matcher.quoteReplacement\(java.lang.String\)](#) to suppress the special meaning of these characters, if desired.

Parameters:

`regex` - the regular expression to which this string is to be matched

`replacement` - the string to be substituted for each match

Returns:

The resulting `String`

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

Figure 4.2: Documentation of method `String.replaceAll()` in JAVA 6

Similarly, a central interface in *Eclipse* is `ITextViewer`, which declares functionality common to most textual editors and viewers. One of its methods, `getVisibleRegion`, is expected to return the

range of text lines that are currently visible in the viewer. Scrutiny of its documentation (Fig. 4.3), however, reveals that viewers implementing a certain extension of this interface may actually change the fraction of the input document that is shown. A side effect from what is expected to be a straightforward getter method is quite surprising. In fact, I found this problem after receiving a bug report that using the *eMoose* tool broke code folding in the JAVA editor. Indeed, the implementation of this method in `ProjectionViewer`, a base of the JAVA editor, has a nasty side effect. As can be seen in Fig. 4.4, the code simply disables folding to obtain a better idea of the visible range, never enabling it again.

IRegion org.eclipse.jface.text.ITextViewer.getVisibleRegion()

Returns the current visible region of this viewer's document. The result may differ from the argument passed to `setVisibleRegion` if the document has been modified since then. The visible region is supposed to be a consecutive region in viewer's input document and every character inside that region is supposed to be visible in the viewer's widget.

Viewers implementing `ITextViewerExtension5` may be forced to change the fractions of the input document that are shown, in order to fulfill this contract.

Returns:
this viewer's current visible region

Figure 4.3: Documentation of method `ITextViewer.getVisibleRegion()`

```
/**
 * @see org.eclipse.jface.text.ITextViewer#getVisibleRegion()
 */
public IRegion getVisibleRegion() {
    disableProjection();
    IRegion visibleRegion= getModelCoverage();
    if (visibleRegion == null)
        visibleRegion= new Region(0, 0);

    return visibleRegion;
}
```

Figure 4.4: Source code of Eclipse method `ProjectionViewer.getVisibleRegion()`

Based on my systematic inspection of several large libraries, clauses in documentation fall somewhere on a continuum between mundane specifications and extremely surprising details. As I've previously argued, the vast majority of clauses, to which we shall refer as "specifications", lie very close to the first extreme. Cases in the other extreme, as illustrated above, are thankfully not very common in major APIs. Such APIs are designed with sufficient attention and evolve continuously, so that major pitfalls like the one in `replaceAll` are eventually documented. Of course, documenting these pitfalls might not be sufficient to help developers avoid them. Because developers do not expect the method to surprise them, they may not read its documentation in the first place, a notion that is supported by the lab study presented in Chapter. 6.

There is, however, a significant set of clauses which may surprise or confirm the suspicions of reasonably skilled developers who are not very familiar with the method or the particular API. I argue that this audience is the most likely to suffer the consequences of knowledge awareness problems and thus most likely to benefit from *eMoose*. Since skilled developers often use new APIs as means to an end, they may do the bare minimum to achieve their goals. They may not invest in learning the fundamental concepts of the API, and will likely rely on code samples which they will try to modify to fit their needs. Reliance on these samples is risky as they were likely written by someone well-familiar with the API. When taken out of context and changed by someone not as familiar, certain clauses may be violated.

Such clauses are a minority - they do not occur in every method, and they may be surrounded by many specification clauses. However, they may be frequent enough to significantly affect seasoned developers in everyday work. These are our directives.

4.1.3 Properties of directives

As explained above, there is no clear predicate for segregating the continuum of documentation clauses into specifications and directives. Instead, I define subjective properties that directives should have. The aggregation of the degrees to which a particular clause has each of these properties may help place it on this continuum. In practice, our *eMoose* tool allows users to rate a directive on a scale between 0 and 5. API authors and users attempting to identify directives in API documentation for the benefit of other users can use this rating mechanism to mimic the placement of clauses in the continuum. The presentation of directive awareness cues takes into account these ratings so that clauses below a specified threshold are not presented.

4.1.4 Property 1 - Directives require or imply client action

Perhaps the most important property of a directive clause is that it demands an action from the caller or suggests such an action. The demand can be explicit or implicit. Applying this property allows us to “eliminate” many clauses which are descriptive or explanatory in nature and are therefore less relevant to clients.

We will use the term *imperative directives* for clauses that demand an action from the caller. For instance, they may instruct the client to avoid making the call in certain situations, or to make another call prior to this one. Imperative directives may be easier to identify. We will use the term *informative directives* for clauses that merely suggest a course of action. For example, warning a potential caller that a particular call may have a performance penalty or an unexpected side effect may lead the caller to avoid the call or take preventive measures.

Let us illustrate the application of this property on several examples. Consider the documentation of `String.replaceAll()` from JAVA 6 (Fig. 4.2). The first two paragraphs merely explain what the method does, and what it is equivalent to. However, the paragraph starting with “note that” indicates to the caller that unexpected results would occur for certain inputs. The caller may want to examine the prospective input and avoid this call, perhaps using the suggested call to `Matcher.quoteReplacement`. Since this clause explicitly requires callers to avoid particular inputs that are not obvious from the method’s signature, we consider it to be a directive.

Imperative directives that demand an action typically specify it explicitly. Those that demand the avoidance of an action that will result in an error, however, often merely specify the condition and the error. That the caller must take action to avoid this result is implied. To illustrate, consider the documentation of `Connection.setClientId` from JMS, which we reproduce in Fig. 4.5. Let us now examine every clause and determine whether it could potentially be a directive.

The first paragraph merely describes the purpose of the method and is thus not a directive. The second paragraph (P1) mentions a “preferred way” to do something. Describing an alternative to the call may suggest an action (a different call) for the client, so this may be a directive. The third paragraph contains multiple clauses. P3 mentions how a client can set the identifier, and is therefore a potential directive. P3 is also a potential directive as it describes when not to use this call. P4 states that an exception will be thrown if an administratively-configured identifier has been set, and can be interpreted as an instruction to avoid doing so. The client is instructed in P5 to avoid making prior calls on this object, and this is clearly a directive. However, the description in P6 merely describes the error that will occur. The fourth

● void javax.jms.Connection.setClientID(String clientID) throws JMSEException

Sets the client identifier for this connection.

The preferred way to assign a JMS client's client identifier is for it to be configured in a client-specific ConnectionFactory object and transparently assigned to the Connection object it creates. P1

Alternatively, a client can set a connection's client identifier using a provider-specific value. P2

The facility to set a connection's client identifier explicitly is not a mechanism for overriding the identifier that has been administratively configured. It is provided for the case where no administratively specified identifier exists. If one does exist, an attempt to change it by setting it must throw an IllegalStateException. P3

If a client sets the client identifier explicitly, it must do so immediately after it creates the connection and before any other action on the connection is taken. P4

After this point, setting the client identifier is a programming error that should throw an IllegalStateException. P5

The purpose of the client identifier is to associate a connection and its objects with a state maintained on behalf of the client by a provider. P6

The only such state identified by the JMS API is that required to support durable subscriptions. P7

If another connection with the same clientID is already running when this method is called, the JMS provider should detect the duplicate ID and throw an InvalidClientIDException. P8

Parameters: P9

clientID the unique client identifier P10

Throws:

JMSEException - if the JMS provider fails to set the client ID for this connection due to some internal error. P11

InvalidClientIDException - if the JMS client specifies an invalid or duplicate client ID. P12

IllegalStateException - if the JMS client attempts to set a connection's client ID at the wrong time or when it has been administratively configured. P13

Figure 4.5: Javadocs of setClientId with enumerated clauses

paragraph (P7 and P8) merely describes the purpose of the identifier, and does not convey instructions to clients. The next paragraph (P10), however, describes an error condition that implies a scenario that the client must avoid. Finally, the later clauses merely describe and elaborate what we already know.

Informational directives are particularly likely to only implicitly suggest a caller reaction. For example, consider the documentation of `ITextView.getVisibleRegion()` from Fig. 4.3. The first paragraph merely describes the behavior of the method. The second paragraph, however, describes a potential side effect. This description does not explicitly require user action. However, side effects implicitly suggest that a caller wishing to avoid them should avoid invoking this method.

The above examples illustrate that careful attention and a subjective decision is required to determine if a clause implies an action.

4.1.5 Property 2 - Directives should not be trivial, expected, or common

A second property of directives is that they should convey meaningful information that the potential caller would not expect to find and would not explicitly seek.

While the conveyed information may not necessarily be a complete surprise to a skilled developer who is familiar with the API, it should definitely not be obvious from the method signature, the containing class, the API, or the domain. The examples of `replaceAll` and `getVisibleRegion` showed behaviors that were far from obvious. In `setClientID`, however, some of the clauses may be obvious to someone familiar with the API or its concepts. For instance, to someone who is familiar with the concept of administrative configuration, the ability to use it to set the identifier may be obvious.

A related issue is that the information should not be a “pattern” that repeats itself in most methods. Such prevalence reduces the “surprise value”, but also allows interested callers to easily search for this

information. For instance, most methods present a list of the exceptions that they may throw, even though some of the error conditions are obvious. Interested callers who wish to program defensively can easily seek and identify this list. Similarly, many methods specify whether incoming parameters can be null, so interested callers can easily seek this information. On the other hand, the fact that a replacement string cannot contain specific symbols is surprising and thus worthy of a directive.

4.1.6 Property 3 - A directive should be relevant to most callers and scenarios

Methods can be called for different purposes and from different contexts, some more common than others. Since we will “push” directives into the awareness of callers, the clauses that we choose to push should apply to the majority of potential callers. If a clause is only relevant in very rare or unique cases, most readers will ignore it and it will be a distraction rather than a surprise. For example, if administrative configuration is extremely rare in JMS, then the directives having to do with it are a distraction (as shall be demonstrated in a subsequent chapter). Nevertheless, the decision should balance the likelihood of occurrence against the magnitude of the consequences. If the consequences of violating a clause are extremely dire, then there may be a benefit to including it in the set of directives even if it applies only in rare situations.

4.1.7 Property 4 - Significant consequences to lack of awareness

Since there is a cognitive cost to bringing a directive into the awareness of callers, the last property requires that the consequences of a lack of awareness be severe. A violation that results in a fatal runtime error or data corruption is particularly significant, whereas exceptions, side effects, and other concerns may be less so. Note, however, that even informative directives that are ignored can lead to serious problems, as was the case with the code folding bug in *eMoose* that resulted from the side effect of `setVisibleRegion`.

4.2 Related work

Documentation artifacts play important roles in the development, maintenance, and use of software. Many projects start with requirements documents that are later used to create design and architecture documents. Many projects conclude with the creation of end-user documentation that avoids exposing implementation details. The focus of this section, however, is on documentation artifacts created during the implementation stage within source code files, and in particular function-level header documentation. We survey related research and relate each work to the ideas presented in this dissertation.

4.2.1 Documentation and comments

It is widely believed that comments should help maintainers understand source code. However, the prevalence, mechanisms, and motivations behind comment creation and use are not always transparent, leading to significant attention from researchers.

Importance of comments

Woodfield, Dunsmore and Shen [87] conducted a 1981 study to learn whether comments were effective in helping developers understand code. They provided 48 professional *Fortran* developers with 8 versions

of a program, created by varying modularization types and the presence of module-level comments. After studying the program, subjects had to answer 20 questions about it. The results showed that comments present in certain versions of the program had significant impact on the number of questions answered correctly. These findings confirm the intuitive expectation that source code, by itself, may not be sufficient for efficiently understanding existing programs.

Use of comments

Soloway et al. [75] sought to learn how maintainers understand code and use documentation to modify procedural `Fortran` programs, and then apply this knowledge to designing more useful documentation representations. To this end they conducted a study using a real system at JPL. Subjects were given a code booklet, and a separate documentation booklet that presented an overview, a call hierarchy chart, and a description of each module.

They argued that as maintainers read a line of code they develop expectations about subsequent lines, and if these expectations are difficult to form or are eventually not met they begin to inquire and explore other resources, including documentation. They also argued that calls to other functions interrupt this process and often force maintainers to go and explore the delocalized materials. Some subjects seemed to handle this delocalization problem by a systematic exploration of code and documentation together, but the authors argue that this is impractical for larger programs. Other subjects, however, followed an as-needed strategy, where they made ad-hoc decision whether to explore documentation or other locations. While this was more efficient, it was also riskier and lead to less optimal results.

Based on their findings, they created a documentation design with a single booklet that combines code and text. Each function receives two opposing pages, the first showing the code and the other the documentation. An additional section in the text provides information relevant to specific calls from the program text to other functions.

The problem of delocalization is well recognized and is an underlying motivation for our work. The same problem is inherent in modern object-oriented languages, where function or class header documentation correspond to the module and function descriptions from the above study. The finding that systematic exploration is effective but inefficient while the more efficient as-needed reading technique is error prone motivates this dissertation's aim to provide cues to help make these decisions. The proposed approach of presenting code and documentation side by side but still independently is, in my view, more practical than Knuth's *literate programming* approach [49] which advocates a deeper interweaving of the two. The proposed approach also resembles modern source code, where much documentation is conveyed as visually distinct source code comments.

The idea of conveying additional call-specific information is intriguing since it presents an opportunity to provide a caller with answers to questions that are specific to this context. However, it is only practical in a small and closed system, where a human can prepare such comments in advance. Nevertheless, conveying delocalized information in proximity to the call is a form of knowledge pushing. In this dissertation, all calls to a function are treated the same independently of context. However, providing information that is specific to a context, based on past activity or automatic analysis, is an important extension.

Keeping comments up-to-date with evolving code

Fluri, Wursch and Gall [33] investigated the extent to which new code is commented and whether comments are kept up-to-date as the code changes. They used an *Eclipse* plugin called *ChangeDistiller* to mine code evolution data for three open-source projects directly from the repository. They used this data

to map comments to program entities in each version, as well as to track changes across versions.

Their first major finding was that new code was rarely commented, so that the increase in the number of comments was marginal even as the actual codebase grew significantly. Second, in two of the three tools, method header comments were more prevalent by orders of magnitude than other types of comments. They were followed, after a great gap, by class header comments (perhaps because there are few classes), and then by documentation on actual method calls. In a third project, documentation on method calls was common, followed by method header comments.

These findings highlight the importance of header documentation, and suggest that it may be present in many projects, ready to be exploited. While the third project is, in my view, a likely outlier, the fact that its developers chose to document calls suggests that they see a need to provide information about the remote targets of the call. The paper did not provide many examples, but it would have been interesting to see if some of these calls described directives.

The study also found significant differences between projects in the level to which the comments were updated following code changes, although when this happened it typically occurred within the same commit. Interestingly, less than a half of *JavaDoc* changes occurred after changes to the method declarations or the method bodies. When such changes did occur, it was typically due to a change in the parameters or return types.

An upcoming paper by the same authors [34] reports on a follow up study in which eight different open-source and closed-source projects are analyzed. Interestingly, in projects that included APIs, when there was a significant change to API methods the documentation was not immediately fixed, but rather rewritten later. This may create risky inconsistencies.

Impact of comment presence

Marin [62] devoted his M.Sc. research to investigating the factors that lead developers to add comments. He studied the repositories of 9 open-source projects and found that comment rates varied widely between projects even for the same developer. One factor was that larger modifications were more likely to be commented. A second and more critical finding was that developers modifying code that was already heavily commented were more likely to comment their own changes. A subsequent lab experiment appeared to confirm this finding.

The implication of this finding is very significant, as it suggests that social or psychological factors may affect the willingness of developers to invest in writing documentation. An important question is whether a similar phenomenon would be seen if developers were asked to explicitly tag directives in the codebase or API. That is, if most methods explicitly list directives, would a developer adding a new method be more likely to do so?

Types of comments

A thorough study of comment types was carried out by operating-system researchers Padioleau, Tan et al. [69].

They initially collected a random sample of about 1000 comments from three operating systems written in *C*. They estimated that about half the comments were not merely explanatory but rather offered some important information that can potentially be “exploited”. They further broke down this group of comments and argued that: 20% clarify the meaning of numeric constants. 16% emphasize relationships such as data or control flow. 10% could be expressed by existing annotation languages, and could thus be automatically checked. 5% percent deal with locks. They further broke this categories down into finer

subcategories.

Next, they took samples of about 350 comments from several non-OS programs, such as *MySql*, *Firefox* and *Eclipse*. They found that about half of the comments in these projects were exploitable, but that the division was quite different. First, a much smaller portion of comments explained constants, perhaps thanks to the use of language constructs for these, but also due to the lower need for bit-level operations in non-OS code. The other C based programs had more comments on memory management while the Eclipse comments were particularly about error management.

While the exact numbers are of little relevance to this dissertation, the fact that the prevalence of comment types differs significantly between languages and domains is critical. We can expect certain programs or APIs to contain more documentation and directives, while others will contain very few. It is possible that the types of directives we will encounter will also vary significantly. For example, in this study operating systems dealt a lot with locking, while other programs dealt more with memory management.

4.2.2 Studies of task comments

While the focus of this dissertation is on documentation that provide guidelines or warnings about using a method, the presence of task related comments can be indicative of limitations or significant weaknesses in the method. Task comments are embedded comments that communicate the need for some action or modification rather than explanations of the associated code.

Ying et al. [90] studied task comments in an eclipse-based internal IBM codebase. They found that while the comments did not describe the source code, they conveyed many important details that may help understand the evolution of the program. For instance, some reminders weren't requests to perform a change, but rather to verify or review a change that took place, or to ask for clarifications about the code. More generally, the authors argued that task comments serve an important role as an asynchronous communications medium, under the premise that the intended reader would indeed read them. They also argued that such comments are often more informal and context sensitive than typical documentation, making them more difficult to read and parse automatically.

Storey et al. [79] carried out a later study focused on task comments. They aimed to identify and understand the annotations created to support programming tasks and workflow practices, and the processes for managing them and keeping them up to date. As a first step, they conducted a survey of 81 developers via an Eclipse community site. The results showed that even heavy *Eclipse* users do not make use of its bookmarking capabilities, and instead embed comments with keywords and metadata on which there is only partial agreement. Next, they extracted task annotations from 10 open source projects, and identified a significant number of task comments, primarily to-dos. Only a very small portion of these comments were automatically generated. As a third phase, they conducted extensive interviews with four developers from three *Eclipse* projects. The projects varied from extensive use of bug-tracking systems with minimal to-do comments to an extensive use of such comments for low-priority issues, marking incomplete solutions, and requesting reviews. As a fourth phase they extracted comments from several projects at regular intervals, and studied the lifetime of the comments. It turned out that most task annotations were short-lived, but a certain portion stayed on for a long time, some eventually turned into official modification requests.

The researchers concluded that to-dos serve many intricate task-management purposes such as denoting shorter tasks, low-priority subtasks, problems, areas for reviews, etc. Their main benefit is the low cost of production, the availability of context, and their informal and temporary nature. Like bug reports, however, to-dos are too visible and permanently accessible, reducing their use in open source projects. The researchers also raise an interesting issue, that some to-do comments lack sufficient context

for future interpretation. This is very similar to our finding for design artifacts.

Task comments are not directives in the sense described in this dissertation because they are rarely part of the public interface of the method. However, the presence of a reminder often means that the method does not perform what one might expect from the signature, or that there may be some manifestation that can affect callers. In addition, one of the interviewees in the study stated that: “*We put all these comments in our code and then we don’t look back at them. None of them actually remind us...*”, and this is clearly an indication of an awareness problem. Since task comments can affect maintainers and developers working within the same codebase, the *eMoose* tool is designed to address them. When such comments are present in a method, the tool will be able to provide corresponding decorations on invoking code, in the hope of alerting callers. In addition, Ying et al’s findings that certain comments are aimed at specific developers suggests that techniques to increase awareness of commented knowledge may need to be fine tuned to the individual.

4.3 Introduction to the taxonomy of directives

We now turn to presenting a taxonomy of directives. This taxonomy was developed based on a detailed survey of the JAVA standard library. In this survey, I examined every `public` method in most `public` interfaces and classes. I read the documentation of each and identified potential directives, which I proceeded to tag. The categories that are presented here are the major ones but are not exhaustive or mutually exclusive. Some directives may fall in multiple categories, or even outside any of them. In addition, only a few representative examples are presented for each type of directive. The goal is to make the reader understand the breadth and importance of directives rather than to identify each and every one. Subsequently, I repeated the detailed study with significant portions of the *Eclipse* API and confirmed the presence of all categories. However, the examples here are limited primarily to the JDK.

4.4 Imperative directives - Restrictions

Imperative directives in a method’s documentation define rules for the use of this method. If these rules are violated, an immediate error may occur during the execution of the method, or the state of the system would be corrupted, leading to a subsequent failure that may be much harder to debug. The set of imperative directives may include many clauses that are already recognized as part of the protocol for using the object, such as explicit preconditions and locking requirements. Therefore, some imperative directives can be stated using special formalisms which would allow automated conformance checking. As we shall see, however, many imperative directives may not be straightforward to translate to a formal specification and instead rely on subjective interpretation

The strongest and most straightforward type of an imperative directive is, perhaps, the invocation restriction.

Object oriented languages typically have specific constructs for restricting access to a method, and these restrictions are enforced at compile time. For example, while a method declared as `public` can be invoked by anyone, methods declared as `private` can only be invoked from within the same class, and methods declared as `protected` can be invoked from within the same class or its subtypes. In *C++*, the *friendship* mechanism allows a class to explicitly specify other classes and specific methods within them that are allowed to access its non-public methods. While JAVA does not support friendship, it introduces a notion of packages, allowing calls to be restricted to items within the same package.

Our survey of APIs suggests that the existing syntactic mechanisms, including the package system,

may not be sufficiently expressive to meet all the restrictions that developers have in mind. This forces developers to declare certain methods as `public`, and use documentation to convey restrictions on use. These restrictions are essentially directives, and as such carry the risk of never reaching the caller's awareness, with potentially severe consequences.

We have encountered different types of restrictions and reasons behind them. We shall now survey them in depth because they illustrate how limitations of the programming language force developers to rely on natural text.

4.4.1 Deprecated methods

As APIs and libraries evolve, certain classes and methods are no longer used. To ensure backwards-compatibility they are kept in the interface, but new callers are encouraged to avoid them.

In JAVA, the `@deprecated` tag can be placed anywhere in the *JavaDoc* of the method to declare that it should not be used. The compiler can produce warnings when deprecated methods are invoked, alerting callers to find a different solution. In addition, the *Eclipse* IDE helps callers identify calls to deprecated methods by marking these calls with a strikethrough line. This is similar to the approach taken by *eMoose* in highlighting calls.

Since the `@deprecated` tag is so well supported, we were surprised to see instances in the JDK where text was used instead of this tag. For instance, the documentation of the `Event` class in AWT GUI toolkit states that this class is obsolete and kept for backwards-compatibility. Its constructors state the same, without using this tag. Similarly, the `RepaintManager.currentManager()` in SWING, seen below, indicates that it should not be used as it is meant for backwards compatibility. Getting and setting the default locale in the `SWING UIDefaults` class is also discouraged without an explicit use of this tag.

currentManager

```
public static RepaintManager currentManager(JComponent c)
```

Return the `RepaintManager` for the calling thread given a `JComponent`.

Note: This method exists for backward binary compatibility with earlier versions of the Swing library. It simply returns the result returned by `currentManager(Component)`.

getDefaultLocale

```
public Locale getDefaultLocale()
```

Returns the default locale. The default locale is used in retrieving localized values via `get` methods that do not take a locale argument. As of release 1.4, Swing UI objects should retrieve localized values using the locale of their component rather than the default locale. The default locale exists to provide compatibility with pre 1.4 behaviour.

4.4.2 Methods made public due to language issues

Quite a few restrictions result from the rules of the JAVA language and the limitations of its restriction mechanisms.

Interaction between packages

The rationale behind the restricted-access modifiers in JAVA: `protected`, `private`, and `package` is to allow the implementation of certain methods to be broken down into smaller operations without making these implementation-specific methods part of the public API. Whereas `private` and `protected` were “inherited” from C++ and allow these methods to be used from within the class, the rationale behind adding the `package` modifier was to enable interaction between related classes and the creation of

package-specific utility classes. Unfortunately, these modifiers do not address all restriction scenarios.

The root of this problem is that the `package` modifier essentially treats all classes in the same package as a single module, but closely coupled classes are often spread across multiple packages. For instance, many components are organized as a hierarchy of subpackages with subcomponents placed in corresponding subpackages. Additional conventions, such as the creation of `internal` subpackages in *Eclipse*, further spread classes across multiple packages. The problem with package hierarchies is that they get no special treatment by the language. A package and its subpackage are treated as if they were completely independent and therefore have no access to each other's package-protected elements. In other words, a subcomponent cannot access the elements of its container and vice versa. In some additional cases, classes in various components may need to interact across the boundaries of a package. In all these cases, the developers must make the method `public`, leaving them no choice but to rely on documentation text to restrict access.

For example, in the AWT UI toolkit, the central `Container` class has an `addNotify` method. Its documentation, shown below, indicates that it is only provided for use by the toolkit implementation. Indeed, a search of the JDK source code reveals that this method is invoked by classes in the `awt.event` subpackage and by classes in `Swing`.

addNotify

```
public void addNotify()
```

Makes this `Container` displayable by connecting it to a native screen resource. Making a container displayable will cause all of its children to be made displayable. This method is called internally by the toolkit and should not be called directly by programs.

In the example below from `SWING`, the `RowSorter.allRowsChanged()` method is `public` only to allow specific `View` classes to access it. The views can be in different packages, such as ones created by users.

allRowsChanged

```
public abstract void allRowsChanged()
```

Invoked when the contents of the underlying model have completely changed. The structure of the table is the same, only the contents have changed. This is typically sent when it is too expensive to characterize the change in terms of the other methods.

You normally do not call this method. This method is public to allow view classes to call it.

Methods made public due to interface implementation

We've seen a few cases where restricted methods are made public due to the implementation of an interface. In `C++`, a class can inherit privately from another class, meaning that other classes are not aware of the inheritance and do not expect it to support the inherited operations. In `JAVA`, subtyping is always public, so that when a class implements a particular interface, it is publicly announcing itself as a subtype of the interface and must publicly support all its methods. If the interface was implemented for specific implementation purposes, such as supporting events, then its methods are exposed.

For example, the `ScrollPaneAdjustable` class in AWT implements the `Adjustable` interface, but explicitly discourages users from calling this method.

setMaximum

```
public void setMaximum(int max)
```

This method should **NOT** be called by user code. This method is public for this class to properly implement `Adjustable` interface.

Similarly, the `getCurrentFocusCycleRoot` method in the AWT's `KeyboardFocusManager` is intended to use in implementations of this interface.

getCurrentFocusCycleRoot

```
public Container getCurrentFocusCycleRoot()
```

Returns the current focus cycle root, if the current focus cycle root is in the same context as the calling thread. If the focus owner is itself a focus cycle root, then it may be ambiguous as to which Components represent the next and previous Components to focus during normal focus traversal. In that case, the current focus cycle root is used to differentiate among the possibilities.

This method is intended to be used only by `KeyboardFocusManagers` and focus implementations. It is not for general client use.

Another example, from SWING, involves the popular `JComboBox` widget class. This class implements several listener types and therefore has some public callback methods that are not meant for general use. In my view this represents a design error for the class since listeners can be “privately” implemented in internal classes.

actionPerformed

```
public void actionPerformed(ActionEvent e)
```

This method is public as an implementation side effect. do not call or override.

Specified by:

[actionPerformed](#) in interface [ActionListener](#)

Constructors and other methods for use by specific builders

Common ways to create and initialize objects include the use of constructors with multiple parameters, a default constructor with subsequent calls to setters, or static factory methods [80]. When multiple means of initializations are allowed, confusion can ensue.

We noticed many cases where classes declared constructors and similar creation methods for the strict purpose of supporting initialization by specific tools. For example, objects that are deserialized, created from XML via `apache-digester`, or loaded from a database via `Hibernate` must all have public constructors, as these mechanisms and tools first create an empty object and then initialize its fields.

Below, we see how the `DataFlavor` class in the AWT library, states that its default constructor is created for supporting the `Externalizable` interface.

DataFlavor

```
public DataFlavor()
```

Constructs a new `DataFlavor`. This constructor is provided only for the purpose of supporting the `Externalizable` interface. It is **not intended for** public (client) use.

An opposite example comes from `GridBagConstraints` in AWT, whose multi-parameter constructor is meant for use only by automatic source generation tools.

GridBagConstraints

```
public GridBagConstraints(int gridx,  
                          int gridy,  
                          int gridwidth,  
                          int gridheight,  
                          double weightx,  
                          double weighty,  
                          int anchor,  
                          int fill,  
                          Insets insets,  
                          int ipadx,  
                          int ipady)
```

Creates a `GridBagConstraints` object with all of its fields set to the passed-in arguments. Note: Because the use of this constructor hinders readability of source code, this constructor should only be used by automatic source code generation tools.

Protected methods that should be private

So far we only looked at `public` methods, but there are cases where the documentation of a `protected` method restricts its use by subtypes. Unlike `C++`, `JAVA` does not allow a subtype to restrict the access level of an inherited method. Now imagine that there is an abstract class in a library, which defines a certain `protected` method that is meant to be overridden by a concrete implementation within the library. However, that library subclass can still be further subclassed by external clients. There is no way to prevent further overriding by these clients, leaving no choice but to document. This is prevalent in GUI toolkits, as illustrated below for the `GridBagLayout.getSize` in `AWT`.

`getMinSize`

```
protected Dimension getMinSize(Container parent,
                               GridBagConstraints info)
```

Figures out the minimum size of the master based on the information from `getLayoutInfo`. This method should only be used internally by `GridBagLayout`.

Parameters:

parent - the layout container
info - the layout info for this parent

Returns:

a `Dimension` object containing the minimum size

Since:

1.4

`GetMinSize`

```
protected Dimension GetMinSize(Container parent,
                               GridBagConstraints info)
```

This method is obsolete and supplied for backwards compatibility only; new code should call `getMinSize` instead. This method is the same as `getMinSize`; refer to `getMinSize` for details on parameters and return value.

4.4.3 Specific callers

Many classes have methods that are meant for use by specific callers, which may be in the same package or elsewhere. Since there are no syntactic mechanisms to define these restrictions, API authors state the permitted or forbidden callers in the API documentation; these restrictions are not enforced. We note that a friendship mechanism of the type supported in `C++` would only address a small portion of these calls in which a specific class or method is allowed to make the access. In many cases, the legal callers are an amorphous group that is defined using natural language or that may include classes that are not yet defined. We divide the specific callers into *internal* and *external* callers.

Internal callers

We have encountered many methods that are meant for the use of the implementation or “the toolkit”, although it is not always clear where the boundaries of these concepts are. We saw an example of such restriction earlier with the `addNotify` method from `Container`. However, additional examples are presented below.

minimumLayoutSize

```
public Dimension minimumLayoutSize(Container parent)
```

Determines the minimum size of the parent container using this grid bag layout.

Most applications do not call this method directly.

firePopupMenuWillBecomeVisible

```
public void firePopupMenuWillBecomeVisible()
```

Notifies `PopupMenuListeners` that the popup portion of the combo box will become visible.

This method is public but should not be called by anything other than the UI delegate.

setSelectedFrame

```
public void setSelectedFrame(JInternalFrame f)
```

Sets the currently active `JInternalFrame` in this `JDesktopPane`. This method is used to bridge the package gap between `JDesktopPane` and the platform implementation code and should not be called directly. To visually select the frame the client must call `JInternalFrame.setSelected(true)` to activate the frame.

Other callers

In more interesting cases, the permitted caller is not necessarily part of the implementation, but is still explicitly designated.

For example, the method `getSource` in the AWT's `Image` class is meant for use by specific types of methods and classes.

getSource

```
public abstract ImageProducer getSource()
```

Gets the object that produces the pixels for the image. This method is called by the image filtering classes and by methods that perform image conversion and scaling.

The `defaultReadObject` method in the standard `ObjectInputStream` is public, but meant only for use by a specific method in classes that are being deserialized.

defaultReadObject

```
public void defaultReadObject()
    throws IOException,
           ClassNotFoundException
```

Read the non-static and non-transient fields of the current class from this stream. This may only be called from the `readObject` method of the class being deserialized. It will throw the `NotActiveException` if it is called otherwise.

The `setSelectedFrame` in the `JDesktopPane` class in SWING is meant for use in bridging the pane and the platform implementation.

setSelectedFrame

```
public void setSelectedFrame(JInternalFrame f)
```

Sets the currently active `JInternalFrame` in this `JDesktopPane`. This method is used to bridge the package gap between `JDesktopPane` and the platform implementation code and should not be called directly. To visually select the frame the client must call `JInternalFrame.setSelected(true)` to activate the frame.

Specific purpose

A particularly interesting form of restriction states the purpose of the use rather than the permitted caller. For example, the documentation of `findDeadlockedThread` in `ThreadMXBean` states that this seemingly useful method should only be used for troubleshooting and debugging rather than synchronization.

`findDeadlockedThreads`

```
long[] findDeadlockedThreads()
```

Finds cycles of threads that are in deadlock waiting to acquire object monitors or [ownable synchronizers](#). Threads are *deadlocked* in a cycle waiting for a lock of these two types if each thread owns one lock while trying to acquire another lock already held by another thread in the cycle.

This method is designed for troubleshooting use, but not for synchronization control. It might be an expensive operation.

Specific thread

Many complex frameworks, and in particular those that involve event-handling and user interfaces, restrict access to specific methods to callers from specific threads. For instance, almost all user-interface operations in *Eclipse* can only be called from the dedicated user interface thread and thus only from listener code or from specially submitted tasks. Even in the standard JAVA library there are certain thread limitations. For instance, in an `AWTEvent`, the `getCurrentEvent` method can only be invoked from an application's event dispatching thread.

`getCurrentEvent`

```
public static AWTEvent getCurrentEvent()
```

Returns the event currently being dispatched by the `EventQueue` associated with the calling thread. This is useful if a method needs access to the event, but was not designed to receive a reference to it as an argument. Note that this method should only be invoked from an application's event dispatching thread. If this method is invoked from another thread, null will be returned.

4.4.4 Object state qualifications

A particularly interesting form of restriction is one that is based on object state. In other words, calls are only allowed or disallowed if the object on which the method is invoked (or the object making the invocation) is in a specific state. The problem with these restrictions is that they might be difficult to determine statically, as the same call can be legal at certain points in time and illegal in others. This is related to the concept of *typestates* [7].

For example, an *SQL* transaction cannot be called while auto-commit mode is active on the `Connection`.

`rollback`

```
void rollback()  
    throws SQLException
```

Undoes all changes made in the current transaction and releases any database locks currently held by this `Connection` object. This method should be used only when auto-commit mode has been disabled.

In AWT, attempting to get the `Graphics` object for an `Image` is only permitted when the object is off the screen.

`getGraphics`

```
public abstract Graphics getGraphics()
```

Creates a graphics context for drawing to an off-screen image. This method can only be called for off-screen images.

Also in AWT, the `getUnitsToScroll` method in `MouseEvent` is only meaningful under a specific scroll type.

getUnitsToScroll

```
public int getUnitsToScroll()
```

This is a convenience method to aid in the implementation of the common-case `MouseWheelListener` - to scroll a `ScrollPane` or `JScrollPane` by an amount which conforms to the platform settings. (Note, however, that `ScrollPane` and `JScrollPane` already have this functionality built in.)

This method returns the number of units to scroll when scroll type is `MouseEvent.WHEEL_UNIT_SCROLL`, and should only be called if `getScrollType` returns `MouseEvent.WHEEL_UNIT_SCROLL`.

Direction of scroll, amount of wheel movement, and platform settings for wheel scrolling are all accounted for. This method does not and cannot take into account value of the `Adjustable/Scrollable` unit increment, as this will vary among scrolling components.

A simplified example of how this method might be used in a listener:

In SWING, the `scrollToReference` method in `JEditorPane` is only meaningful when the component is not visible.

scrollToReference

```
public void scrollToReference(String reference)
```

Scrolls the view to the given reference location (that is, the value returned by the `UL.getRef` method for the URL being displayed). By default, this method only knows how to locate a reference in an `HTMLDocument`. The implementation calls the `scrollRectToVisible` method to accomplish the actual scrolling. If scrolling to a reference location is needed for document types other than HTML, this method should be reimplemented. This method will have no effect if the component is not visible.

The `notify` threading construct in the standard `Object` should only be invoked by a thread that already owns the monitor of this object.

notify

```
public final void notify()
```

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the `wait` methods.

The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:

- By executing a synchronized instance method of that object.
- By executing the body of a `synchronized` statement that synchronizes on the object.
- For objects of type `Class`, by executing a synchronized static method of that class.

Only one thread at a time can own an object's monitor.

Throws:

[IllegalMonitorStateException](#) - if the current thread is not the owner of this object's monitor.

See Also:

[notifyAll\(\)](#), [wait\(\)](#)

4.4.5 Discourage use

We have also encountered examples where a client is allowed to invoke a method, but is discouraged from doing so. We survey several types of such restrictions below and will see an additional form of this type when we later describe *alternative directives*.

Indicating that there is no benefit for using

A common variation of stating that a method is meant for internal use is to state that calling from other locations is allowed but is not likely to provide meaningful value.

For example, the `ordinal` value of enumerations should not be used but is in fact used in many programs.

ordinal

```
public final int ordinal()
```

Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero). Most programmers will have no use for this method. It is designed for use by sophisticated enum-based data structures, such as [EnumSet](#) and [EnumMap](#).

Returns:

the ordinal of this enumeration constant

Similarly, purging timers have limited impact in most cases.

purge

```
public int purge()
```

Removes all cancelled tasks from this timer's task queue. *Calling this method has no effect on the behavior of the timer*, but eliminates the references to the cancelled tasks from the queue. If there are no external references to these tasks, they become eligible for garbage collection.

Most programs will have no need to call this method. It is designed for use by the rare application that cancels a large number of tasks. Calling this method trades time for space: the runtime of the method may be proportional to $n + c \log n$, where n is the number of tasks in the queue and c is the number of cancelled tasks.

Note that it is permissible to call this method from within a task scheduled on this timer.

Callers to `drawBytes` in the AWT's `Graphics` class are told not to use it due to an issue with unicode.

drawBytes

```
public void drawBytes(byte[] data,  
                    int offset,  
                    int length,  
                    int x,  
                    int y)
```

Draws the text given by the specified byte array, using this graphics context's current font and color. The baseline of the first character is at position (x, y) in this graphics context's coordinate system.

Use of this method is not recommended as each byte is interpreted as a Unicode code point in the range 0 to 255, and so can only be used to draw Latin characters in that range.

Debugging

A variation on this type of restriction is to state that the method is meant for debugging purposes and therefore its values are inaccurate or may change over time. For example, the values returned by `actionCount` in `threadGroup` should only be used for informational purposes.

activeCount

```
public int activeCount()
```

Returns an estimate of the number of active threads in this thread group. The result might not reflect concurrent activity, and might be affected by the presence of certain system threads.

Due to the inherently imprecise nature of the result, it is recommended that this method only be used for informational purposes.

Highlighting severe risk of incorrect use

Another way of discouraging use is to warn callers of severe consequences if the method is used incorrectly or in the wrong context.

For example, the documentation of `halt` in the `Runtime` class encourages “extreme caution” since it avoids an orderly shutdown.

halt

```
public void halt(int status)
```

Forcibly terminates the currently running Java virtual machine. This method never returns normally.

This method should be used with extreme caution. Unlike the [exit](#) method, this method does not cause shutdown hooks to be started and does not run uninvoked finalizers if finalization-on-exit has been enabled. If the shutdown sequence has already been initiated then this method does not wait for any running shutdown hooks or finalizers to finish their work.

Parameters:

`status` - Termination status. By convention, a nonzero status code indicates abnormal termination. If the [exit](#) (equivalently, [System.exit](#)) method has already been invoked then this status code will override the status code passed to that method.

Throws:

[SecurityException](#) - If a security manager is present and its [checkExit](#) method does not permit an exit with the specified status

Since:

1.3

See Also:

[exit\(int\)](#), [addShutdownHook\(java.lang.Thread\)](#), [removeShutdownHook\(java.lang.Thread\)](#)

In the AWT’s drag-and-drop infrastructure, callers to `addNotify` or `removeNotify` in the `DropTarget` class are warned that misuse can break the whole infrastructure.

addNotify

```
public void addNotify(java.awt.peer.ComponentPeer peer)
```

Notify the `DropTarget` that it has been associated with a `Component`. This method is usually called from `java.awt.Component.addNotify()` of the `Component` associated with this `DropTarget` to notify the `DropTarget` that a `ComponentPeer` has been associated with that `Component`. Calling this method, other than to notify this `DropTarget` of the association of the `ComponentPeer` with the `Component` may result in a malfunction of the DnD system.

In the AWT’s central `Graphics2D` class, callers to `setTransform` are told to never apply transforms on top of another.

setTransform

```
public abstract void setTransform(AffineTransform Tx)
```

Overwrites the `Transform` in the `Graphics2D` context. **WARNING:** This method should **never** be used to apply a new coordinate transform on top of an existing transform because the `Graphics2D` might already have a transform that is needed for other purposes, such as rendering Swing components or applying a scaling transformation to adjust for the resolution of a printer.

To add a coordinate transform, use the `transform`, `rotate`, `scale`, or `shear` methods. The `setTransform` method is intended only for restoring the original `Graphics2D` transform after rendering, as shown in this example:

4.5 Imperative directives - Alternatives

The restriction directives discussed in the previous section demand or strongly encourage that a method not be invoked. Implicitly, however, many of them suggest that prospective caller seeking to accomplish a particular goal seek alternative ways to meet them. This section describes a related group of directives, which we call “alternatives”. These directives are similar to restrictions in the sense that they discourage the use of the method. However, they typically do not forbid its use, and more importantly, they explicitly state which method may be a better fit.

As we shall see, while certain alternatives are only preferable due to “style” and architectural concerns, many others provide a different and often the only correct behavior. A lack of awareness of these

directives may therefore cause callers and maintainers to settle for an incorrect call, which may lead to breakdowns.

Perhaps because of the conceptual similarity between restrictions and alternatives, we found that many examples of alternatives correspond to the types we have found for restriction directives. The structure of our presentation is therefore similar.

4.5.1 Deprecated methods

Regardless of whether the `@deprecated` tag is used, many obsolete methods explicitly state the new alternative which callers should use. In the standard library and in major APIs, calling an obsolete method may be safe as they are designed for backwards compatibility. In proprietary APIs, however, such compatibility is rarely ensured, and calling such methods may result in a failure.

Below are several examples of alternative directives in obsolete methods:

addItem

```
public void addItem(String item)
```

Obsolete as of Java 2 platform v1.1. Please use the `add` method instead.

Adds an item to this `Choice` menu.

isModal

```
public boolean isModal()
```

Indicates whether the dialog is modal.

This method is obsolete and is kept for backwards compatibility only. Use [getModalityType\(\)](#) instead.

contains

```
public boolean contains(Object value)
```

Legacy method testing if some key maps into the specified value in this table. This method is identical in functionality to [containsValue\(\[java.lang.Object\]\(#\)\)](#), and exists solely to ensure full compatibility with class [Hashtable](#), which supported this method prior to introduction of the Java Collections framework.

setHorizontalScrollBarPolicy

```
public void setHorizontalScrollBarPolicy(int x)
```

Sets the horizontal scrollbar-display policy. The options are:

- `ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED`
- `ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER`
- `ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS`

Note: Applications should use the `JScrollPane` version of this method. It only exists for backwards compatibility with the Swing 1.0.2 (and earlier) versions of this class.

4.5.2 Specific callers

We have seen that many API methods are meant for use by specific callers and that some merely state the permitted callers. Other methods, however, explicitly state alternative calls that can be made to achieve similar goals.

Internal callers

We have encountered many cases where implementation-specific methods are `public` for the reasons demonstrated earlier, but are meant to be used through a different object or method, or through another part of the usage model. While use of the internal method may not cause immediate harm, it increases coupling and thus future risk.

The SWING toolkit provides many examples of these issues in central classes. For example, the `JDesktopPane` class, offers a method called `setSelectedFrame` for selecting a specific internal frame in the pane. However, developers are actually supposed to call `setSelected` on the internal frame object.

`setSelectedFrame`

```
public void setSelectedFrame(JInternalFrame f)
```

Sets the currently active `JInternalFrame` in this `JDesktopPane`. This method is used to bridge the package gap between `JDesktopPane` and the platform implementation code and should not be called directly. To visually select the frame the client must call `JInternalFrame.setSelected(true)` to activate the frame.

A set of additional examples involves the complex focusing mechanism of the SWING and AWT toolkits, which provide many methods that are used by widgets and window to handle and transfer focus. One of these operations is the `grabFocus` method in the central `JComponent` class. As can be seen below, this method is meant for use by the implementations, so clients who want the widget to take focus need to use the `requestFocusInWindow` method instead.

`grabFocus`

```
public void grabFocus()
```

Requests that this Component get the input focus, and that this Component's top-level ancestor become the focused Window. This component must be displayable, visible, and focusable for the request to be granted.

This method is intended for use by focus implementations. Client code should not use this method; instead, it should use `requestFocusInWindow()`.

Avoiding the use of inherited methods

The above example is interesting since both method names (`grabFocus` and `requestFocusInWindow`) appear to meet the goals of the clients. As we shall later demonstrate in our study, two alternatives with similar names pose significant awareness problems. We note, however, that there is actually a third such method. The AWT `Component` class, from which the `SWINGJComponent` class inherits, defines a `requestFocus` method that is `public` and therefore also inherited into `JComponent`. The documentation for `requestFocusInWindow`, shown below, elaborates this.

`requestFocusInWindow`

```
public boolean requestFocusInWindow()
```

Requests that this Component get the input focus, if this Component's top-level ancestor is already the focused Window. This component must be displayable, focusable, visible and all of its ancestors (with the exception of the top-level Window) must be visible for the request to be granted. Every effort will be made to honor the request; however, in some cases it may be impossible to do so. Developers must never assume that this Component is the focus owner until this Component receives a `FOCUS_GAINED` event.

This method returns a boolean value. If `false` is returned, the request is **guaranteed to fail**. If `true` is returned, the request will succeed **unless** it is vetoed, or an extraordinary event, such as disposal of the Component's peer, occurs before the request can be granted by the native windowing system. Again, while a return value of `true` indicates that the request is likely to succeed, developers must never assume that this Component is the focus owner until this Component receives a `FOCUS_GAINED` event.

This method cannot be used to set the focus owner to no Component at all. Use `KeyboardFocusManager.clearGlobalFocusOwner()` instead.

The focus behavior of this method can be implemented uniformly across platforms, and thus developers are strongly encouraged to use this method over `requestFocus` when possible. Code which relies on `requestFocus` may exhibit different focus behavior on different platforms.

Note: Not all focus transfers result from invoking this method. As such, a component may receive focus without this or any of the other `requestFocus` methods of `Component` being invoked.

The situation where methods inherited from the AWT classes confound the use of their SWING subclasses demonstrates the challenges of not having ways of removing inherited methods from a public interface.

A similar issue results from interface implementation. When we discussed restrictions, we saw examples of “listener” or “handler” interface methods that were made public due to language restrictions. The documentation of the restriction can be accompanied by a directive referring callers to the method that will trigger this listener as a side effect.

We note that while such listener methods are often easy to detect based on their naming, that is not always the case. In one of the layout managers, for example, we found the `addLayoutComponent` method that is actually an event handler that should have been named better. Its documentation clarifies how callers can actually add the component, which will eventually bring about this notification.

addLayoutComponent

```
public void addLayoutComponent(Component component,  
                               Object constraints)
```

Notification that a `Component` has been added to the parent container. You should not invoke this method directly, instead you should use one of the `Group` methods to add a `Component`.

In complex event-driven systems such as GUI toolkits, it is easy for a caller to become confused about the correct “entry point” for a particular behavior. This is the case with painting components, as we can see below.

paint

```
public void paint(Graphics g)
```

Invoked by Swing to draw components. Applications should not invoke `paint` directly, but should instead use the `repaint` method to schedule the component for redrawing.

This method actually delegates the work of painting to three protected methods: `paintComponent`, `paintBorder`, and `paintChildren`. They're called in the order listed to ensure that children appear on top of component itself. Generally speaking, the component and its children should not paint in the insets area allocated to the border. Subclasses can just override this method, as always. A subclass that just wants to specialize the UI (look and feel) delegate's `paint` method should just override `paintComponent`.

4.5.3 Use in special cases or purposes

Use alternative for specific rare subcases

A major difference between restriction directives and alternatives directives is that the latter rarely forbid the use of the method. One common reason, discussed below, is that the method is actually useful in most cases, and the alternative is reserved for special cases and situations that are important enough to point out.

For example, in the AWT `ColorSpace` class, the methods for converting to and from RGB prescribe other methods for “colorimetric conversions”

toRGB

```
public abstract float[] toRGB(float[] colorvalue)
```

Transforms a color value assumed to be in this `ColorSpace` into a value in the default `CS_sRGB` color space.

This method transforms color values using algorithms designed to produce the best perceptual match between input and output colors. In order to do colorimetric conversion of color values, you should use the `toCIEXYZ` method of this color space to first convert from the input color space to the `CS_CIEXYZ` color space, and then use the `fromCIEXYZ` method of the `CS_sRGB` color space to convert from `CS_CIEXYZ` to the output color space. See [toCIEXYZ](#) and [fromCIEXYZ](#) for further information.

In the standard interface for double-ended queues, it is recommended that `offerFirst` be used

when dealing with capacity-restricted queues to check whether an element can actually be added.

addFirst

```
void addFirst(E e)
```

Inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions. When using a capacity-restricted deque, it is generally preferable to use method [offerFirst\(E\)](#).

In SWING, a non-default constructor must be called to create “unowned” `JDialog` objects; most dialogs are owned, making the default acceptable.

JDialog

```
public JDialog()
```

Creates a modeless dialog without a title and without a specified `Frame` owner. A shared, hidden frame will be set as the owner of the dialog.

This constructor sets the component's locale property to the value returned by `JComponent.getDefaultLocale`.

NOTE: This constructor does not allow you to create an unowned `JDialog`. To create an unowned `JDialog` you must use either the `JDialog(Window)` or `JDialog(Dialog)` constructor with an argument of `null`.

Use this for specific rare circumstances

While the documentation of the commonly-used method may prescribe the alternative to use for special purposes, the documentation of such alternatives may refer back to the commonly-used method for the general case.

For example, painting is a major performance issue in most GUI toolkits. Accordingly, `Swing` users are expected to ask the toolkit to file a repaint request which will be handled in the next cycle. The method for requesting an immediate repaint, presented below, acknowledges that there are situations where immediate painting is necessary, but points out the preferred behavior to its callers.

paintImmediately

```
public void paintImmediately(int x,  
                             int y,  
                             int w,  
                             int h)
```

Paints the specified region in this component and all of its descendants that overlap the region, immediately.

It's rarely necessary to call this method. In most cases it's more efficient to call `repaint`, which defers the actual painting and can collapse redundant requests into a single paint call. This method is useful if one needs to update the display while the current event is being dispatched.

Similarly, there may be rare situations where callers may want to abruptly shut down the JAVA virtual machine, but the documentation of `Runtime.exit()` highly recommends the more gentle use of `System.exit()`.

exit

```
public void exit(int status)
```

Terminates the currently running Java virtual machine by initiating its shutdown sequence. This method never returns normally. The argument serves as a status code; by convention, a nonzero status code indicates abnormal termination.

The virtual machine's shutdown sequence consists of two phases. In the first phase all registered [shutdown hooks](#), if any, are started in some unspecified order and allowed to run concurrently until they finish. In the second phase all uninvoked finalizers are run if [finalization-on-exit](#) has been enabled. Once this is done the virtual machine [halts](#).

If this method is invoked after the virtual machine has begun its shutdown sequence then if shutdown hooks are being run this method will block indefinitely. If shutdown hooks have already been run and on-exit finalization has been enabled then this method halts the virtual machine with the given status code if the status is nonzero; otherwise, it blocks indefinitely.

The [System.exit](#) method is the conventional and convenient means of invoking this method.

When dealing with font transforms in AWT, the `getTransform` method recognizes that most fonts are not transformed and therefore recommend that `isTransformed` be invoked first to see if there is even a need to call this method.

getTransform

```
public AffineTransform getTransform()
```

Returns a copy of the transform associated with this `Font`. This transform is not necessarily the one used to construct the font. If the font has algorithmic superscripting or width adjustment, this will be incorporated into the returned `AffineTransform`.

Typically, fonts will not be transformed. Clients generally should call [isTransformed\(\)](#) first, and only call this method if `isTransformed` returns true.

When dealing with the new JAVA enumerations, there is sometimes a need to print the name of a value. The documentation encourages the use of `toString` for readability, stating that this method is designed to ensure getting a consistent name between releases.

name

```
public final String name()
```

Returns the name of this enum constant, exactly as declared in its enum declaration. **Most programmers should use the [toString\(\)](#) method in preference to this one, as the `toString` method may return a more user-friendly name.** This method is designed primarily for use in specialized situations where correctness depends on getting the exact name, which will not vary from release to release.

4.5.4 Discouraged use

As we saw with restrictions, the documentation of many methods attempts to discourage their use. In some cases, as we shall now see, it also offers a preferable alternative.

Discouraging use for specific goals

When developers are attempting to accomplish a particular goal, they examine available classes and methods looking for a potential match `match`. In some cases, the authors of a method may be aware of the likelihood that their method may be used to accomplish a particular goal for which it was not designed and issue a warning in the documentation.

For example, a common but inherently unreliable programming practice is to use empty disk files as locks. The `createNewFile` method in the standard `File` class explicitly warns callers against using it to establish this goal.

createNewFile

```
public boolean createNewFile()  
    throws IOException
```

Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist. The check for the existence of the file and the creation of the file if it does not exist are a single operation that is atomic with respect to all other filesystem activities that might affect the file.

Note: this method should *not* be used for file-locking, as the resulting protocol cannot be made to work reliably. The [FileLock](#) facility should be used instead.

Similarly, developers seeking to use the `weakCompareAndSet` method in the standard `AtomicInteger` are discouraged from relying it for comparing and setting, and are referred to the standard version.

weakCompareAndSet

```
public final boolean weakCompareAndSet(int expect,  
    int update)
```

Atomically sets the value to the given updated value if the current value == the expected value.

May [fail spuriously](#) and does not provide ordering guarantees, so is only rarely an appropriate alternative to `compareAndSet`.

Constructors

As mentioned earlier, some objects can be created and initialized in a variety of ways, with specific methods intended for specific uses. In addition, the use of factory methods is often preferable to the use of a constructor, and the constructors of singleton objects should never be called by users.

LayoutStyle

```
public LayoutStyle()
```

Creates a new `LayoutStyle`. You generally don't create a `LayoutStyle`. Instead use the method `getInstance` to obtain the current `LayoutStyle`.

getLayoutStyle

```
public LayoutStyle getLayoutStyle()
```

Returns the `LayoutStyle` for this look and feel. This never returns `null`.

You generally don't use the `LayoutStyle` from the look and feel, instead use the `LayoutStyle` method `getInstance`.

RepaintManager

```
public RepaintManager()
```

Create a new `RepaintManager` instance. You rarely call this constructor directly. To get the default `RepaintManager`, use `RepaintManager.currentManager(JComponent)` (normally "this").

Debugging

As with restrictions, the use of certain methods is discouraged since they are meant for informational purposes, but this time an alternative is explicitly proposed.

For example, asking a `Component` in AWT for its location may return outdated values, so it is better to listen for movements and capture them in the event handler.

getLocation

```
public Point getLocation()
```

Gets the location of this component in the form of a point specifying the component's top-left corner. The location will be relative to the parent's coordinate space.

Due to the asynchronous nature of native event handling, this method can return outdated values (for instance, after several calls of `setLocation()` in rapid succession). For this reason, the recommended method of obtaining a component's position is within `java.awt.event.ComponentListener.componentMoved()`, which is called after the operating system has finished moving the component.

Also in the AWT, calls to `TextLayout.getCaretInfo()`, should be replaced with calls to `getCaretShape`. Interestingly, this directive does not appear in the second overload.

getCaretInfo

```
public float[] getCaretInfo(TextHitInfo hit,  
                             Rectangle2D bounds)
```

Returns information about the caret corresponding to `hit`. The first element of the array is the intersection of the caret with the baseline, as a distance along the baseline. The second element of the array is the inverse slope (run/rise) of the caret, measured with respect to the baseline at that point.

This method is meant for informational use. To display carets, it is better to use `getCaretShapes`.

getCaretInfo

```
public float[] getCaretInfo(TextHitInfo hit)
```

Returns information about the caret corresponding to `hit`. This method is a convenience overload of `getCaretInfo` and uses the natural bounds of this `TextLayout`.

```
getCaretShapes(int, Rectangle2D, TextLayout.CaretPolicy), Font.getItalicAngle()
```

Platform dependence

Some methods are platform-dependent, so callers are encouraged to use other calls. For example, in the AWT focus subsystem, the use of `requestFocus` is discouraged and `requestFocusInWindow` is preferred.

requestFocus

```
public void requestFocus()
```

Requests that this `Component` gets the input focus. Refer to [Component.requestFocus\(\)](#) for a complete description of this method.

Note that the use of this method is discouraged because its behavior is platform dependent. Instead we recommend the use of [requestFocusInWindow\(\)](#). If you would like more information on focus, see [How to Use the Focus Subsystem](#), a section in *The Java Tutorial*.

4.5.5 Better fit

Certain methods are useful for many callers and operate as expected, but they may be limited in their accuracy compared to some (possibly more expensive calls).

For example, the definition of bounds in the AWT toolkit is that a rectangle can fully enclose the shape. However, the standard `getBounds` call does not always return the tightest fit, due to representation and calculation issues. In the case of `Arc2D`, the difference can be significant, so callers are warned about this.

getBounds

```
public Rectangle getBounds()
```

Returns an integer [Rectangle](#) that completely encloses the shape. Note that there is no guarantee that the returned `Rectangle` is the smallest bounding box that encloses the shape, only that the shape lies entirely within the indicated `Rectangle`. The returned `Rectangle` might also fail to completely enclose the shape if the shape overflows the limited range of the integer data type. The `getBounds2D` method generally returns a tighter bounding box due to its greater flexibility in representation.

The SWING `JLayeredPane` class offers the ability to organize components into layers, so that elements in higher layer block the view of those in lower layers. To place an element in a layer, one can use `putLayer`, but that does not lead to a repainting, so callers are encouraged to use `setLayer`. As we shall later see in our study, this presents a significant problem.

```

void javax.swing.JLayeredPane.putLayer(JComponent c, int layer)

```

Sets the layer property on a JComponent. This method does not cause any side effects like setLayer() (painting, add/remove, etc). Normally you should use the instance method setLayer(), in order to get the desired side-effects (like repainting).

See Also:
[setLayer](#)

Parameters:
c the JComponent to move
layer an int specifying the layer to move it to

Similarly, callers to `checkImage` on an AWT Component, which checks on the construction of an image, are instructed to call `prepareImage` to actually make the image start loading.

checkImage

```

public int checkImage(Image image,
                     ImageObserver observer)

```

Returns the status of the construction of a screen representation of the specified image.

This method does not cause the image to begin loading. An application must use the `prepareImage` method to force the loading of an image.

Information on the flags returned by this method can be found with the discussion of the `ImageObserver` interface.

Many properties in UI toolkits are merely hints that are up to specific implementations to honor. In the focus subsystem of SWING, a call to `setRequestFocusEnabled` merely offers a hint on whether the component can get focus. To fully prevent it from getting focus, however, one needs to call `setFocusable`.

setRequestFocusEnabled

```

public void setRequestFocusEnabled(boolean requestFocusEnabled)

```

Provides a hint as to whether or not this `JComponent` should get focus. This is only a hint, and it is up to consumers that are requesting focus to honor this property. This is typically honored for mouse operations, but not keyboard operations. For example, look and feels could verify this property is true before requesting focus during a mouse operation. This would often times be used if you did not want a mouse press on a `JComponent` to steal focus, but did want the `JComponent` to be traversable via the keyboard. If you do not want this `JComponent` focusable at all, use the `setFocusable` method instead.

Please see [How to Use the Focus Subsystem](#), a section in *The Java Tutorial*, for more information.

In SWING, the way in which tables are printed can be customized. Callers to `getPrintable` are discouraged from dealing with the `Printable` class directly if they simply wish to print the table.

getPrintable

```

public Printable getPrintable(JTable.PrintMode printMode,
                             MessageFormat headerFormat,
                             MessageFormat footerFormat)

```

Return a `Printable` for use in printing this `JTable`.

This method is meant for those wishing to customize the default `Printable` implementation used by `JTable`'s print methods. Developers wanting simply to print the table should use one of those methods directly.

Finally, we note that some methods designate other methods as a more efficient way to achieve the same goal. Since calling the original is not an error, we consider these directives to be informative performance directives, and shall discuss them later.

4.6 Imperative directives - Protocols

Many methods are designed to be used as part of a sequence of actions that sets state and context. Their documentation includes protocol constraints that specify what should occur *before* and *after* the call. Common examples state that the method must: “*only be called once*”, “*be called prior to any calls to X*”, “*first call X*”, “*be the last call on this object*”. Since these statements demand an action from the caller,

we consider them to be directives.

Because such protocols are so important and prevalent, they are the focus of much research. Many recent techniques and tools (e.g., [31, 8, 57]) offer means for API authors to formally specify facets of the expected protocol, and for clients to automatically check the conformance of their programs against these specifications. While such tools are indeed highly effective in applicable situations, our API inspection also revealed many cases where writing such formal specifications may be difficult, expensive, or impractical. In the examples of this section, we will present examples of both types of protocol directives.

4.6.1 Instructions about repeated invocations

While most protocol directives deal with a sequence of calls to multiple functions, some deal with sequences that involve the same call multiple times.

Quite a few methods forbid repeated invocation, suggesting that a language level construct or at least an annotation should be created.¹ Standard threads provide a classic example of this restriction, as a thread cannot be started multiple times.

start

```
public void start()
```

Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.

The result is that two threads are running concurrently: the current thread (which returns from the call to the `start` method) and the other thread (which executes its `run` method).

It is never legal to start a thread more than once. In particular, a thread may not be restarted once it has completed execution.

Similarly, when creating a throwable object for an exception, it is forbidden to call `initCause` multiple times; this is actually surprising because the effect is likely to simply be the setting of a string field. In addition, depending on the state of the object (i.e., how it was initialized), calls to this method may be completely forbidden.

initCause

```
public Throwable initCause(Throwable cause)
```

Initializes the *cause* of this throwable to the specified value. (The cause is the throwable that caused this throwable to get thrown.)

This method can be called at most once. It is generally called from within the constructor, or immediately after creating the throwable. If this throwable was created with `Throwable(Throwable)` or `Throwable(String,Throwable)`, this method cannot be called even once.

In some cases, the restriction will not actually cause an ill effect, but rather have no effect.

execute

```
public final void execute()
```

Schedules this `SwingWorker` for execution on a *worker* thread. There are a number of *worker* threads available. In the event all *worker* threads are busy handling other `SwingWorkers` this `SwingWorker` is placed in a waiting queue.

Note: `SwingWorker` is only designed to be executed once. Executing a `SwingWorker` more than once will not result in invoking the `doInBackground` method twice.

Things become more complicated as certain methods may only be called once per “state”, and when the state changes they may be callable again. For example, result sets in SQL are created for specific results, and cannot be retrieved twice for the same result. Formal specification would require a specification of what consists the current result and a modeling of result changes.

¹JAVA 5 allows metadata elements called *annotations* to be associated with methods, and automatic tools can check conformance. For example, methods tagged as `override` must indeed override an inherited version.

getResultSet

`ResultSet` `getResultSet()`
throws [SQLException](#)

Retrieves the current result as a `ResultSet` object. This method should be called only once per result.

Iterators are a good example of objects that reflect some underlying state, so certain operations cannot be repeated. The exact rules can become quite complex, as they are for removing an element on an SQL list iterator.

remove

`void remove()`

Removes from the list the last element that was returned by `next` or `previous` (optional operation). This call can only be made once per call to `next` or `previous`. It can be made only if `ListIterator.add` has not been called after the last call to `next` or `previous`.

The definition of a repeated call becomes less clear with recursive calls that return objects of the same type, a common pattern for things such as transforms. In the example from AWT below, asking a text layout object for a justified layout object produces a new object, but the call is not permitted if the element is already justified. In addition, in certain states a repeated call has no effect.

getJustifiedLayout

`public TextLayout getJustifiedLayout(float justificationWidth)`

Creates a copy of this `TextLayout` justified to the specified width.

If this `TextLayout` has already been justified, an exception is thrown. If this `TextLayout` object's justification ratio is zero, a `TextLayout` identical to this `TextLayout` is returned.

Finally, repeated-invocation restrictions can be combined with rules about other calls. For instance, in AWT's `InputMethod`, one may only call `setInputMethodContext` once, but there are also precise instructions on when that one call should occur.

setInputMethodContext

`void setInputMethodContext(InputMethodContext context)`

Sets the input method context, which is used to dispatch input method events to the client component and to request information from the client component.

This method is called once immediately after instantiating this input method.

We note that while methods that explicitly address repeated calls typically forbid it, some explicitly allow it, such as the `cancel` operation on a `Timer`.

cancel

`public void cancel()`

Terminates this timer, discarding any currently scheduled tasks. Does not interfere with a currently executing task (if it exists). Once a timer has been terminated, its execution thread terminates gracefully, and no more tasks may be scheduled on it.

Note that calling this method from within the run method of a timer task that was invoked by this timer absolutely guarantees that the ongoing task execution is the last task execution that will ever be performed by this timer.

This method may be called repeatedly; the second and subsequent calls have no effect.

Some methods anecdotally specify that they are the only ones that can be called after the initial call, as demonstrated by the `free` operation from the SQL `Array`.

free

```
void free()  
    throws SQLException
```

This method frees the `Array` object and releases the resources that it holds. The object is invalid once the `free` method is called.

After `free` has been called, any attempt to invoke a method other than `free` will result in a `SQLException` being thrown. If `free` is called multiple times, the subsequent calls to `free` are treated as a no-op.

4.6.2 Placement in sequence

A typical protocol directive will convey a constraint on the placement of a call to the associated method in some sequence of calls. If other calls have already been made, then the directive should lead the caller to ensure that the placement is correct. If they have not, then the directive may actually remind the caller that it is necessary to invoke other methods.

Examples of such directives are quite common. For example, the documentation of `init` in the `JAVA Applet` class clarifies the initialization process: First, `init` is called by the containing viewer, and only later is a call to `start` made; the wording indicates that the subsequent call will always occur at some point and possibly more than once.

init

```
public void init()
```

Called by the browser or applet viewer to inform this applet that it has been loaded into the system. It is always called before the first time that the `start` method is called.

A subclass of `Applet` should override this method if it has initialization to perform. For example, an applet with threads would use the `init` method to create the threads and the `destroy` method to kill them.

The implementation of this method provided by the `Applet` class does nothing.

Similarly, the documentation of `start` in `Applet` shows the corresponding rule, alerting callers that a call to `init` will have to occur first. It also permits multiple calls to this method.

start

```
public void start()
```

Called by the browser or applet viewer to inform this applet that it should start its execution. It is called after the `init` method and each time the applet is revisited in a Web page.

A subclass of `Applet` should override this method if it has any operation that it wants to perform each time the Web page containing it is visited. For example, an applet with animation might want to use the `start` method to resume animation, and the `stop` method to suspend the animation.

Note: some methods, such as `getLocationOnScreen`, can only provide meaningful results if the applet is showing. Because `isShowing` returns `false` when the applet's `start` is first called, methods requiring `isShowing` to return `true` should be called from a `ComponentListener`.

The implementation of this method provided by the `Applet` class does nothing.

Protocol directives are particularly important when the method name matches the expectations of a user seeking to accomplish a particular goal, but additional actions are needed. For example, developers seeking to add widgets to a `SWING` container need to be aware of the need to invoke `validate` to actually cause the container to be redrawn with the new widgets.

void java.awt.Container.add(Component comp, Object constraints)

Adds the specified component to the end of this container. Also notifies the layout manager to add the component to this container's layout using the specified constraints object. This is a convenience method for [addImpl](#).

Note: If a component has been added to a container that has been displayed, `validate` must be called on that container to display the new component. If multiple components are being added, you can improve efficiency by calling `validate` only once, after all the components have been added.

Parameters:

`comp` the component to be added
`constraints` an object expressing layout constraints for this component

Since:

JDK1.1

See Also:

[addImpl](#)
[validate](#)
[javax.swing.JComponent.revalidate\(\)](#)
[LayoutManager](#)

Similarly, ignoring deprecation, the impact of calling `setModal` is not evident until the dialog visible state is reset with other methods.

setModal

```
public void setModal(boolean modal)
```

Specifies whether this dialog should be modal.

This method is obsolete and is kept for backwards compatibility only. Use [setModalityType\(\)](#) instead.

Note: changing modality of the visible dialog may have no effect until it is hidden and then shown again.

In some cases, the documentation is so verbose that there is a risk that readers may not get to the main point - the necessary sequence. For example, only the last line tells when to invoke `preferProportionalFonts` in SWING's `GraphicsEnvironment` class.

preferProportionalFonts

```
public void preferProportionalFonts()
```

Indicates a preference for proportional over non-proportional (e.g. dual-spaced CJK fonts) fonts in the mapping of logical fonts to physical fonts. If the default mapping contains fonts for which proportional and non-proportional variants exist, then calling this method indicates the mapping should use a proportional variant.

The actual change in font rendering behavior resulting from a call to this method is implementation dependent; it may have no effect at all. The behavior may differ between font rendering in lightweight and peered components. Since calling this method requests a different font, clients should expect different metrics, and may need to recalculate window sizes and layout. Therefore this method should be called before user interface initialisation.

Similarly, only the last sentence instructs when to call `setFocusableWindowState` on a SWING Window, and what should be done afterwards. I argue that readers may benefit if the instruction would appear *before* the explanation.

setFocusableWindowState

```
public void setFocusableWindowState(boolean focusableWindowState)
```

Sets whether this Window can become the focused Window if it meets the other requirements outlined in `isFocusableWindow`. If this Window's focusable Window state is set to `false`, then `isFocusableWindow` will return `false`. If this Window's focusable Window state is set to `true`, then `isFocusableWindow` may return `true` or `false` depending upon the other requirements which must be met in order for a Window to be focusable.

Setting a Window's focusability state to `false` is the standard mechanism for an application to identify to the AWT a Window which will be used as a floating palette or toolbar, and thus should be a non-focusable Window. Setting the focusability state on a visible window can have a delayed effect on some platforms — the actual change may happen only when the window becomes hidden and then visible again. To ensure consistent behavior across platforms, set the window's focusable state when the window is invisible and then show it.

An interesting form of sequence protocol defines that a call must take during the execution of another method. For example, calling `defaultReadObject` on an `ObjectInputStream` is only allowed from the `readObject` method or from a method invoked by it.

defaultReadObject

```
public void defaultReadObject()
    throws IOException,
           ClassNotFoundException
```

Read the non-static and non-transient fields of the current class from this stream. This may only be called from the `readObject` method of the class being deserialized. It will throw the `NotActiveException` if it is called otherwise.

Finally, the sequence may be described as an algorithm rather than as a linear sequence of calls. For example, `nextToken` on `StreamTokenizer` is invoked as part of a loop.

nextToken

```
public int nextToken()
    throws IOException
```

Parses the next token from the input stream of this tokenizer. The type of the next token is returned in the `tttype` field. Additional information about the token may be in the `nval` field or the `sval` field of this tokenizer.

Typical clients of this class first set up the syntax tables and then sit in a loop calling `nextToken` to parse successive tokens until `TT_EOF` is returned.

4.6.3 Directives that depend on state

The directives presented so far described protocols in terms of specific method invocations, or in terms of specific actions that likely correspond to a single method, such as making a widget visible. Such protocols are relatively easier to formally specify and validate. There are some directives, however, that work in more general terms of state. Because there may be different ways to reach the specified state, formal specification and validation may be more difficult.

For example, the `setDaemon` method on a standard Java thread may only be invoked before the thread had been started. From the point of view of the caller, the thread can be started with a call to `start`, but it may indirectly be started by other methods.

setDaemon

```
public final void setDaemon(boolean on)
```

Marks this thread as either a daemon thread or a user thread. The Java Virtual Machine exits when the only threads running are all daemon threads.

This method must be called before the thread is started.

This method first calls the `checkAccess` method of this thread with no arguments. This may result in throwing a `SecurityException` (in the current thread).

The standard `ThreadGroup` presents a more challenging example. Prior to destroying the group, callers must ensure that it is empty, and that all of its threads were stopped. It is not immediately clear how this can be verified, although one may assume that corresponding methods exist for writing the relevant code. Nevertheless, specifying this is not trivial, as many invocation sequences may be followed to accomplish this goal.

destroy

```
public final void destroy()
```

Destroys this thread group and all of its subgroups. This thread group must be empty, indicating that all threads that had been in this thread group have since stopped.

First, the `checkAccess` method of this thread group is called with no arguments; this may result in a security exception.

Similarly, the `close` and `getWarnings` operations may not be invoked on a closed SQL Connection. However, there may be different ways to close a connection in addition to a call to `close`, such as not opening it in the first place, or perhaps failures of other operations.

close

```
void close()  
    throws SQLException
```

Releases this `Connection` object's database and JDBC resources immediately instead of waiting for them to be automatically released.

Calling the method `close` on a `Connection` object that is already closed is a no-op.

It is **strongly recommended** that an application explicitly commits or rolls back an active transaction prior to calling the `close` method. If the `close` method is called and there is an active transaction, the results are implementation-defined.

getWarnings

```
SQLException getWarnings()  
    throws SQLException
```

Retrieves the first warning reported by calls on this `Connection` object. If there is more than one warning, subsequent warnings will be chained to the first one and can be retrieved by calling the method `SQLException.getNextWarning` on the warning that was retrieved previously.

This method may not be called on a closed connection; doing so will cause an `SQLException` to be thrown.

Note: Subsequent warnings will be chained to this `SQLException`.

Certain frameworks such as SQL support revolve around operations on objects that change their state in response to other operations. For example, various operations change the SQL cursor, so the legality

of a call to `insertRow` on an `SQL ResultSet` may depend on previous calls on this object.

insertRow

```
void insertRow()  
    throws SQLException
```

Inserts the contents of the insert row into this `ResultSet` object and into the database. The cursor must be on the insert row when this method is called.

Similarly, the `wasNull` operation on an `SQL CallableStatement` can only be invoked after some “getter” method had been invoked. There could be different getter methods, and there could possibly be calls to other methods.

wasNull

```
boolean wasNull()  
    throws SQLException
```

Retrieves whether the last OUT parameter read had the value of `SQL NULL`. Note that this method should be called only after calling a getter method; otherwise, there is no value to use in determining whether it is `null` or not.

Another example of state dependence comes from the `SWING JFileChooser` class, whose `setAccessory` method requires callers to check if there are any previous accessories that still have registered listeners. Verifying this is quite complex.

setAccessory

```
public void setAccessory(JComponent newAccessory)
```

Sets the accessory component. An accessory is often used to show a preview image of the selected file; however, it can be used for anything that the programmer wishes, such as extra custom file chooser controls.

Note: if there was a previous accessory, you should unregister any listeners that the accessory might have registered with the file chooser.

We note that in some cases the exact state is unclear. For example, in the `SWING DesktopManager`, it is not clear how users indicate that they will begin dragging a component before invoking `beginDraggingFrame`.

beginDraggingFrame

```
void beginDraggingFrame(JComponent f)
```

This method is normally called when the user has indicated that they will begin dragging a component around. This method should be called prior to any `dragFrame()` calls to allow the `DesktopManager` to prepare any necessary state. Normally `f` will be a `JInternalFrame`.

4.7 Imperative directives - Parameters

One of the main roles of function documentation is to describe its inputs and outputs, accurately specify the conditions on the inputs, and the properties of the outputs. In this section we are concerned with inputs, while the next one deals with the outputs.

In `JAVA`, all method parameters are (generally) inputs, although a function can modify the state of passed parameters. Restrictions on parameters are essentially pre-conditions, and following them is critical for the correctness of the program. To convey instructions to callers, *JavaDoc* provides a `param` tag. The official guide to writing *JavaDocs* instructs writers to create these tags for every parameter “even in straightforward” cases. Indeed, in standard libraries and major APIs like *Eclipse*, I found that this instruction is indeed followed in most cases.

While many parameter clauses place restrictions on allowed inputs, not all of them can be considered directives. For example, many parameter descriptions indicate whether `null` values are permitted, and callers can be reasonably expected to check the documentation if in doubt. Under the criteria defined earlier, a clause is only a directive if it can be considered surprising or is rare or somewhat unexpected.

To illustrate these differences, we now survey a variety of such directives found in the JDK. As we shall see, many clauses are considered directives because they restrict the parameters further than what could be expected from the signature of the method. Interestingly, many of these directives will appear in the method writeup, rather than in the parameter summary that uses the `@param` tag.

4.7.1 Restricting simple parameter value by contents

Most parameter directives are concerned with restricting the legal values of parameters which can be passed, but the restriction can have different facets.

Some methods restrict the legal values of atomic values passed to primitive variables. These restrictions are interesting because they typically cannot be expressed using the type system, and typically cannot be validated at compile time.

For example, the `load` methods of the standard `Runtime` class (below) and the standard `System` class take a filename as a string. However, the documentation demands that the passed filename would be absolute, rather than relative. This may surprise callers, as it is common to specify files in relation to the current application or working directory.

load

```
public void load(String filename)
```

Loads the specified filename as a dynamic library. The filename argument must be a complete path name, (for example `Runtime.getRuntime().load("/home/avh/lib/libX11.so");`).

First, if there is a security manager, its `checkLink` method is called with the `filename` as its argument. This may result in a security exception.

This is similar to the method `loadLibrary(String)`, but it accepts a general file name as an argument rather than just a library name, allowing any file of native code to be loaded.

The method `System.load(String)` is the conventional and convenient means of invoking this method.

Similarly, when asking an `AppletContext` to load an image from a URL, the URL must be absolute, rather than one that is relative to the location from which the applet was loaded.

getImage

```
Image getImage(URL url)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument that is passed as an argument must specify an absolute URL.

This method always returns immediately, whether or not the image exists. When the applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

`url` - an absolute URL giving the location of the image.

Returns:

the image at the specified URL.

When creating a `TextLayout` object in the AWT toolkit, the string passed to the `string` parameter should actually represent a single paragraph of text. Note again how this is not mentioned in the `param` list.

TextLayout

```
public TextLayout(String string,  
                  Font font,  
                  FontRenderContext frc)
```

Constructs a `TextLayout` from a `String` and a `Font`. All the text is styled using the specified `Font`.

The `String` must specify a single paragraph of text, because an entire paragraph is required for the bidirectional algorithm.

Parameters:

`string` - the text to display
`font` - a `Font` used to style the text
`frc` - contains information about a graphics device which is needed to measure the text correctly. Text measurements can vary slightly depending on the device resolution, and attributes such as antialiasing. This parameter does not specify a translation between the `TextLayout` and user space.

Restrictions are not limited to strings. For example, when setting the maximal field size on an `SQL Statement`, callers are encouraged to set a value of at least 256 for “portability” reasons.

setMaxFieldSize

```
void setMaxFieldSize(int max)  
    throws SQLException
```

Sets the limit for the maximum number of bytes that can be returned for character and binary column values in a `ResultSet` object produced by this `Statement` object. This limit applies only to `BINARY`, `VARBINARY`, `LONGVARBINARY`, `CHAR`, `VARCHAR`, `NCHAR`, `NVARCHAR`, `LONGNVARCHAR` and `LONGVARCHAR` fields. If the limit is exceeded, the excess data is silently discarded. For maximum portability, use values greater than 256.

Parameters:

`max` - the new column size limit in bytes; zero means there is no limit

In some cases, values are actually restricted in ways that are unintuitive and completely unexpected. For example, we have seen that when using `replaceAll` in the standard `String` class must not include dollar signs or backslashes due to an implementation issue.

replaceAll

```
public String replaceAll(String regex,  
                        String replacement)
```

Replaces each substring of this string that matches the given [regular expression](#) with the given replacement.

An invocation of this method of the form `str.replaceAll(regex, repl)` yields exactly the same result as the expression

```
Pattern.compile(regex).matcher(str).replaceAll(repl)
```

Note that backslashes (`\`) and dollar signs (`$`) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string; see [Matcher.replaceAll](#). Use [Matcher.quoteReplacement\(java.lang.String\)](#) to suppress the special meaning of these characters, if desired.

Parameters:

`regex` - the regular expression to which this string is to be matched
`replacement` - the string to be substituted for each match

Returns:

The resulting `String`

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

A common convention for identifying long-value integer literals is to add an `L` character following all digits. When converting such literals to a string using `Long`'s `parseLong` method, however, this character must not be included.

parseLong

```
public static long parseLong(String s)
    throws NumberFormatException
```

Parses the string argument as a signed decimal `long`. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign '-' (`\u002D`) to indicate a negative value. The resulting `long` value is returned, exactly as if the argument and the radix 10 were given as arguments to the [parseLong\(java.lang.String, int\)](#) method.

Note that neither the character `L` (`\u004C`) nor `l` (`\u006C`) is permitted to appear at the end of the string as a type indicator, as would be permitted in Java programming language source code.

When inserting character sequences into a `StringBuilder`, passing a `null` value will for some reason actually add the corresponding four characters.

insert

```
public StringBuilder insert(int dstOffset,
    CharSequence s)
```

Inserts the specified `CharSequence` into this sequence.

The characters of the `CharSequence` argument are inserted, in order, into this sequence at the indicated offset, moving up any characters originally above that position and increasing the length of this sequence by the length of the argument `s`.

The result of this method is exactly the same as if it were an invocation of this object's `insert(dstOffset, s, 0, s.length())` method.

If `s` is `null`, then the four characters "null" are inserted into this sequence.

4.7.2 Restricting complex parameter values by contents

Unique restrictions on parameter values can also involve more complex types of parameters. These restrictions are even more difficult to specify or automatically validate.

For example, when building a custom cursor using the AWT `Toolkit` class, the given cursor image must not have multiple frames or the program will hang.

createCustomCursor

```
public Cursor createCustomCursor(Image cursor,
    Point hotSpot,
    String name)
    throws IndexOutOfBoundsException,
    HeadlessException
```

Creates a new custom cursor object. If the image to display is invalid, the cursor will be hidden (made completely transparent), and the hotspot will be set to (0, 0).

Note that multi-frame images are invalid and may cause this method to hang.

Parameters:

`cursor` - the image to display when the cursor is activated
`hotSpot` - the X and Y of the large cursor's hot spot; the `hotSpot` values must be less than the Dimension returned by `getBestCursorSize`
`name` - a localized description of the cursor, for Java Accessibility use

When building a `CubicCurve2D` and specifying points for a subdivision, the first point should be identical to the last.

subdivide

```
public static void subdivide(double[] src,
                             int srcoff,
                             double[] left,
                             int leftoff,
                             double[] right,
                             int rightoff)
```

Subdivides the cubic curve specified by the coordinates stored in the `src` array at indices `srcoff` through `(srcoff + 7)` and stores the resulting two subdivided curves into the two result arrays at the corresponding indices. Either or both of the `left` and `right` arrays may be `null` or a reference to the same array as the `src` array. Note that the last point in the first subdivided curve is the same as the first point in the second subdivided curve. Thus, it is possible to pass the same array for `left` and `right` and to use offsets, such as `rightoff` equals `(leftoff + 6)`, in order to avoid allocating extra storage for this common point.

Parameters:

`src` - the array holding the coordinates for the source curve
`srcoff` - the offset into the array of the beginning of the the 6 source coordinates
`left` - the array for storing the coordinates for the first half of the subdivided curve
`leftoff` - the offset into the array of the beginning of the the 6 left coordinates
`right` - the array for storing the coordinates for the second half of the subdivided curve
`rightoff` - the offset into the array of the beginning of the the 6 right coordinates

When working with the AWT `FocusTraversalPolicy`, the passed container object must be a cycle root of the passed component value or a different policy provider. Clearly, this would be difficult to verify or specify at compile time.

getComponentAfter

```
public abstract Component getComponentAfter(Container aContainer,
                                             Component aComponent)
```

Returns the Component that should receive the focus after aComponent. aContainer must be a focus cycle root of aComponent or a focus traversal policy provider.

Parameters:

`aContainer` - a focus cycle root of aComponent or focus traversal policy provider
`aComponent` - a (possibly indirect) child of aContainer, or aContainer itself

Returns:

the Component that should receive the focus after aComponent, or null if no suitable Component can be found

Throws:

[IllegalArgumentException](#) - if aContainer is not a focus cycle root of aComponent or a focus traversal policy provider, or if either aContainer or aComponent is null

We note that many legacy APIs use integer values that are restricted to a specific set of constants. For example, when building a color profile, each color component is represented by a specific integer, and the result when using a different integer value is unclear. In situations where a caller would reasonably expect to have to provide a specific value, we do not consider the description to be a directive because the caller is likely to consult the documentation to understand what values are legal. In newer APIs this should be even more straightforward as JAVA finally introduced an enumeration type.

getGamma

```
public float getGamma(int component)
```

Returns a gamma value representing the tone reproduction curve (TRC) for a particular component. The component parameter must be one of REDCOMPONENT, GREENCOMPONENT, or BLUECOMPONENT.

If the profile represents the TRC for the corresponding component as a table rather than a single gamma value, an exception is thrown. In this case the actual table can be obtained through the [getTRC\(int\)](#) method. When using a gamma value, the linear component (R, G, or B) is computed as follows:

```
linearComponent = deviceComponentgamma
```

Parameters:

`component` - The `ICC_ProfileRGB` constant that represents the component whose TRC you want to retrieve

4.7.3 Restricting parameter value by type

Object oriented languages differ from one another in whether they allow *variance* when methods are overridden. Variance usually refers to a change in type along a particular hierarchy path. For example, suppose that `B` extends `A` and `Z` extends `Y`. If `A` has a method that takes a parameter of type `Y`, can the overriding version in `B` be defined to take a different parameter type, such as a value that must be an instance of `Z`? In JAVA, return values can be different, but parameters must be identical. To achieve the

desired restriction, some methods use the documentation to demand that passed objects would belong to a specific subtype, rather than to the more general supertype with which the method is declared.

A well known example of that behavior is the `put` operation from the standard `Map` interface. In the overriding version in `TreeMap`, the passed object must implement the `Comparable` interface.

In SWING, the `setDocument` method of `JTextPane` is inherited from `JTextComponent` with a parameter of type `Document`. Its specific documentation, however, demands an instance of the subtype, `JStyledDocument`.

setDocument

```
public void setDocument(Document doc)
```

Associates the editor with a text document. This must be a `StyledDocument`.

Overrides:

[setDocument](#) in class [JTextComponent](#)

Parameters:

`doc` - the document to display/edit

Throws:

[IllegalArgumentException](#) - if `doc` can't be narrowed to a `StyledDocument` which is the required type of model for this text component

See Also:

[JTextComponent.getDocument\(\)](#)

We note that these situations carry the risk of an awareness problem when callers have a reference for the supertype but the actual object belongs to a subtype that adds these parameter restrictions.

4.7.4 Restricting parameter value by origin

In many cases, a parameter value is restricted in terms of the state and “history” or “origin” of the passed object. In other words, one cannot simply pass any instance of the specified type but rather one that has been obtained in a specific way. Often, this implies a protocol of invocations, so the clause can implicitly be considered to be a protocol directive as well.

For instance, the `JAVA SecurityManager` accepts a `context` parameter which must have been obtained with a call to `getSecurityContext`.

checkRead

```
public void checkRead(String file,  
                     Object context)
```

Throws a `SecurityException` if the specified security context is not allowed to read the file specified by the string argument. The context must be a security context returned by a previous call to `getSecurityContext`.

If `context` is an instance of `AccessControlContext` then the `AccessControlContext.checkPermission` method will be invoked with the `FilePermission(file, "read")` permission.

If `context` is not an instance of `AccessControlContext` then a `SecurityException` is thrown.

If you override this method, then you should make a call to `super.checkRead` at the point the overridden method would normally throw an exception.

Parameters:

`file` - the system-dependent filename.

`context` - a system-dependent security context.

Locales passed to `getInputMethodDisplayName` method in the AWT's `InputMethodDescriptor` class can only be obtained by the descriptor's `getAvailableLocales` function.

getInputMethodDisplayName

```
String getInputMethodDisplayName(Locale inputLocale,  
                               Locale displayLanguage)
```

Returns the user-visible name of the corresponding input method for the given input locale in the language in which the name will be displayed.

The inputLocale parameter specifies the locale for which text is input. This parameter can only take values obtained from this descriptor's [getAvailableLocales\(\)](#) method or null. If it is null, an input locale independent name for the input method should be returned.

If a name for the desired display language is not available, the method may fall back to some other language.

Parameters:

inputLocale - the locale for which text input is supported, or null
displayLanguage - the language in which the name will be displayed

4.7.5 Restricting parameter value by receiving object state

Some parameters are restricted by the state of the object on which the method is invoked.

For example, when setting a cursor name on an SQL Statement, the name must be unique so that it must not conflict with previous names set on that statement object.

setCursorName

```
void setCursorName(String name)  
                 throws SQLException
```

Sets the SQL cursor name to the given string, which will be used by subsequent statement object execute methods. This name can then be used in SQL positioned update or delete statements to identify the current row in the ResultSet object generated by this statement. If the database does not support positioned update/delete, this method is a noop. To insure that a cursor has the proper isolation level to support updates, the cursor's SELECT statement should have the form SELECT FOR UPDATE. If FOR UPDATE is not present, positioned updates may fail.

Note: By definition, the execution of positioned updates and deletes must be done by a different statement object than the one that generated the ResultSet object being used for positioning. Also, cursor names must be unique within a connection.

Parameters:

--- name - the new cursor name, which must be unique within a connection

When working with the AWT CompositeContext and invoking compose, the passed destinations must be “compatible” with the color model which was used when initially constructing the context.

compose

```
void compose(Raster src,  
            Raster dstIn,  
            WritableRaster dstOut)
```

Composes the two source Raster objects and places the result in the destination WritableRaster. Note that the destination can be the same object as either the first or second source. Note that dstIn and dstOut must be compatible with the dstColorModel passed to the [createContext](#) method of the Composite interface.

Parameters:

src - the first source for the compositing operation
dstIn - the second source for the compositing operation
dstOut - the WritableRaster into which the result of the operation is stored

4.7.6 Warning of unexpected behaviors

Certain parameters can be misinterpreted based on the signature, or the signature is simply not specific enough. In those cases, the documentation can warn callers against making a mistake.

For example, in the SWING JTable class, callers to getValueAt, setValueAt, and several other methods are instructed that the column be specified in the table view's display order and not in the TableModel's column order, as the difference can be substantial.

getValueAt

```
public Object getValueAt(int row,  
                        int column)
```

Returns the cell value at `row` and `column`.

Note: The column is specified in the table view's display order, and not in the `TableModel`'s column order. This is an important distinction because as the user rearranges the columns in the table, the column at a given index in the view will change. Meanwhile the user's actions never affect the model's column ordering.

Parameters:

`row` - the row whose value is to be queried
`column` - the column whose value is to be queried

4.7.7 Mutability

Finally, a common concern of many documentation clauses involves the mutability of passed complex objects. Recall that while all parameters in Java are copied, the copy is shallow, so that it is still possible to change the state of a referenced object.

Some methods will explicitly warn callers that the passed object is saved as-is, so that outside changes can affect the containing object.

ProcessBuilder

```
public ProcessBuilder(List<String> command)
```

Constructs a process builder with the specified operating system program and arguments. This constructor does *not* make a copy of the `command` list. Subsequent updates to the list will be reflected in the state of the process builder. It is not checked whether `command` corresponds to a valid operating system command.

In some cases, callers are assured that a copy is made, though this indicates that outside changes are not a valid way of changing the state of the receiver.

addLayoutComponent

```
public void addLayoutComponent(Component comp,  
                               Object constraints)
```

Adds the specified component to the layout, using the specified `constraints` object. Note that constraints are mutable and are, therefore, cloned when cached.

Specified by:

[addLayoutComponent](#) in interface [LayoutManager2](#)

Parameters:

`comp` - the component to be added
`constraints` - an object that determines how the component is added to the layout

Throws:

[IllegalArgumentException](#) - if `constraints` is not a `GridBagConstraint`

Finally, here is an example where the passed object should not be modifiable, although the reason is not clear.

InputMethodHighlight

```
public InputMethodHighlight(boolean selected,
                             int state,
                             int variation,
                             Map<TextAttribute,?> style)
```

Constructs an input method highlight record. The style attributes map provided must be unmodifiable.

Parameters:

`selected` - whether the text range is selected
`state` - the conversion state for the text range - RAW_TEXT or CONVERTED_TEXT
`variation` - the variation for the text range
`style` - the rendering style attributes for the text range, or null

4.8 Imperative directives - Return values

Just as parameter directives convey knowledge that is more pressing than what is typically conveyed with the standard `param` tag, we use the term “return-directives” for important clauses about return values that go beyond the standard description in the `return` tag.

4.8.1 Mutability

Just as certain parameter directives described whether the passed value is cloned before it is kept in the receiver and whether external changes affect the receiver, some return directives convey the same kind of information about the return value. In particular, is the returned value a clone, or is it a direct reference that can be used to change the internal state?

For example, the `command` method in the `ProcessBuilder` class returns a list that callers are free to modify.

`command`

```
public List<String> command()
```

Returns this process builder's operating system program and arguments. The returned list is *not* a copy. Subsequent updates to the list will be reflected in the state of this process builder.

Returns:

This process builder's program and its arguments

Similarly, many operations in the JAVA reflection library explicitly state that they return clones that callers are free to modify.

Many other methods warn callers against modifying the returned collection. This is the case with much of the standard collections library, where methods such as `keys` or `values` are designed for rapid access and do not create a clone.

4.8.2 Cleanup

A related issue to mutability is that of deallocation. If a method returns a reference to an object that has been created or acquired within it, who is responsible for cleaning up the object? The standard garbage collector will eventually recycle unreferenced objects, but there are no guarantee of the timing of this cleanup. Some objects need specific and more pressing cleanup, deinitialization or release, and some are bound to costly system resources.

For example, callers to `createInputMethodJFrame` in the `InputMethodContext` class are instructed to invoke `dispose` on the returned frame when it is no longer needed.

createInputMethodJFrame

```
JFrame createInputMethodJFrame(String title,  
                               boolean attachToInputContext)
```

Creates a top-level Swing JFrame for use by the input method. The intended behavior of this window is:

- it floats above all document windows and dialogs
- it and all components that it contains do not receive the focus
- it has lightweight decorations, such as a reduced drag region without title

However, the actual behavior with respect to these three items is platform dependent.

The title may or may not be displayed, depending on the actual type of window created.

If `attachToInputContext` is true, the new window will share the input context that corresponds to this input method context, so that events for components in the window are automatically dispatched to the input method. Also, when the window is opened using `setVisible(true)`, the input context will prevent deactivate and activate calls to the input method that might otherwise be caused.

Input methods must call [Window.dispose](#) on the returned input method window when it is no longer needed.

4.8.3 Limitations

Some methods return a value which may fall short of what a caller might expect.

For example, the methods for getting an image or an audio clip in a JAVA applet do not actually load the image or the clip. Instead, they return a reference to a lazily-loaded object that would only be filled once the image is displayed or the clip is played. In fact, the corresponding file may not even exist at the time of the call.

getAudioClip

```
public AudioClip getAudioClip(URL url)
```

Returns the `AudioClip` object specified by the URL argument.

This method always returns immediately, whether or not the audio clip exists. When this applet attempts to play the audio clip, the data will be loaded.

Parameters:

`url` - an absolute URL giving the location of the audio clip.

Callers to `getId` on the standard `Thread` class are notified that the number may eventually be recycled.

getId

```
public long getId()
```

Returns the identifier of this `Thread`. The thread ID is a positive `long` number generated when this thread was created. The thread ID is unique and remains unchanged during its lifetime. When a thread is terminated, this thread ID may be reused.

When calling the static `getWindows` method in the AWT `Window` class, callers are warned that the returned list may include some system-generated windows, rather than only user created ones.

getWindows

```
public static Window[] getWindows()
```

Returns an array of all `Window`s, both owned and ownerless, created by this application. If called from an applet, the array includes only the `Window`s accessible by that applet.

Warning: this method may return system created windows, such as a print dialog. Applications should not assume the existence of these dialogs, nor should an application assume anything about these dialogs such as component positions, `LayoutManagers` or serialization.

When obtaining the list of files in a particular directory represented by a `File`, there are no guarantees about order.

list

```
public String[] list()
```

Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

If this abstract pathname does not denote a directory, then this method returns `null`. Otherwise an array of strings is returned, one for each file or directory in the directory. Names denoting the directory itself and the directory's parent directory are not included in the result. Each string is a file name rather than a complete path.

There is no guarantee that the name strings in the resulting array will appear in any specific order; they are not, in particular, guaranteed to appear in alphabetical order.

Returns:

An array of strings naming the files and directories in the directory denoted by this abstract pathname. The array will be empty if the directory is empty. Returns `null` if this abstract pathname does not denote a directory, or if an I/O error occurs.

Some methods indicate that the return value may not be valid. We have already seen examples of this with methods intended for debugging purposes only. These can also result from threading issues, as is the case for `getLocation` in the `AWT Component` class.

getLocation

```
public Point getLocation()
```

Gets the location of this component in the form of a point specifying the component's top-left corner. The location will be relative to the parent's coordinate space.

Due to the asynchronous nature of native event handling, this method can return outdated values (for instance, after several calls of `setLocation()` in rapid succession). For this reason, the recommended method of obtaining a component's position is within `java.awt.event.ComponentListener.componentMoved()`, which is called after the operating system has finished moving the component.

Similarly, the capacity reported by `remainingCapacity` in an `ArrayBlockingQueue` cannot be used to determine if an insertion will be successful.

remainingCapacity

```
public int remainingCapacity()
```

Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking. This is always equal to the initial capacity of this queue less the current size of this queue.

Note that you *cannot* always tell if an attempt to insert an element will succeed by inspecting `remainingCapacity` because it may be the case that another thread is about to insert or remove an element.

Other examples, however, refer to specific states. For example, in the `AWT KeyEvent` class, the `getKeyChar` method only returns a meaningful value for `key_typed` events. There are many similar examples in the `AWT`.

getKeyChar

```
public char getKeyChar()
```

Returns the character associated with the key in this event. For example, the `KEY_TYPED` event for shift + "a" returns the value for "A".

`KEY_PRESSED` and `KEY_RELEASED` events are not intended for reporting of character input. Therefore, the values returned by this method are guaranteed to be meaningful only for `KEY_TYPED` events.

4.8.4 Typing issues

As with parameters, where the type of the passed object may be more restricted than the declared types, a return type may also not be sufficiently specific.

For example, In an `AWT InputContext`, callers to `getInputMethodControlObjec` are instructed to check what kind of object was actually returned.

getInputMethodControlObject

```
public Object getInputMethodControlObject()
```

Returns a control object from the current input method, or null. A control object provides methods that control the behavior of the input method or obtain information from the input method. The type of the object is an input method specific class. Clients have to compare the result against known input method control object classes and cast to the appropriate class to invoke the methods provided.

If no input methods are available or the current input method does not provide an input method control object, then null is returned.

Also, many specific types of collections use documentation to indicate to callers of certain methods that the returned objects do not support certain operations. This is the case, for example, with the `keySet` operation on a standard `Map`.

keySet

```
Set<K> keySet()
```

Returns a [Set](#) view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. If the map is modified while an iteration over the set is in progress (except through the iterator's own `remove` operation), the results of the iteration are undefined. The set supports element removal, which removes the corresponding mapping from the map, via the `Iterator.remove`, `Set.remove`, `removeAll`, `retainAll`, and `clear` operations. It does not support the `add` or `addAll` operations.

4.9 Information directives - Side effects

Our discussion so far has focused on *imperative directives*, which gave callers explicit instructions on actions that must be carried out to ensure correctness; violating these instructions will almost always result in some kind of error. From here on our focus is on *informative directives*, which merely provide callers with important information that they should take into account although they can choose to ignore it. Since inaction does not necessarily lead to an error and as the information may not imply a clear reaction that should take place, the formal specification and automatic verification of such directives would be particularly difficult.

We begin our presentation of informative directives with side effect directives.

The “single responsibility principle” [63] states that each method should be designed to accomplish a single goal. This goal should be obvious from the method’s name, or at least from its signature. If that fails, the goal should be described by the first summary line in the *JavaDoc* description. Nevertheless, for various reasons, we encountered many calls that result in additional *side effects*. These effects, by definition, are different from the method’s primary purpose, and therefore not conveyed by the mediums described above. When unanticipated, these effects can cause software errors and be difficult to trace back to the method that has caused them. We survey types of side effects below.

4.9.1 Lazy creation

A fairly benign and often expected type of side effect is the lazy creation of objects. Many inspection and “getter” operations lazily create an instance of the requested object if one does not yet exist. Since the primary effect is obtaining an existing object, the just-in-time creation can be considered a side effect, especially as the state of the container and its memory footprint changes. For example, calling `getAccessibleContext` on an `Applet` creates and stores an instance of the context. The same occurs when obtaining the accessible context of an `AWT Dialog`.

getAccessibleContext

```
public AccessibleContext getAccessibleContext()
```

Gets the `AccessibleContext` associated with this Applet. For applets, the `AccessibleContext` takes the form of an `AccessibleApplet`. A new `AccessibleApplet` instance is created if necessary.

Object creation may also be a side effect of more complex operations. For example, inserting a character to an AWT `StyledParagraph` may create a new paragraph.

In some cases, a call to a create or obtain an object not only creates the object but also destroys an old version. For example, in an AWT `Canvas` or `Window`, calling `createBufferStrategy` will discard the previous buffer strategy.

createBufferStrategy

```
public void createBufferStrategy(int numBuffers,  
                                BufferCapabilities caps)  
    throws AWTException
```

Creates a new strategy for multi-buffering on this component with the required buffer capabilities. This is useful, for example, if only accelerated memory or page flipping is desired (as specified by the buffer capabilities).

Each time this method is called, the existing buffer strategy for this component is discarded.

4.9.2 Cascading effects

Many object models, such as those used in GUI toolkits, are designed so that certain objects “dominate” or affect others. In some cases, calls to methods on the dominating objects can have an effect on the associated methods. A particularly common pattern is the firing of events or the notification of specific listeners, which may in turn trigger a cascade of method calls. Awareness of these side effects is important because even triggering schemes are inherently complex and difficult to understand from the static structure of a program.

For example, calling `flush` on a `PipedOutputStream` will notify readers that they can begin reading materials.

flush

```
public void flush()  
    throws IOException
```

Flushes this output stream and forces any buffered output bytes to be written out. This will notify any readers that bytes are waiting in the pipe.

Similarly, closing a `RandomAccessFile` closes the associated channel.

close

```
public void close()  
    throws IOException
```

Closes this random access file stream and releases any system resources associated with the stream. A closed random access file cannot perform input or output operations and cannot be reopened.

If this file has an associated channel then the channel is closed as well.

4.9.3 Affecting multiple parts of the state

Each function that changes the state of an object should in general be responsible for a single cohesive and well-defined change. However, some methods intentionally affect additional parts of the state of the same object, often in an attempt to keep the object in a consistent form.

For example, calling `setContentAreaFilled` on a SWING `AbstractButton` may cause the opacity property of the widget to change.

setContentAreaFilled

```
public void setContentAreaFilled(boolean b)
```

Sets the `contentAreaFilled` property. If `true` the button will paint the content area. If you wish to have a transparent button, such as an icon only button, for example, then you should set this to `false`. Do not call `setOpaque(false)`. The default value for the `contentAreaFilled` property is `true`.

This function may cause the component's `opaque` property to change.

The exact behavior of calling this function varies on a component-by-component and L&F-by-L&F basis.

4.9.4 Destruction of existing state

Invoking certain methods `destroys` existing parts of the state *beyond* the part overwritten by the primary goal.

For example, disposing of the last AWT `Window` may cause the JVM to terminate.

dispose

```
public void dispose()
```

Releases all of the native screen resources used by this `Window`, its subcomponents, and all of its owned children. That is, the resources for these components will be destroyed, any memory they consume will be returned to the OS, and they will be marked as undisplayable.

The `Window` and its subcomponents can be made displayable again by rebuilding the native resources with a subsequent call to `pack` or `show`. The states of the recreated `Window` and its subcomponents will be identical to the states of these objects at the point where the `Window` was disposed (not accounting for additional modifications between those actions).

Note: When the last displayable window within the Java virtual machine (VM) is disposed of, the VM may terminate. See [AWT Threading Issues](#) for more information.

A less drastic example is setting the row sorter in a SWING `JTable`, which clears the current selection and resets variable row heights. This is not necessarily intuitive, since a toolkit implementation could conceivably maintain selections when row order changes.

setRowSorter


```
public void setRowSorter(RowSorter<? extends TableModel> sorter)
```

Sets the `RowSorter`. `RowSorter` is used to provide sorting and filtering to a `JTable`.

This method clears the selection and resets any variable row heights.

If the underlying model of the `RowSorter` differs from that of this `JTable` undefined behavior will result.

Unexpected state changes may carry serious consequences. For example, in *Eclipse*, the documentation of the `getVisibleRegion` method in `ITextViewer` states that this seemingly straightforward inspection method may actually change the state of the editor's visible regions. In the actual implementation for JAVA editors, folding is permanently disabled, which appears as a bug to editor users.

 **IRegion org.eclipse.jface.text.ITextViewer.getVisibleRegion()**

Returns the current visible region of this viewer's document. The result may differ from the argument passed to `setVisibleRegion` if the document has been modified since then. The visible region is supposed to be a consecutive region in viewer's input document and every character inside that region is supposed to be visible in the viewer's widget.

Viewers implementing [ITextViewerExtension5](#) may be forced to change the fractions of the input document that are shown, in order to fulfill this contract.

Returns:
this viewer's current visible region

4.9.5 Delayed side effects

Finally, we note that some methods have a “delayed effect” on subsequent calls to other methods of the same object. For example, the documentation for `getBinaryStream` on an `SQL ResultSet` it indicates that the next call to a getter method will close the stream.

`getBinaryStream`

```
InputStream getBinaryStream(String columnLabel)
                throws SQLException
```

Retrieves the value of the designated column in the current row of this `ResultSet` object as a stream of uninterpreted bytes. The value can then be read in chunks from the stream. This method is particularly suitable for retrieving large `LONGVARBINARY` values.

Note: All the data in the returned stream must be read prior to getting the value of any other column. The next call to a getter method implicitly closes the stream. Also, a stream may return 0 when the method `available` is called whether there is data available or not.

4.10 Information directives - Limitations

The name, signature, and summary documentation line of a method are intended to provide potential callers with a valid understanding of the purpose and effect of the call. Since developers often use APIs as means to an end, there is a risk that they may be “optimistic” in their expectations of the abilities of the method and its fit for their specific needs. In some cases, method authors may anticipate potential misunderstandings and use documentation clauses to attempt and prevent them. We term these clauses *limitation directives*, and note that if callers are not aware of them, the program may not operate as expected.

4.10.1 Null effect

The most severe limitation of a method is, perhaps, that it has no effect under specific conditions. This is different from a restriction or a protocol directive because the call is still legal. However, callers not aware of these limitations may be confused as to the lack of desired effect.

For example, the `AppletContext` class provides a `showDocument` method to display a web page. However, while applets are often embedded in a browser, they may also be embedded in other contexts, in which case this operation may have no effect.

`showDocument`

```
void showDocument(URL url)
```

Requests that the browser or applet viewer show the Web page indicated by the `url` argument. The browser or applet viewer determines which window or frame to display the Web page. This method may be ignored by applet contexts that are not browsers.

Similarly, the `SWING JTable` class offers a `print` operation that displays a standard printing dialog and can then send the table to the chosen printer. In “headless mode”, however, the dialog is skipped and the table is printed straight to the default printer, which may not be the intended effect.

`print`

```
public boolean print()
                throws PrinterException
```

A convenience method that displays a printing dialog, and then prints this `JTable` in mode `PrintMode.FIT_WIDTH`, with no header or footer text. A modal progress dialog, with an abort option, will be shown for the duration of printing.

Note: In headless mode, no dialogs are shown and printing occurs on the default printer.

Some methods qualify the conditions under which the effect will not be null, as is the case with focus requests in an `AWT Component`.

requestFocus

```
public void requestFocus()
```

Requests that this Component get the input focus, and that this Component's top-level ancestor become the focused Window. This component must be displayable, focusable, visible and all of its ancestors (with the exception of the top-level Window) must be visible for the request to be granted. Every effort will be made to honor the request; however, in some cases it may be impossible to do so. Developers must never assume that this Component is the focus owner until this Component receives a FOCUS_GAINED event. If this request is denied because this Component's top-level Window cannot become the focused Window, the request will be remembered and will be granted when the Window is later focused by the user.

This method cannot be used to set the focus owner to no Component at all. Use `KeyboardFocusManager.clearGlobalFocusOwner()` instead.

Because the focus behavior of this method is platform-dependent, developers are strongly encouraged to use `requestFocusInWindow` when possible.

Note: Not all focus transfers result from invoking this method. As such, a component may receive focus without this or any of the other `requestFocus` methods of `Component` being invoked.

4.10.2 Accuracy limitations

In the earlier discussion of return-value directives, we saw examples of limitations and shortcomings in the values returned by certain methods. Since these values were potentially quite different from expectations, they could also have been considered to be limitation directives. Additional examples are presented here.

A well known problem in JAVA is the lack of availability of fine-grained timing primitives for benchmarking and similar purposes. A developer can ask for the current time in milliseconds, but that value is only updated once in a while by the operating system. Instead, developers can ask `System` for the current time in nanoseconds, but the result is thread- and processor- specific. The values can therefore only be used for comparisons, and this complex limitation is described in the documentation. If results are taken at face value, serious calculation errors may occur.

nanoTime

```
public static long nanoTime()
```

Returns the current value of the most precise available system timer, in nanoseconds.

This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed but arbitrary time (perhaps in the future, so values may be negative). This method provides nanosecond precision, but not necessarily nanosecond accuracy. No guarantees are made about how frequently values change. Differences in successive calls that span greater than approximately 292 years (2^{63} nanoseconds) will not accurately compute elapsed time due to numerical overflow.

When objects are cloned, there is generally an expectation that the cloned object is identical to the original. Some collections, however, choose to perform only a shallow clone for performance reasons, as is the case with `HashTable`. Confusion about this can result in data corruption.

clone

```
public Object clone()
```

Creates a shallow copy of this hashtable. All the structure of the hashtable itself is copied, but the keys and values are not cloned. This is a relatively expensive operation.

Asking an `SQL ResultSet` for its warnings will only chain those caused by its own methods; the warnings caused by the `Statement` object that was used to obtain this result set are not included.

getWarnings

```
SQLWarning getWarnings() throws SQLException
```

Retrieves the first warning reported by calls on this `ResultSet` object. Subsequent warnings on this `ResultSet` object will be chained to the `SQLWarning` object that this method returns.

The warning chain is automatically cleared each time a new row is read. This method may not be called on a `ResultSet` object that has been closed; doing so will cause an `SQLException` to be thrown.

Note: This warning chain only covers warnings caused by `ResultSet` methods. Any warning caused by `Statement` methods (such as reading OUT parameters) will be chained on the `Statement` object.

Most GUI toolkits provide methods for bound calculations, and these are often limited in their accu-

racy. Below are two examples from `GlyphVector` and `TextLayout` in the AWT.

getVisualBounds

```
public abstract Rectangle2D getVisualBounds()
```

Returns the visual bounds of this `GlyphVector`. The visual bounds is the bounding box of the outline of this `GlyphVector`. Because of rasterization and alignment of pixels, it is possible that this box does not enclose all pixels affected by rendering this `GlyphVector`.

getBounds

```
public Rectangle2D getBounds()
```

Returns the bounds of this `TextLayout`. The bounds are in standard coordinates.

Due to rasterization effects, this bounds might not enclose all of the pixels rendered by the `TextLayout`.

It might not coincide exactly with the ascent, descent, origin or advance of the `TextLayout`.

4.10.3 Implementation limitations

Some limitations are not intentionally designed by the method author but rather originate from limitations in the implementation, often outside the author's control. These limitations are particularly risky because they may only appear in specific cases or in specific platforms, and may therefore be difficult to anticipate or reproduce.

For example, when asking an `Exception` object for the stack trace, callers expect to receive the complete breakdown so that they can pinpoint errors. In some conditions, however, this listing may not be complete.

getStackTrace

```
public StackTraceElement[] getStackTrace()
```

Provides programmatic access to the stack trace information printed by `printStackTrace()`. Returns an array of stack trace elements, each representing one stack frame. The zeroth element of the array (assuming the array's length is non-zero) represents the top of the stack, which is the last method invocation in the sequence. Typically, this is the point at which this throwable was created and thrown. The last element of the array (assuming the array's length is non-zero) represents the bottom of the stack, which is the first method invocation in the sequence.

Some virtual machines may, under some circumstances, omit one or more stack frames from the stack trace. In the extreme case, a virtual machine that has no stack trace information concerning this throwable is permitted to return a zero-length array from this method. Generally speaking, the array returned by this method will contain one

Similarly, asking an `SQL databaseMetaData` object for the list of tables in the database, may not return some tables under some database implementations.

getTables

```
ResultSet getTables(String catalog,  
                    String schemaPattern,  
                    String tableNamePattern,  
                    String[] types)  
throws SQLException
```

Retrieves a description of the tables available in the given catalog. Only table descriptions matching the catalog, schema, table name and type criteria are returned. They are ordered by `TABLE_TYPE`, `TABLE_CAT`, `TABLE_SCHEM` and `TABLE_NAME`.

Note: Some databases may not return information for all tables.

Graphic toolkits face similar challenges. For example, AWT offers a `SystemTray` class, but the actual tray may not be available in some operating systems.

createContext

```
PaintContext createContext(ColorModel cm,  
                           Rectangle deviceBounds,  
                           Rectangle2D userBounds,  
                           AffineTransform xform,  
                           RenderingHints hints)
```

Creates and returns a [PaintContext](#) used to generate the color pattern. Since the `ColorModel` argument to `createContext` is only a hint, implementations of `Paint` should accept a null argument for `ColorModel`. Note that if the application does not prefer a specific `ColorModel`, the null `ColorModel` argument will give the `Paint` implementation full leeway in using the most efficient `ColorModel` it prefers for its raster processing.

Since the API documentation was not specific about this in releases before 1.4, there may be implementations of `Paint` that do not accept a null `ColorModel` argument. If a developer is writing code which passes a null `ColorModel` argument to the `createContext` method of `Paint` objects from arbitrary sources it would be wise to code defensively by manufacturing a non-null `ColorModel` for those objects which throw a `NullPointerException`.

getSystemTray

```
public static SystemTray getSystemTray()
```

Gets the `SystemTray` instance that represents the desktop's tray area. This always returns the same instance per application. On some platforms the system tray may not be supported. You may use the [isSupported\(\)](#) method to check if the system tray is supported.

If a `SecurityManager` is installed, the AWTPermission `accessSystemTray` must be granted in order to get the `SystemTray` instance. Otherwise this method will throw a `SecurityException`.

Changing the ordering of windows in Swing is also affected by platform issues.

toBack

```
public void toBack()
```

If this `Window` is visible, sends this `Window` to the back and may cause it to lose focus or activation if it is the focused or active `Window`.

Places this `Window` at the bottom of the stacking order and shows it behind any other `Windows` in this VM. No action will take place if this `Window` is not visible. Some platforms do not allow `Windows` which are owned by other `Windows` to appear below their owners. Every attempt will be made to move this `Window` as low as possible in the stacking order; however, developers should not assume that this method will move this `Window` below all other windows in every situation.

Because of variations in native windowing systems, no guarantees about changes to the focused and active `Windows` can be made. Developers must never assume that this `Window` is no longer the focused or active `Window` until this `Window` receives a `WINDOW_LOST_FOCUS` or `WINDOW_DEACTIVATED` event. On platforms where the top-most window is the focused window, this method will **probably** cause this `Window` to lose focus. In that case, the next highest, focusable `Window` in this VM will receive focus. On platforms where the stacking order does not typically affect the focused window, this method will **probably** leave the focused and active `Windows` unchanged.

4.10.4 Limited domain

Limitations can apply not only to the value returned by a method, but also to the input values that it can operate on. Certain parameter directives can also be considered limitation directives when the method can only operate on a subset of the inputs which it can receive.

For example, the `canDisplay` method on the AWT `Font` object indicates that it cannot handle certain types of characters.

canDisplay

```
public boolean canDisplay(char c)
```

Checks if this `Font` has a glyph for the specified character.

Note: This method cannot handle [supplementary characters](#). To support all Unicode characters, including supplementary characters, use the [canDisplay\(int\)](#) method or `canDisplayUpTo` methods.

4.10.5 No event triggering

When we discussed side-effect directives, we saw examples of methods that trigger a cascade of events and notifications. This functionality is crucial in many GUI toolkits. Some methods, however, indicate the opposite: they can be used to change specific parts of the state, without triggering the notifications. If callers do expect them to trigger the cascade, errors will occur.

setState

```
public void setState(boolean state)
```

Sets the state of this check box to the specified state. The boolean value `true` indicates the "on" state, and `false` indicates the "off" state.

Note that this method should be primarily used to initialize the state of the checkbox. Programmatically setting the state of the checkbox will *not* trigger an `ItemEvent`. The only way to trigger an `ItemEvent` is by user interaction.

select

```
public void select(int index)
```

Selects the item at the specified index in the scrolling list.

Note that passing out of range parameters is invalid, and will result in unspecified behavior.

Note that this method should be primarily used to initially select an item in this component. Programmatically calling this method will *not* trigger an `ItemEvent`. The only way to trigger an `ItemEvent` is by user interaction.

An interesting example of such limitations will be presented in our later lab study. In a SWING `JLayeredPane`, calling `putLayer` will only associate an element with a layer but will not actually trigger the expected redraw. Another example from that study is concerned with adding or removing items from SWING containers, as there is no visual effect until `validate` is called.

4.10.6 Cannot support particular goals

We have previously seen method documentations that warn callers against using them for certain purposes. In some cases these warnings were stated as restrictions or as alternatives, but in some cases there is simply a warning that the method is limited in its ability to support this goal.

For example, the `isClosed` method on an SQL `Connection` seems like a logical choice for checking whether the connection is valid, as it would only be open if it is. However, the method warns against it and indicates that validity can only be checked while operations are executed.

isClosed

```
boolean isClosed()  
throws SQLException
```

Retrieves whether this `Connection` object has been closed. A connection is closed if the method `close` has been called on it or if certain fatal errors have occurred. This method is guaranteed to return `true` only when it is called after the method `Connection.close` has been called.

This method generally cannot be called to determine whether a connection to a database is valid or invalid. A typical client can determine that a connection is invalid by catching any exceptions that might be thrown when an operation is attempted.

Similarly, a common issue in UI toolkits is determining whether a certain component is actually visible, as this may have usability and performance implications. The `isShowing` method in an AWT `Component` looks like it should provide answers to these queries, but its documentation indicates that in some cases it may be impossible to calculate this correctly.

isShowing

```
public boolean isShowing()
```

Determines whether this component is showing on screen. This means that the component must be visible, and it must be in a container that is visible and showing.

Note: sometimes there is no way to detect whether the `Component` is actually visible to the user. This can happen when:

- the component has been added to a visible `ScrollPane` but the `Component` is not currently in the scroll pane's view port.
- the `Component` is obscured by another `Component` or `Container`.

A common strategy for comparing two strings in a case insensitive manner or use them as mapping keys is to change them to all-lowercase or all-uppercase. The standard implementation in a Java string, however, warns that it depends on the locale, so the resulting values aren't globally consistent.

toLowerCase

```
public String toLowerCase()
```

Converts all of the characters in this `String` to lower case using the rules of the default locale. This is equivalent to calling `toLowerCase(Locale.getDefault())`.

Note: This method is locale sensitive, and may produce unexpected results if used for strings that are intended to be interpreted locale independently. Examples are programming language identifiers, protocol keys, and HTML tags. For instance, `"TITLE".toLowerCase()` in a Turkish locale returns `"t?tle"`, where '?' is the LATIN SMALL LETTER DOTLESS I character. To obtain correct results for locale insensitive strings, use `toLowerCase(Locale.ENGLISH)`.

4.11 Informative directives - Performance

API methods generally offer callers means to accomplish a particular goal while encapsulating details of the implementation. When using high-quality APIs, clients can reasonably expect that the most optimal implementations and algorithms are used. Nevertheless, clients are expected to have some understanding of performance issues and the possibilities of implementation. For example, they need to be able to choose the appropriate data structure or meet the preconditions for a particular search algorithm.

While implementation typically meets expectations, there are situations where the method does not perform as optimally as it possibly could, or where the performance limitation is not straightforward. In addition, the platform, hardware, or network may add additional delays. To make callers aware of these issues, some methods state performance directives. Ignoring them is not necessarily an error, but could be risky in high-performance systems.

4.11.1 Inherently expensive operations

Some operations are inherently expensive by definition, but the domain may not be familiar enough to potential callers.

For example, font transformation is quite expensive, so the method `setGlyphTransform` on an `AWT GlyphVector` adds a warning that setting a transformation on the vector (which can represent a line of text) will carry a performance hit.

setGlyphTransform

```
public abstract void setGlyphTransform(int glyphIndex,  
                                     AffineTransform newTX)
```

Sets the transform of the specified glyph within this `GlyphVector`. The transform is relative to the glyph position. A `null` argument for `newTX` indicates that no special transform is applied for the specified glyph. This method can be used to rotate, mirror, translate and scale the glyph. Adding a transform can result in significant performance changes.

Algorithms for detecting deadlocks are inherently expensive and so are dynamic operations for identifying them. Callers to `findMonitorDeadlockedThreads` in `ThreadMXBean` are warned that it should be used only for troubleshooting rather than for regular synchronization control.

findDeadlockedThreads

```
long[] findDeadlockedThreads()
```

Finds cycles of threads that are in deadlock waiting to acquire object monitors or [ownable synchronizers](#). Threads are *deadlocked* in a cycle waiting for a lock of these two types if each thread owns one lock while trying to acquire another lock already held by another thread in the cycle.

This method is designed for troubleshooting use, but not for synchronization control. It might be an expensive operation.

Asking a `ResultSet` whether the current row is the last one may be expensive since the driver may need to fetch ahead an extra row.

isLast

```
boolean isLast()  
    throws SQLException
```

Retrieves whether the cursor is on the last row of this `ResultSet` object. **Note:** Calling the method `isLast` may be expensive because the JDBC driver might need to fetch ahead one row in order to determine whether the current row is the last row in the result set.

Note: Support for the `isLast` method is optional for `ResultSet`s with a result set type of `TYPE_FORWARD_ONLY`

Similarly, various performance issues also plague calls to `refreshRow` on the `ResultSet`

refreshRow

```
void refreshRow()  
    throws SQLException
```

Refreshes the current row with its most recent value in the database. This method cannot be called when the cursor is on the insert row.

The `refreshRow` method provides a way for an application to explicitly tell the JDBC driver to refetch a row(s) from the database. An application may want to call `refreshRow` when caching or prefetching is being done by the JDBC driver to fetch the latest value of a row from the database. The JDBC driver may actually refresh multiple rows at once if the fetch size is greater than one.

All values are refetched subject to the transaction isolation level and cursor sensitivity. If `refreshRow` is called after calling an updater method, but before calling the method `updateRow`, then the updates made to the row are lost. Calling the method `refreshRow` frequently will likely slow performance.

In many mapping implementations, checking whether a key is mapped is relatively cheap, but checking whether any key maps to a particular value is expensive. Certain implementations warn callers against these costs, as is the case with `ConcurrentHashMap`.

containsValue

```
public boolean containsValue(Object value)
```

Returns `true` if this map maps one or more keys to the specified value. **Note:** This method requires a full internal traversal of the hash table, and so is much slower than method `containsKey`.

Similarly, while some data structures track the number of elements inside them to be able to report it in constant time, that is not the case with many of the concurrent data structures such as `ConcurrentLinkedListQueue` and others; care is therefore required, especially in loops.

size

```
public int size()
```

Returns the number of elements in this queue. If this queue contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

Beware that, unlike in most collections, this method is *NOT* a constant-time operation. Because of the asynchronous nature of these queues, determining the current number of elements requires an $O(n)$ traversal.

4.11.2 Presenting more efficient alternatives

Some APIs offer “expert level” versions of their standard operations which allow fine tuning for performance.

For example, methods for setting different types of streams on an SQL `CallableStatement` instruct callers to check if their JDBC driver documentation wants them to use a more customizable version of this method to achieve better performance. The same instruction appears on many other operations throughout the SQL API.

setBinaryStream

```
void setBinaryStream(String parameterName,  
                    InputStream x)  
    throws SQLException
```

Sets the designated parameter to the given input stream. When a very large binary value is input to a LONGVARIABLE parameter, it may be more practical to send it via a java.io.InputStream object. The data will be read from the stream as needed until end-of-file is reached.

Note: This stream object can either be a standard Java stream object or your own subclass that implements the standard interface.

Note: Consult your JDBC driver documentation to determine if it might be more efficient to use a version of setBinaryStream which takes a length parameter.

Similarly, callers to createStatement are told to use PreparedStatement objects if the same statement will be executed multiple times.

createStatement

```
Statement createStatement()  
    throws SQLException
```

Creates a Statement object for sending SQL statements to the database. SQL statements without parameters are normally executed using Statement objects. If the same SQL statement is executed many times, it may be more efficient to use a PreparedStatement object.

Result sets created using the returned Statement object will by default be type TYPE_FORWARD_ONLY and have a concurrency level of CONCUR_READ_ONLY. The holdability of the created result sets can be determined by calling getHoldability().

Rather than merely warn callers about performance issues, some method documentations suggest ways to improve it. For example, making changes to containers in a UI toolkit is often expensive because of redraw, and can cause flicker. Many methods of the SWING Container instruct callers to add multiple items and call validate only once they are ready.

```
• void java.awt.Container.add(Component comp, Object constraints)
```

Adds the specified component to the end of this container. Also notifies the layout manager to add the component to this container's layout using the specified constraints object. This is a convenience method for [addImpl](#).

Note: If a component has been added to a container that has been displayed, validate must be called on that container to display the new component. If multiple components are being added, you can improve efficiency by calling validate only once, after all the components have been added.

Parameters:
comp the component to be added
constraints an object expressing layout constraints for this component

Since:
JDK1.1

See Also:
[addImpl](#)
[validate](#)
[javax.swing.JComponent.revalidate\(\)](#)
[LayoutManager](#)

The Eclipse framework offers many additional example. For instance, when using Eclipse content type matching, names should be provided to “avoid querying the entire registry”. Similarly, performance critical code should not invoke ITextFileBufferManager.getTextFileBuffer() since it compares all buffers in the system. When using an Eclipse IFileStore, the fetchInfo method should only be called on a highly available system unless a progress monitor is set up. In the Eclipse Platform class, users should only call getResourceBundle for externalizing strings from the manifest file, as memory performance will degrade for other strings. Clients of PerformanceStats.isEnabled() should cache the result, which is expensive to calculate.

4.12 Information directives - Threading

The final type of informative directives that we discuss here are concerned with threading. When we discussed imperative directives, we have seen directives that require calls to always (or never) come from specific threads. Similarly, there are many methods that require callers to acquire specific locks, and these can be considered protocol directives. These locking comments have been noted in many languages, and recent research has attempted to parse them automatically and match against system behavior [83, 84].

Informative threading directives, however, do not make specific demands of the caller. Instead, they supply information about threading related issues that may affect an unaware caller and potentially result

in errors.

4.12.1 Thread safety

The first issue conveyed by threading directives is the thread safety of the function. Documentation indicates whether the method itself is thread safe (and can be invoked concurrently without additional locking), or whether the returned objects are thread-safe and can be modified without affecting threads that also called the same method.

For example, the path iterator returned by a `Line2D` object in the AWT user interface toolkit is not thread safe.

`getPathIterator`

```
public PathIterator getPathIterator(AffineTransform at)
```

Returns an iteration object that defines the boundary of this `Line2D`. The iterator for this class is not multi-threaded safe, which means that this `Line2D` class does not guarantee that modifications to the geometry of this `Line2D` object do not affect any iterations of that geometry that are already in process.

Surprisingly, the same operation for an `Ellipse2D` object from the same package is thread safe. Clearly, callers need to be warned when calling one in case they are familiar with the other.

`getPathIterator`

```
public PathIterator getPathIterator(AffineTransform at)
```

Returns an iteration object that defines the boundary of this `Ellipse2D`. The iterator for this class is multi-threaded safe, which means that this `Ellipse2D` class guarantees that modifications to the geometry of this `Ellipse2D` object do not affect any iterations of that geometry that are already in process.

In some cases concurrent access results in undefined behavior, as is the case with draining a `BlockingQueue` in the concurrent collections library.

`drainTo`

```
int drainTo(Collection<? super E> c)
```

Removes all available elements from this queue and adds them to the given collection. This operation may be more efficient than repeatedly polling this queue. A failure encountered while attempting to add elements to collection `c` may result in elements being in neither, either or both collections when the associated exception is thrown. Attempts to drain a queue to itself result in `IllegalArgumentException`. Further, the behavior of this operation is undefined if the specified collection is modified while the operation is in progress.

Finally, the threading behavior may be very complex and multiple directives may be potential callers to read the entire documentation.

`getMostRecentEventTime`

```
public static long getMostRecentEventTime()
```

Returns the timestamp of the most recent event that had a timestamp, and that was dispatched from the `EventQueue` associated with the calling thread. If an event with a timestamp is currently being dispatched, its timestamp will be returned. If no events have yet been dispatched, the `EventQueue`'s initialization time will be returned instead. In the current version of the JDK, only `InputEvents`, `ActionEvents`, and `InvocationEvents` have timestamps; however, future versions of the JDK may add timestamps to additional event types. Note that this method should only be invoked from an application's event dispatching thread. If this method is invoked from another thread, the current system time (as reported by `System.currentTimeMillis()`) will be returned instead.

4.12.2 Blocking

Another threading related issue involves the blocking of the invoking thread. Certain methods that depend on outside event are designed so that the thread blocks until a certain condition is met. If the caller is not aware of this behavior, serious errors may result, at least on the first invocation.

For example, when reading an element from a `ByteArrayInputStream` in the standard IO library, the method returns immediately with a value or an indication that the end of the stream has been reached.

read

```
public int read()
```

Reads the next byte of data from this input stream. The value byte is returned as an `int` in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value `-1` is returned.

This `read` method cannot block.

When reading from a `FileInputStream`, on the other hand, the thread will block until the input is available. If the program is single-threaded and the input never materializes, the program will effectively hang.

read

```
public int read()
    throws IOException
```

Reads a byte of data from this input stream. This method blocks if no input is yet available.

The definition of `read` in the parent interface `InputStream` is more general, telling us that the method may block until a variety of conditions are met.

read

```
public abstract int read()
    throws IOException
```

Reads the next byte of data from the input stream. The value byte is returned as an `int` in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value `-1` is returned. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown.

We note that our lab study, described later, shows that unexpected blocking behavior can be quite challenging for users to debug. In our first task, there is a method that blocks until input is available or until another operation is invoked.

Blocking directives are also very common in UI toolkits, which even offer special methods for that purpose as is the case with `SwingUtilities`.

invokeAndWait

```
public static void invokeAndWait(Runnable doRun)
    throws InterruptedException,
    InvocationTargetException
```

Causes `doRun.run()` to be executed synchronously on the AWT event dispatching thread. This call blocks until all pending AWT events have been processed and (then) `doRun.run()` returns. This method should be used when an application thread needs to update the GUI. It shouldn't be called from the `EventDispatchThread`. Here's an example that creates a new application thread that uses `invokeAndWait` to print a string from the event dispatching thread and then, when that's finished, print a string from the application thread.

4.12.3 Performance

Finally, we note that some methods present warnings and issues related to performance in the context of multithreading.

For example, the standard `random` operation recommends that if multiple threads need to generate random number, they may benefit from each having its own local generator.

random

```
public static double random()
```

Returns a `double` value with a positive sign, greater than or equal to 0.0 and less than 1.0. Returned values are chosen pseudorandomly with (approximately) uniform distribution from that range.

When this method is first called, it creates a single new pseudorandom-number generator, exactly as if by the expression

```
new java.util.Random
```

This new pseudorandom-number generator is used thereafter for all calls to this method and is used nowhere else.

This method is properly synchronized to allow correct use by more than one thread. However, if many threads need to generate pseudorandom numbers at a great rate, it may reduce contention for each thread to have its own pseudorandom-number generator.

The standard `ScheduledExecutorService` warns when scheduling tasks at fixed rates about the consequences of using intervals that are too short.

`scheduleAtFixedRate`

```
ScheduledFuture<?> scheduleAtFixedRate(Runnable command,  
                                       long initialDelay,  
                                       long period,  
                                       TimeUnit unit)
```

Creates and executes a periodic action that becomes enabled first after the given initial delay, and subsequently with the given period; that is executions will commence after `initialDelay` then `initialDelay+period`, then `initialDelay + 2 * period`, and so on. If any execution of the task encounters an exception, subsequent executions are suppressed. Otherwise, the task will only terminate via cancellation or termination of the executor. If any execution of this task takes longer than its period, then subsequent executions may start late, but will not concurrently execute.

4.13 Conclusions

In this chapter, I presented guidelines for the recognition of directives and a detailed taxonomy with many examples of directive types. These examples illustrate the breadth of directive types and their prevalence across various facets of the JAVA standard library. While only few of these methods may be previously familiar to the reader, it should be straightforward to understand how a lack of awareness of these directives will cause problems for developers who seek to use these methods.

It is important to note that while the presence of directives could be indicative of design, documentation, or naming issues, in some cases such surprises are not avoidable. Certain APIs, such as SQL, rely so much on state and complex relations between methods, that it is impossible to convey everything via method naming. Such APIs may simply be too difficult to use without a general understanding of their architecture and usage principles, although there still a need for awareness of the directives in each method.

One reason that this chapter focused on a classification of directives is the potential utility in helping users filter or adjust the presentation of specific types. For instance, a developer that is currently focused on correctness issues could be distracted by directives that recommend performance improvements. On the other hand, a developer concerned with threading problems may want increased visibility for calls that have associated threading directives.

Accordingly, our *eMoose* tool is used with sets of directives that are each assigned a major type from the above taxonomy. When examining the *JavaDoc* of a method, the type of each directive precedes its text, allowing readers to skip types that they are less interested in. Users can also filter out specific types.

Currently, the annotation over a call that has an associated directive is the same for all directive types - a simple dashed box. When the necessary support is added in future *Eclipse* versions, the annotation will be adjusted and accompanied by an icon to help distinguish the type of directive. This information should help developers with the decision whether to investigate the call or ignore it.

An opportunity for future research is to investigate whether the directive type can be automatically tied to the user's recent activities to dynamically adjust the priority and presentation of displayed directives.

Chapter 5

The eMoose tool

This chapter presents *eMoose*, an implementation of the approach proposed in this dissertation of “pushing” directives into the awareness of developers who are examining calling code.

The presentation is focused on the functionality of the tool as it appears to API authors and to developers who use these APIs, but includes some relevant implementation details. It is also concerned only with the standard version of the tool, which is available online at <http://emoose.cs.cmu.edu>. The tool was initially developed as a more comprehensive client-server based framework for capturing episodic activity [26]. Inspired by my studies of design, it preserved a complete record of the activities of developers along with the viewports and documents they were exposed to. However, this version was never released and is outside the current scope.

From the point of view of IDE users, *eMoose* has three primary features: First, it highlights calls to methods that have associated directives. Second, where applicable it augments the *JavaDoc* hover with a list of directives for the method. Third, it allows developers to easily add new directives.

eMoose is written in JAVA and is currently aimed only at JAVA developers. We chose to focus on JAVA for several reasons: Most importantly, this language is relatively straightforward to parse and analyze. Each file can be analyzed independently, and it is straightforward to determine the static type of the object on which a method is invoked. While header comments similar to *JavaDocs* are present in languages like C++, the linking model, complex file inclusion rules, and the presence of function pointers, all make it difficult to resolve calls efficiently. Similar comments are also available in dynamic languages like *python* or *Smalltalk* but the lack of compile-time types makes it difficult to resolve calls. Nevertheless, our approach could potentially be implemented for these languages at higher computation costs. Another reason for choosing JAVA is that libraries are often provided as obfuscated archives with external HTML based documentation but no human-readable header files. I believe that the higher cost of investigating documentation makes it even more crucial to bring important directives to the awareness of callers.

eMoose is built as a set of plug-ins for the popular open-source *Eclipse* IDE. This IDE is designed for extension, has a very robust representation of JAVA projects, and also allows developers to add decorations to program text and augment the *JavaDoc* hover. Nevertheless, it should be relatively straightforward to implement our approach in other IDEs, although not necessarily as external plug-ins.

5.1 Knowledge space

The *eMoose* tool revolves around an abstract *knowledge space* which consists of *knowledge items* (KIs). A KI is an atomic and concise element which is intended to be captured rapidly and cost-effectively as a single sentence or text line conveying one idea. In the scope of the standard version, each KI is associated with a method. Most KIs correspond to a single directive in the *JavaDoc* of a method, but some are associated with an embedded to-do comment. Every KI is also assigned a single type from a predefined set that is based on the major types described in Chap. 4.

5.1.1 Creating embedded knowledge items

All KIs are initially created as *embedded KIs*. That is, they are automatically generated to reflect specially tagged lines within comment blocks in the source code loaded within the current *Eclipse* workspace. For *eMoose*, we borrowed a notation proposed by the *TagSEA* tool [78], of using a `@tag` marker to identify specific lines within the documentation. However, while tags in *TagSEA* serve primarily as bookmarks, in *eMoose* they serve to define a `KnowledgeItem`. In our case, a KI is created with the following syntax:

```
@tag usage.TYPE [-rating=0..5] [-author=AUTHOR]: Text until EOL
```

That is, the line starts with `@tag`, followed immediately by `usage.TYPE` where `TYPE` is replaced by a literal from specific set (e.g., `restriction`, `protocol`, `sideeffect`, etc.). It can then be followed by optional arguments such as a numeric rating (default is 3) and an author id string. All this is followed by a colon, followed by the text of the KI until the end of the line is reached.

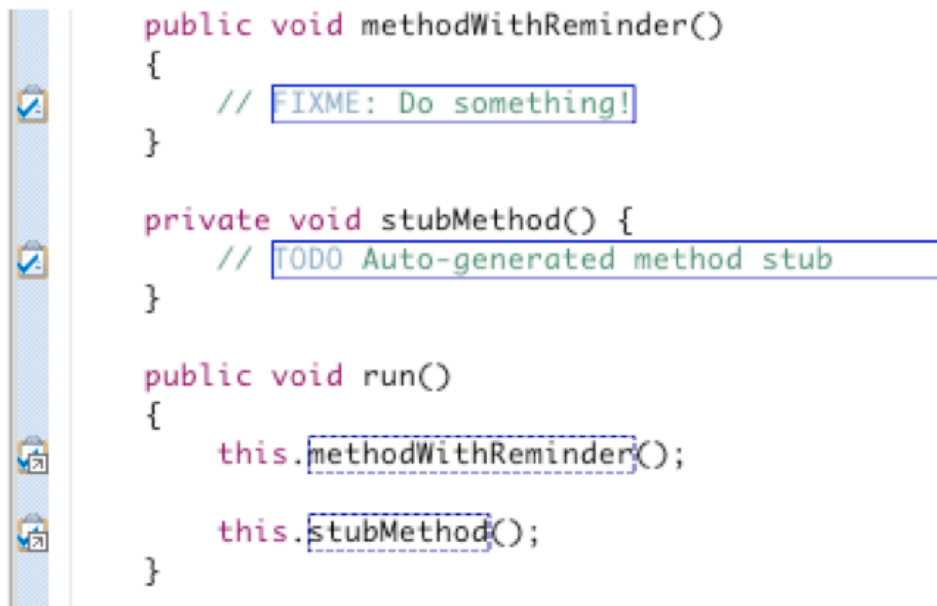
These tagged lines can be embedded in the comment block by anyone using any editor, and are therefore distributed with the source code. If they are embedded within the *JavaDoc* block, they will also become part of the public HTML-based documentation and be visible through the *JavaDoc* hover. An investment in creating the tagged line still offers benefits to non-*eMoose* users who will see explicitly tagged directives when they read the text.

If authors wish to avoid having the tagged line visible within the *JavaDoc*, they can place it within the source code of the method or as a line-comment (`//`) between the *JavaDoc* block and the actual method declaration. In both cases, the line would be transformed into a KI but would not appear inside the *JavaDoc* when it is exported or presented.

In addition to tagged lines with the format described above, *eMoose* also recognizes all variants of to-do comments [79] and creates corresponding KIs, as illustrated in Fig. 5.1. While the *Eclipse* IDE already recognizes such comments, it merely adds them to a global task list that encompasses all files in the project and might be very lengthy. *eMoose* is designed under the premise that the presence of a to-do comment in a function is likely to indicate a limitation in its current ability to meet its contract. If it is invoked without the caller's awareness, the effects may not match expectations and errors may result. By automatically creating KIs and "pushing them", we let callers know that the method they are invoking is incomplete, and possibly also lead them to complete it.

One of the costs associated with function documentation is the need to switch from the current location to the header and back. For example, a developer who realizes a precondition or limitation while writing a complex conditional in the body of a long function may not want to leave the current location. This can be avoided with a key combination that brings up a series of pop-ups, requiring only keyboard input, for the creation of KIs. The new KI is embedded in the *JavaDoc* block while the insertion point remains in its original location in the source code.

In principle, the set of embedded KIs maintained by *eMoose* at any point in time should correspond



```
public void methodWithReminder()
{
    // FIXME: Do something!
}

private void stubMethod() {
    // TODO Auto-generated method stub
}

public void run()
{
    this.methodWithReminder();

    this.stubMethod();
}
```

Figure 5.1: Knowledge items that are automatically created from to-do comments

to the set of all tags in the current source files. For performance reasons, however, we had to somewhat relax this continuous consistency. To avoid flicker and slowdowns, a new tagged line or an edited existing line is not recognized or updated until the file is saved by the user. Once it is recognized, a solid box appears around the line to indicate that it has become a KI, as depicted in Fig. 5.1.

A side benefit of these solid box decorations is that they help readers who are skimming the source code to quickly recognize KIs and to-do comments. By default *Eclipse* adds a small icon on the side for to-dos, but nothing for directives.

5.1.2 Exporting and sharing knowledge items

APIs and their documentation are typically meant to be used by multiple developers. While the sharing of embedded KIs may be practical for in-house code and APIs, it is not practical for third-party APIs. First, available library source code is rarely loaded into the actual *Eclipse* workspace, where it would be monitored for KIs by *eMoose*. Instead, individual files are loaded temporarily only when relevant, for example when interactively debugging into platform code. Second, while the source code may be available for us to add embedded KIs, other users would still have the default distribution without them. Third, many libraries are distributed as archives without the source code, which may be proprietary.

To this end, *eMoose* also supports *virtual KIs*. These have similar properties to embedded KIs, such as type, text, and possible rating. They are also tied to specific methods identifiers, but without being tied to specific locations in the source code. As a result, *eMoose* will be able to decorate calls to these methods even if their source code is not available. However, they cannot be removed by changing the method source code, or if the method signature changes.

When API authors use *eMoose* on the annotated source code of the API, they can export all the KIs into an XML file. This file can be distributed to other users and loaded to create virtual KIs. To further facilitate distribution, the XML files can be embedded in special plug-in files, which can be distributed and regularly updated using *Eclipse* update sites.

This export-import mechanism is effective for distributing sets of directives for APIs that are outside

our control. For example, I obtained and annotated the publicly-available source code of the JDK and of *Eclipse* with lines for directives. I then exported these into XML and created wrapping plug-ins. These plug-ins are now distributed with *eMoose*, allowing users to immediately benefit from the tool for some of their existing code without having to invest in creating any KIs.

A similar benefit of the export-import mechanism is its potential support for distributing community-generated information. Popular APIs have active user communities that generate new knowledge about it, including: best practices and solutions for common problems, limitations and errors in the API and workarounds, and references to additional material. Since communities typically cannot change the API documentation, this knowledge is currently maintained externally on the web and requires an active search (e.g. [20]). Using *eMoose*, community volunteers could potentially obtain the source code, embed KIs corresponding to the community-knowledge, and distribute them as plug-ins.

One problem with this community-generated information is that it would not be visible when the method is examined as it does not appear in the embedded documentation. *eMoose* will signal the availability of such information by creating a solid box decoration over the method name. By hovering over this box and opening the *JavaDoc* hover, users can see the list of associated KIs.

5.1.3 Rating directives

As mentioned above, every KI has a rating that can be specified explicitly, or is 3 by default. The rating represents a (subjective) estimate of the importance of the directive, a single dimension that should encompass associated risk, relevancy, confidence that this is indeed a directive, and other concerns. I arbitrarily decided to use a 5-point scale, where 5 is a very clear and important directive, while 1 is a clause that has little importance or is not clearly a directive. A rating of 0 is perceived to represent a clause that is not a directive at all. The default rating for a directive is 3, exactly at the middle of the scale.

Directive ratings have three purposes: First, they are used to make the calls to methods with highly-rated directives more visible by using a higher-contrast decoration. Second, to determine the order in which directives appear in the *JavaDoc* hover. Third, to allow lower-rated directives to be filtered out.

In the client-server version of *eMoose*, other users (in addition to the author) can rate the directive. These user ratings are averaged with the original rating provided by the author, and the average is rounded to the closest integer for presentation.

5.2 Contextual model

To enable the decoration of calls and the augmentation of the *JavaDoc* hover, *eMoose* continuously maintains a *contextual model*. This model reflects the currently visible source code, the methods invoked by it, and the directives which may be associated with them. Much of the complexity of this model derives from the fact that the same method may be invoked from multiple locations, and that a single call may have multiple potential targets due to polymorphism.

The contextual model consists, in principle, of *relevancy trees* for every source file in the system. For performance reasons, however, only a single tree is built and managed at each point in time, and reflects the current source file. The relevancy tree has several types of nodes: Call nodes represent specific offsets in the source code, allowing decorations to be placed and updated as the code changes. Method nodes represent all potential call targets, so that call nodes can point at them while nodes representing KIs can be associated with them. Finally, type nodes are used internally to represent the relation between types in order to manage polymorphism-related presentation.

To illustrate how the relevancy tree is constructed, suppose that we have a file named `Driver.java` file, which contains a single method named `run`. This method, depicted on the left side of Fig. 5.3, operates on a parameter of type `B`, which is part of the hierarchy on the right side. Some methods in this hierarchy have associated KIs. A version of the relevancy tree that does not include KI nodes is depicted in Fig. 5.3.

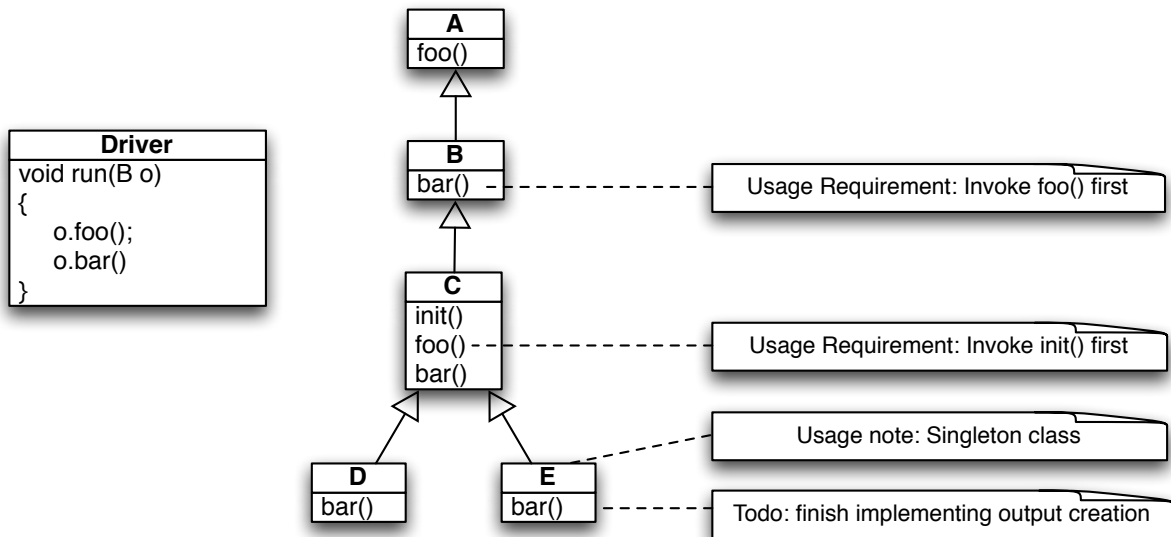


Figure 5.2: Sample class hierarchy

A relevancy tree like the one of Fig. 5.2 is created in the following manner: First, a node is created for the “enclosing class” represented by the source file (e.g., `Driver`) and subnodes are created for each of its “enclosing methods” (e.g., `run`). A subnode is then created for each compile-time static type on which at least one method is invoked within the enclosing method (e.g., `B` for `run`). For each actual call in the enclosing method, a call subnode is created. In this example, we have subnodes for `foo` and `bar`, although if there was a second call to `foo` in `run` there would also be another call node.

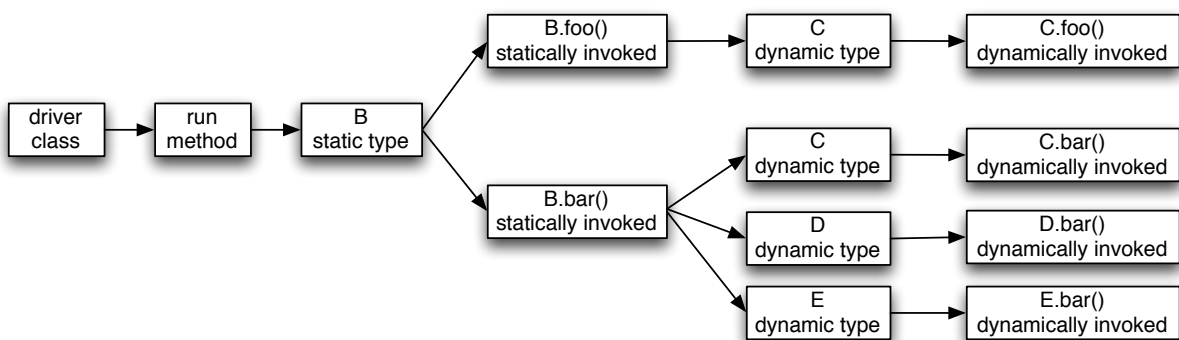


Figure 5.3: Relevancy tree example

If there are any subtypes or supertypes of the static type that explicitly override (or declare) this method, a subnode for each type is created under the call node. In this case, since `bar` is redefined in `C`, `D`, and `E` and all are subtypes of `B`, there are three children to the `B.bar()` node. Each of these dynamic-type nodes has a subnode representing the overridden version of the method.

Finally, although not shown in the figure, every call node and some overriding method nodes have

subnodes representing the relevant KIs associated with that specific version of the method. For example, `B.bar()` will have a subnode for the need to invoke `foo`, `E.bar()` will have a subnode for the to-do reminder, and `C.bar()` and `D.bar()` will not have any.

The current implementation of *eMoose* uses the *Eclipse* built-in facilities to identify all outgoing calls from a particular method, and to identify all possible dynamic types for the compile-time types on which a method is invoked. These calculations are moderately expensive, and require us to use result-caching. However, they are also inaccurate, as they assume that every declared subtype of a particular static type is a potential dynamic type and thus a call target. For instance, in the following code, all subtypes of `List` (and there are dozens of them) are considered potential call targets for `add` even though a smarter analysis may deduce that the dynamic type is a `LinkedList`:

```
List li = new LinkedList(); li.add("Value");
```

A consequence of the construction algorithms described above is that the relevancy tree may contain many duplicates. In other words, if there were two calls to `bar` in `run`, the entire subtree would be replicated for each. This redundancy (rather than the sharing of sub-lattices) allow us to maintain a tree structure.

One reason for doing so is that it minimizes computation (and thus presentation delay) when the user hovers over a particular call. The subtree of the call node corresponds exactly to the structure that will be displayed to users in a tree widget, as we shall later explain. We seek to minimize this delay as increased latency when hovering over a call may reduce the willingness of developers to explore its documentation. The relevancy tree is calculated in the background, and therefore has limited impact on users.

A second, and more critical reason for the redundancy is that by maintaining a separate subtree for every invocation, our implementation is ready to support “smarter” type resolution-algorithms. Suppose, for example, that method `run` had the following source code:

```
if (o instanceof E) { XXXX; o.bar(); YYYYY} else { ZZZZ; o.bar(); WWWW;}
```

While our default implementation considers all subtypes of `B` to be potential targets for calls on `o` in `run`, a smarter algorithm could restrict this set for each branch of the conditional. By maintaining separate subtrees, the subtree for the first call to `bar` can be different than the subtree to the second call.

We note that the set of KIs associated with nodes in the relevancy tree can be a subset of all the potential KIs in our knowledge space. Based on user preferences, KIs with a low rating, those with unwanted types, and those that have previously been explored can be omitted from the tree.

Finally, while the relevancy tree should ideally reflect the current state of the code, we relaxed this requirement for performance reason. While code is being edited, the need to continuously update the relevancy tree and the locations of the decorations would significantly slow down the editor and cause visible flicker. Instead, the decorations may disappear, and only reappear when there is a break in the editing activities. As platforms become faster, this compromise could be disabled for a more fluent user-experience.

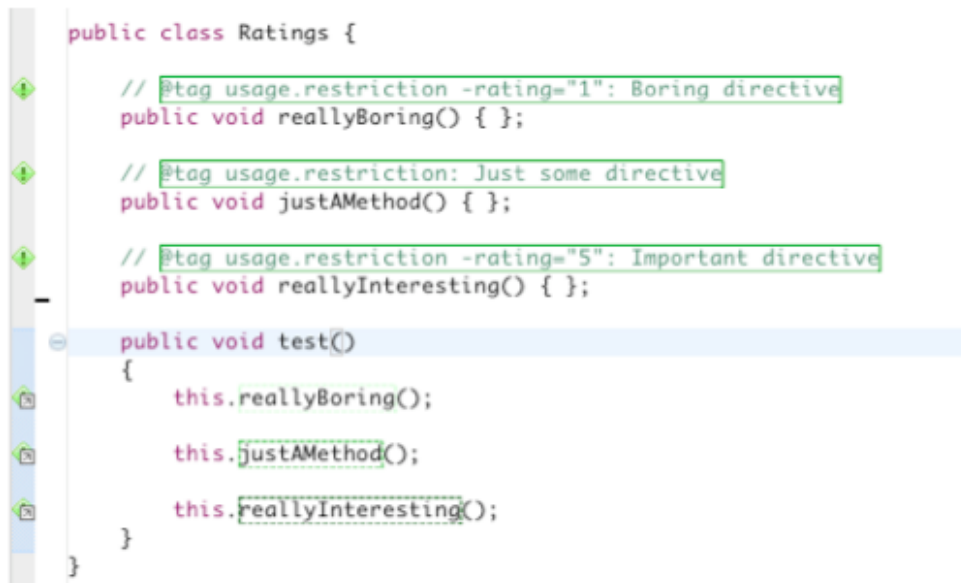
5.3 Contextual presentation

The main feature of *eMoose* is its ability to present decorations over calls to functions with associated directives. We now explain how this is implemented.

The *Eclipse* IDE maintains an *annotation model*, an internal collection of `Annotation` objects. Each annotation has a type that determines its presentation, and is associated with a specific range of characters in the source code. This model is designed for efficiently making transient visual changes (“annotations”) over the source code. Internally, it is used for things like the red underlines that appear

under syntax errors, the yellow underlines for warnings, or the strikethroughs for deprecated methods. The decoration over the code is accompanied by an icon on the left bar and an indicator on the overview bar.

Eclipse allows plug-in authors to define new annotation types with some custom visual properties. In writing *eMoose*, we defined new annotation types. Five directive types correspond to five possible directive ratings (from 1 to 5). Each is visually represented as a green dashed box. In each subsequent rating level, however, the contrast of the green against the white editor background is increased, so that higher rated directives are more visible. This is illustrated in Fig. 5.4, where the decoration on the first call in `test` is barely visible while the decoration for the last call has high contrast. We created a similar range for to-do comments. Ideally we should have different types with different presentations for each directive type, but at present *Eclipse* does not offer sufficient decoration options to make this worthwhile.



```
public class Ratings {
    // @tag usage.restriction -rating="1": Boring directive
    public void reallyBoring() { };
    // @tag usage.restriction: Just some directive
    public void justAMethod() { };
    // @tag usage.restriction -rating="5": Important directive
    public void reallyInteresting() { };
    public void test()
    {
        this.reallyBoring();
        this.justAMethod();
        this.reallyInteresting();
    }
}
```

Figure 5.4: Contrast level of method decorations adjusted by ratings

When the user opens a new source file or the code of the current file changes, the relevancy tree is created or updated. *eMoose* then attempts to synchronize the details in the tree against our custom annotations in the *Eclipse* annotation model. To do so, it maintains a bidirectional mapping between call nodes in the relevancy tree and annotation objects from the annotation model. The synchronization process works as follows:

1. Take all current *eMoose* annotations for the current source file and put them in the kill list.
2. For every call node in the relevancy tree for the current source file:
 - (a) Scan the subtree in the relevancy graph that is rooted at the call node and obtain the set of all KIs.
 - (b) Filter out any KIs whose types or rating fall outside or below the user's current preference settings.
 - (c) If there are no remaining KI, continue to next call node in relevancy tree.
 - (d) If the only KIs are tied to a dynamic type (rather than the call node), check if polymorphism support is enabled; if not, continue to next call node.

- (e) Identify the KI with the highest rating in the subtree, and obtain its type and rating. Determine the appropriate custom annotation type to match that type and rating.
- (f) Check if there is an `Annotation` object already associated with the current call node. If there is, update its type if necessary, and remove it from the kill-list. If there isn't one, create a new annotation object with the necessary type.

3. Eliminate all annotations which remain in the kill-list.

When this algorithm finishes executing, every call that has an associated KI that falls within the viewport of the caller and meets the filter settings is decorated. The contrast of the decoration depends on the highest rated directive in the subtree for that call. Calls whose targets do not have associated KIs are not decorated, offering *some* assurance of lack of KIs. The problem with this assurance is that with the present implementation, users cannot distinguish between targets that have no associated directives even though they belong to APIs that have already been tagged with directives, and targets from other APIs that have simply not been tagged. A planned extension will keep track of all packages for which at least one KI has been created. Calls to methods in that package will be decorated in gray to indicate that there really is no associated directive.

5.4 Augmented *JavaDoc* hover

Once the user decides to explore a decorated call, hovering over it opens a floating tooltip window with two panes. The upper pane contains the standard *JavaDoc* presentation which would have been displayed by default, while the lower pane presents KIs. Each KI knowledge item is preceded by its type, helping readers quickly distinguish its purpose before noticing the text. Users are expected to first read the KIs in the lower pane, and decide accordingly whether to read the entire *JavaDoc* in the upper pane.

In non-polymorphic situations, the lower pane looks like a two-level tree - it lists the method as the root and all KIs as its subnodes.

In polymorphic situations, however, the tree is deeper. The statically-invoked version of the target becomes the root, and its associated KIs become child-nodes. Overriding versions and their KIs are represented via subtrees with a similar structure, but all of them are at the same level, just as they are in the relevancy tree. If KIs are only associated with the overriding method but not with the overridden version, the latter is grayed out so users can focus on the former. For example, Fig. 5.5 presents the *JavaDocs* for the method `containsAll` in the standard `JAVA Collection`. In the lower pane, however, that method is grayed out to indicate that it has no associated KIs, while the subnode for `containsAll` in the subtype `Bag` appears with full contrast to indicate that there are associated KIs if the actual dynamic type is `Bag`.

One drawback of the use of the *JavaDoc* hover is that an extra step is necessary for a user to understand why a particular call was decorated. To alleviate this, *eMoose* also supports an *overlay layer* that presents all KIs in a semitransparent “bubble” next to relevant locations. However, this increases clutter and received negative feedback from pilot users.

5.5 Related work

Before we proceed to the evaluation of *eMoose*, we discuss related several tools in the domain.



Figure 5.5: Example of JavaDoc hover for polymorphic code

5.5.1 Improving reference documentation

The central role that reference documentation plays in software development has motivated various attempts to create better and more useful formats, such as Soloway's two-sided booklet [75]. Since the introduction of *JavaDocs*, automatically-generated API reference documentation has gained widespread acceptance for many modern language. Because of the widely standardized formats, it offers many opportunities for researchers to test out incremental enhancements that preserve the look-and-feel of the underlying format.

Early versions of the *JavaDoc* tool produced fixed HTML files which were difficult to scrape and recreate in enhanced forms. Newer versions, however, offered hooks that allowed *JavaDoc* generation to be easily customized. One successful example of a *JavaDoc* enhancement is *Jadeite* [81]. The tool builds on research [80] which showed that developers face difficulties in determining how to perform a certain operation when no single direct method exists to accomplish it. To this end, it allows method placeholders to be defined, with instructions on the appropriate sequence of operations to be invoked. These placeholders appear in the lexical table of methods, and can be found by developers seeking a method with the corresponding name. Other features of the tool include the addition of object construction samples that are automatically gathered from publicly-available code, and the ability to vary the size of elements based on the prevalence of their use in the wild.

Another tool from the same time called *Apatite* replaces the traditional static form of a *JavaDoc* document with a more concise and interactive version. The tool shows a small selection of the most popular elements in the API, and allows the user to interactively collapse or expand listings, search, and make selections. It also provides suggestions based on the user's selections.

eMoose is fundamentally different from the above tools in its goals and mode of operation. Whereas these tools aim to assist developers who are already looking up information on a specific class or trying to learn the API, *eMoose* focuses on making developers aware that they may want to consult the documentation. *eMoose* works within the IDE and aims to provide value to the developer before there is any decision to explore documentation. Furthermore, while *Jadeite*'s mechanism of placeholders aims to help developers who want to add a new call, *eMoose* is focused on method calls that are already there.

5.5.2 Search tools for delocalized knowledge

eMoose is concerned with addressing the problem of delocalization, where important information is present in one medium but is useful in the context of another. Delocalization and heterogeneous information present a more general problem in software engineering. It is not rare for an entity such as a line of code to appear in different artifacts in project support tools, such as commit logs, bug reports, electronic communications and other documents. One tool that addresses these problems is *Hipikat* [20], which builds a representation of the knowledge networks and allows users to search it. Though the scope of this tool is much greater than that of *eMoose*, it does not address the problem of directive awareness in documentation. More fundamentally, however, *Hipikat* is an active search tool that requires users to initiate a search for specific artifacts. *eMoose*, on the other hand, is a passive tool that aims to operate in the background and alert users to the availability of information.

5.5.3 Comment parsing and checking tools

Automated software verification is a major focus of contemporary software engineering research. Many tools (e.g., [8], [57]) can apply static analysis techniques to check programs against formal specifications. At present, however, the applicability of these techniques is limited because the specifications must typically be provided by humans in a time-consuming process that requires proficiency in the formalism. For these reasons, in recent years several researchers began exploring automated ways to elicit some specifications from natural text comments and documentation.

One of the first attempts at automatic recognition and verification was carried out by Tan et al. [84]. Coming from an OS background, they chose to focus on locking correctness, as that is relatively easy to analyze statically and also to recognize as natural text. They argued that focusing on a specific type of comments rather than general comments allowed them to improve recognition percentages. By constructing locking specifications for many *C* functions in *Linux* and comparing them to the results of an analysis, they were able to identify previously unknown discrepancies that were filed as actual bugs. In subsequent work [83], they applied additional recognition techniques to further improve the quality of their engine, and were able to find locking errors in non-OS programs.

Other NLP based techniques were presented by Hill [42, 41], who proposed applying them to comments for automated verification.

Another tool with similar goals is Zhong et al.'s *Doc2Spec* [91]. This tool is particularly interesting because it focuses on APIs and *JavaDocs*, rather than actual programs. The tool first parses the summary sentence from each method's documentation to determine what it is supposed to do. This is represented as an action and associated target resource, where the action belongs to one of several categories. The representations for multiple methods operating on the same resource are then joined into an automata. The tool then uses static analysis to determine what actions actually occur and warns of potential discrepancies between specifications and implementations. The tool was used to detect actual errors in a real system.

The above tools aim to provide the significant value of identifying actual bugs. *eMoose*, on the other hand, does not attempt to automatically detect bugs; its more modest goal is to help increase the developers' awareness of potential risks. This, however, may help maintainers detect bugs faster, as we shall see later in this dissertation, or may even help developers not create the bugs in the first place. It is important to note though, that the above tools intentionally operate on a more limited scope: Tan et al.'s tool is limited to issues such as locking, while Zhong et al.'s tool is focused on specific method patterns that can be determined from summary sentences. Directives, which can appear anywhere in the documentation, can convey much more complex instructions and issues. In addition, some directives are

informative: violating them is not necessarily an error, making it more complicated to provide automatic awareness.

Chapter 6

Comparative Lab Study of eMoose

In this chapter we describe the comparative lab study that we have conducted to evaluate the severity of the neighbor knowledge awareness problem in code, and the impact of *eMoose*.

This chapter is the largest in this dissertation, and is organized as follows: Sec. 6.1 presents the intent and goals of our study. Sec. 6.2 describes the decisions that we have made about its design and the rationale behind them. Sec. 6.3 describes the recruitment process for the study, the procedures, and the subjects. Next we devote a full section (Secs. 6.4, 6.5, 6.6, 6.7, 6.8, 6.9) to each of the tasks, in which we introduce the API, describe the task, present the results, and discuss them. We then describe additional general results and findings in Sec.6.10. The limitations of this study are discussed in Sec. 6.12.

6.1 Intent and goals

The primary focus of this dissertation is on the neighbor awareness problem in software implementation and maintenance: whether developers become aware of directives in methods that are invoked by the code that they are examining.

This existence of this problem in real development scenarios is quite plausible. It is not difficult to envision, based on the examples from Chap. 4, situations where certain directives would be missed with severe consequences. Indeed, in my personal experience as a developer and even while working on the *eMoose* tool, I have frequently encountered problems due to unexpected details in the documentation. Similar anecdotes were reported by others.

To properly establish the importance of the problem, however, it is necessary to demonstrate that given a situation where a directive in an invoked method is important, a significant portion of developers with present techniques will indeed miss this directive. While such evidence may not be predictive of whether a particular developer will miss a particular directive in real-life work, it would establish the potential for such problems and may help us understand better why such situations occur.

To establish that my approach of knowledge pushing has the potential to be useful in real-life situations, it is necessary to first demonstrate that given a situation where a directive is important, the tool will have a significant impact on whether a directive is missed. Even if there is a positive effect, it is necessary to ensure that the tool is not prohibitively expensive to use. If the effects of the intervention are too strong, then there is a risk that developers would explore every decorated call as soon as they first encounter it. This will not only waste much time, as some directives are already known or irrelevant, but will also disrupt the goal-driven approach taken by many developers. To understand the greater implications of the *eMoose* interventions, we must understand how it is used or what its important mechanisms

are.

In order to address these initial concerns, I chose to conduct a lab study in which multiple developers would face identical situations where a directive plays an important role in resolution, and I would measure the proportion of developers who fail to become aware of it. In other words, I attempt to induce situations in which the directive could potentially be missed. My goal is not merely to obtain quantifiable evidence for the incidence of the problem, but also a qualitative understanding of the sources of difficulties.

Nevertheless, we must also evaluate whether the proposed interventions, first decorating calls whose targets have directives and then explicitly listing directives in the text, have a positive impact on reducing these difficulties. Since these interventions also carry the risk of distraction, it is important to also verify that the costs are not greater than the benefits.

My study is therefore comparative, so that each task will be performed by a roughly equal number of subjects using *eMoose* and of controls using standard tooling. The limitations of the study will be discussed in Sec. 6.12.

In reading the description and results of the study, consider the following four research questions: **1)** How well did controls and *eMoose* users perform these tasks? **2)** Why did some controls who are using existing tooling fail to fix relatively simple problems, and what differentiated those who succeeded? **3)** In what ways did *eMoose* change how its users performed the tasks? **4)** What are the implications for documentation writers?

6.2 Study design decisions

We now turn to describing the “design space” for our study and the steps and decisions that we followed to come up with the final design.

6.2.1 Decision to use multiple tasks

Rather than have subjects perform one large task, I decided to have them perform a sequence of multiple independent tasks.

One reason for this choice is that it allows us to obtain multiple and independent “data points”, for each subject. This will allow us to not only compare the performance of multiple subjects on the same task, but also the performance of the same subject across multiple tasks.

A second reason, elaborated later, is that this separation will allow us to focus tasks on smaller code sections and the use of documentation, and reduce the influence of other development activities.

A third, and perhaps most important reason, is that this separation allows us to try and separately evaluate each of the mechanisms by which *eMoose* can affect users:

1. The decoration of methods should lead users to investigate methods that they may not have otherwise explored or that would have been investigated much later.
2. When documentation is read, the augmentation of the *JavaDoc* hover to reveal a list of directives should increase the likelihood that readers become aware of relevant directives.
3. As we have noted, there is a significant risk that the approach would distract users by presenting too many decorated methods with directives that are not relevant to the task at hand.

4. In the case of polymorphic code where directives are associated with a specific dynamic type, the decorations should make the user aware of the potentially important information.
5. Also in polymorphic situations, the augmented *JavaDoc* hover should make it easy to locate the directives associated with each overridden version of the method.

There are therefore five tasks corresponding to each of these mechanisms. In addition, I added a sixth task to evaluate whether newly added calls are explored, but as shall later be described, ended up omitting that task. These tasks, their purposes, the mechanism by which *eMoose* affects them are summarized in Fig. 6.1.

Task	API	Codebase	Task attempts to evaluate if subjects...	Expected mechanism of eMoose
1	JMS	Short fragment	can find directives in seemingly trivial call	Call decoration leads to reading
2	JMS	Short fragment	can find directive in verbose text	Lower pane with explicit directive list
3	Swing	Full program	suspect directive presence in seemingly valid call	Distraction from many decorated calls
4	Swing	Created by subject	investigate calls they have just added	Call decorated as soon as it is added
5	Collections	Custom program	suspect conformance violations in subtypes	eMoose presents directives from subtypes
6	Collections	Custom program	effectively find directives in overridden version	eMoose presents directives from subtypes

Figure 6.1: Summary of tasks in the lab study

The main drawback of using multiple tasks is that there is a significant risk for a “memory effect”, in the sense that with each successive task whose solution depends on directives in the documentation, subjects are more likely to pay careful attention to documentation and specifically seek out directives when working on the next task. Eventually, this behavior may differ significantly from the level of attention that the same subjects devote to documentation in earlier tasks and in their everyday work. Because of this, both controls and *eMoose* users might be more likely to identify directives towards the end of the study than they would at the beginning.

While I cannot fully eliminate this effect, I attempted to minimize it in two ways: First, all subjects perform the tasks in the same order, so that effects should be comparable across subjects. In other words, any differences we see between the conditions at the later tasks are valid in the sense that subjects in both conditions have the same prior experience from the study. Second, the tasks are ordered so that earlier ones evaluate whether subjects become aware of directives when they are not specifically expecting to find any, whereas later ones involve situations in which subjects should expect to find directives but may have difficulty doing so due to polymorphism.

6.2.2 Decision to use limited set of directives

This study attempts to obtain meaningful quantitative data on the differences between developers who use existing tools and those who also use *eMoose* in becoming aware of directives in invoked methods. Since directives are very different from one another in their content, purpose, and importance, I chose to restrict the study to a small set of specific directives and compare the subjects’ awareness of each of them.

As part of my effort to understand directives better and create libraries of directives for use by *eMoose*, I systematically surveyed several APIs and tagged directives that could potentially pose problems if ignored. During this effort I used special tags for directives which I felt to be particularly “interesting”. A directive was considered interesting if it were straightforward and yet was not obvious from the name of the method whose documentation contained it.

When I turned to search for directives for our study, I used the set of specially tagged directives as a starting point. In deciding whether these could potentially serve our goals, I laid out several criteria:

First, I required that the directive be straightforward to interpret even without significant domain knowledge. Thus, even though it could refer to less familiar concepts, it should treat those concepts as “black boxes” and allow most readers to understand what it requires. Second, I required that the directive be unambiguous and specific, so that it would be straightforward for readers to determine whether a certain call fragment complies with it, and what the consequences of violating it would be. Our intention is also that if a user is looking for an explanation for a specific erroneous behavior, and if that behavior is listed in the directive as a consequence of ignoring it, then it would be straightforward for a user examining the directive to correlate it to the erroneous behavior.

Obviously, many directives meet these criteria, so I attempted to find a set that would represent a wide range of directive types, while covering a limited number of APIs. I then looked at public code samples that invoked the associated methods to see if I could use them with minor changes as codebases for the tasks. In all samples, I required that some other calls in proximity to the invocation of the method containing the prospective directive would have some associated directives so that these calls could serve as potential distractors.

I ended up selecting a set of six directives for this study, one for each task, and these will be described with the tasks and codebases.

6.2.3 Decisions on codebase scale

My study is designed to allow a comparison of the subsets of calls that different subjects choose to explore out of the set of calls to which they are exposed. To this end I needed to ensure that all subjects are exposed to the same set of calls for comparable amounts of time.

Since subjects are given a time limit for each task, these exposure periods also need to be sufficiently long to give everyone a chance to notice every call and to read it with sufficient attention. To allow this, subjects must be immediately exposed to the relevant calls, rather than have to spend significant effort on searching them. Otherwise, success may depend on an early identification of the focus region which would leave sufficient time to identify the offending call and the directive. In other words, it may be difficult to disentangle the time it takes to become aware of the directive from the time it takes to encounter or become aware of the call in the first place. I initially carried out pilot sessions using real full-scale programs with complex control flows. I found that subjects differed so much in their interprocedural exploration and search strategies that they were not exposed to the same code for fragments for similar amounts of time, which would have made it difficult to compare what they read.

To this end, I decided to either use very short code fragments, in which the relevant calls would be visible most of the time, or to use specific fragments within larger programs with a “cover story”. In the latter cases, I guided subjects to relatively limited code scopes, representing it as if they have already debugged the program to the point where the offending region was found, and they must now continue and identify the bug. All subjects were therefore faced with the calls immediately or very early, making an analysis of their moves meaningful. Another benefit of my choice to use smaller fragments is that it was easier to obtain a codebase that can be comprehended in limited time without significant background knowledge about the program, domain, and API.

An obvious risk of a limited scope is that one can use a *systematic approach* and follow a process of elimination within the allotted time; this is not possible for longer programs [76]. This does not pose a threat to validity because all subjects are given sufficient time to explore every call in depth, if they so wish, even if they do not follow a systematic strategy. However, the *eMoose* interventions have no impact in a systematic approach, so my findings may fail to identify an impact. If, on the other hand, we find that subjects without *eMoose* face significant difficulties and fail to become aware of directives even in such small fragments, and that subjects with *eMoose* are significantly more successful, then it can be

argued that the problem could be even more severe in larger programs.

Another risk of using only small code fragments is that it is not clear whether my findings will translate to realistic large programs. To obtain some data on this question, I decided to have one task which will use an entire (though not very long) program. This task also offers us enough opportunities to create potentially distracting decorations, and to evaluate whether developers investigate all directives even in blocks that are not necessarily related to their goals.

A related decision that I made was to forbid the use of the *Eclipse* interactive debugger on all tasks but the full-program one. Since my goal was to investigate which calls would be examined in an attempt to understand a failure in a small fragment, I had to prevent users from stepping the debugger until the point of failure. As interactive debuggers are used primarily for exploring large programs with complex interprocedural connections, I were not withholding a critical tool from subjects. In addition, during the pilot sessions I found that subjects tended to try and address the failure by stepping through code and into invoked functions. Since many of the APIs used in my tasks are provided via interfaces that can be implemented by different vendors, pilot subjects either got frustrated when they couldn't see the implementation or when they became lost in the complex implementation by a specific vendor. Note that in the one task that used a full program, use of the debugger was permitted as it is an important tool for understanding complex control flows.

6.2.4 Decision to use customized programs

Although it would have been beneficial to use tasks that are based on actual problems that developers have faced with actual codebases, that was not practical for this study. First, there are no straightforward ways to search existing project bug databases for bugs that resulted from a lack of awareness of a directive, as typically the manifestations are described rather than the underlying cause. Second, even if we could find such programs, it would be difficult to distill them into a short and simple fragment that subjects can easily understand and debug. Third, it may be difficult to place enough “baits” to measure distraction.

For these reasons, I made the decision to “manufacture” codebases in which awareness of directives can lead to resolution. For the first three tasks, I chose to use official examples for the API and artificially break them. Such examples are generally simple and straightforward and are meant to be understood and used for learning. Since one of the goals of *eMoose* is to assist with API learning, a lack of awareness of an important directive may represent a breakdown in the learning of the API. However, I took special care to create breakdowns that result from very minimal changes yet are plausible, such as erasing, moving, or renaming a single call. I have frequently encountered such bugs in my development experience. For the fourth task, which involved writing new code, I made slight modifications to an official example and had developers implement new functionality. For the polymorphism tasks, I chose to design a code fragment from scratch so that the subject's work could be limited to becoming aware of the conformance violation.

6.2.5 Decision to use fixed codebases

The decision to choose a specific subset of directives aims to allow a comparison of directive awareness between subjects. However, it is very likely that the location and context of a particular call, and the goals of developers who are exposed to it, affect the decision of these developers on whether to explore it and the amount of attention to pay to the text. To make a reliable comparison, all subjects need to see the same call and context.

In addition, *eMoose* currently handles new calls by decorating them as soon as they are created. It does not, at present, augment function listings in the IDE such as the autocompletion recommendation mechanism. Therefore, if our target directive is in a call that is not initially in the code, there are no

guarantees that the call would ever be added, reducing the sample sizes on which we can compare awareness of the associated directive.

For these reasons, we decided to base all but one of the study's tasks on an existing codebase and not require subjects to write any new code. The other task aimed to evaluate whether *eMoose* users immediately notice decorations that appear on newly-added calls and therefore required subjects to add new code.

6.2.6 Decision to use debugging tasks

A major choice in my design was to cast all but one task as a debugging task. The tasks are designed so that the bug is caused by a violation of the directive. If a subject is capable of identifying (and preferably fixing) the bug, then there is evidence that the subject became aware of the directive. On the other hand, if the subject never becomes aware of the directive, then the task is essentially unsolvable. Recall that our criteria in selecting directives is that they are phrased in a way that is clearly related to the erroneous behavior.

A major limitation of this decision is that focused debugging does not actually represent the main usage situation for which *eMoose* is designed. The goal of *eMoose* is to help developers become aware of the directive as soon as they have added a call and when they are examining existing code or learning the API. While the error would be in place, the resolution will preferably occur before executing an incorrect program. However, success in debugging gives us an objective measure of whether the subject has become aware of the directive. In addition, if subjects fail to become aware of a directive during the intense and focused effort involved in debugging, I argue that there is a lower chance that they would become aware of directives in other situations, making our support even more necessary. Nevertheless, the tool clearly needs to be evaluated in the field.

6.2.7 Decisions on time limits and measurements of success

By this point, I have decided to have 6 tasks in the study:

- A debugging task involving a small code fragment where a directive will be hidden in an unexpected call.
- A debugging task involving a very small code fragment where a directive will be “hidden” in a method with a very straightforward signature.
- A debugging task involving a full program with significant potential for distraction
- A task where the developer adds new code.
- A task that tests the ability of the developer to become aware of a conformance violation in an overriding method.
- A task that tests the ability of the developer to find directives added in overridden methods.

Since I needed subjects to perform multiple tasks, I had no choice but to place a cap on the amount of time available for each task. To facilitate comparisons between subjects, leftover time from one task *could not* be transferred to later tasks. In the end I chose to let developers work for up to 15 minutes on each task, with an additional 5 minutes of instruction reading and preparation prior to “starting the stopwatch” on each task. This allowed me to fit the six planned tasks in two hours, budgeting another half an hour for introductions and a debriefing questionnaire.

Pilot sessions confirmed that this time limit would be quite sufficient for the successful completion of all tasks, even for subjects who would not be using *eMoose*. Specifically, in the tasks involving small code fragments, this limit would allow subjects to read everything, and with time to spare. In the full-program debugging task, the program is small enough to allow it to be surveyed in the given time, although success would require subjects to identify and focus on the area of the breakdown. In the task that involves adding code, users need to add less than 10 lines. In the task that requires users to become aware of a conformance violation, the code is so short that success would largely be determined by the ability to consider the option of a conflict, and then to understand its implication. In the task where users just need to find violating directives, there is sufficient time for a systematic survey of all the overriding versions.

Because of the time limits, the definition for success in performing a task requires that it is observed within the first 15 minutes. For analysis purposes, I let subjects work for a few minutes, so that I could determine if they were on the cusp of success, though that rarely ended up being the case. Since all tasks, with the exception of the one involving writing new code, use a fixed amount of code, given unlimited time it is likely that every subject would eventually be able to explore every option and become aware of the directive. Therefore, success rates on tasks must be interpreted as measures of efficiency rather than of capability. If *eMoose* users are more successful, it will be because they become aware of the relevant information faster, rather than that they have gained capabilities that allow them to solve otherwise-unsolvable problems.

Success in the first three debugging tasks is defined as the ability to *identify* the cause of an error, and to present a *well-motivated* and correct plan for a solution. In other words, if a subject accidentally stumbles across a solution, he would still have to explain why the problem happens and why the solution works. Since the directives are chosen to be straightforward, once the subject becomes aware of the directive, the solution for these tasks should be immediate and straightforward. Since the actions to break these programs are minimal, fix time is extremely short. In fact, I decided to count success as either the point of actual fix, or the point where a subject voices the fix.

The interpretation of success is different in the other tasks. For the task where developers have to write code, success is achieved when the program works as expected. For the two polymorphism tasks, success was measured as the ability to identify the directives, without requiring a fix since a solution would require significant changes to the code or finding an alternate class to provide the same service.

6.2.8 Decision to have two conditions

To get the maximal value from a limited subject pool, I decided to make the study compare between subjects who use a standard *Eclipse* distribution, and subjects who use *Eclipse* with *eMoose* and our set of annotations for the API. I decided not to have additional groups that will only receive the decoration support or only the augmented *JavaDoc* hover. As a result, I will not be able to fully differentiate the impact of each of these mechanisms, although some of the tasks are designed to specifically test them.

Rather than divide the subjects into two groups for the entire duration of the study, I chose to have each subject perform some tasks in the control condition (CTL) where *eMoose* is not available, and some tasks in the experimental condition (EXP) where it is available. One benefit of this decision is that it minimizes the impact of a particularly skilled or unskilled subject on the overall evaluation of the impact of the tool. A second benefit is that I get all subjects to experience the tool and provide feedback. The most important benefit, however, is that in addition to comparing the performance between subjects in the two conditions for each task, it is also possible to compare the performance of the same user across tasks and evaluate the impact of the tool. While the tasks are different from one another, if we see a pattern where subjects generally succeed more with the tool then that is a positive indication about the

strength of my approach.

The risk of this approach, however, is that we are potentially creating a memory effect or priming effect. If a subject performs a task with *eMoose* and receives a benefit from it, and then has to perform the next task without it, he may be more cautious and systematic in that task to compensate for the lack of cues.

My plan was to have six tasks that will be carried out in the same sequence by all subjects. I decided to split them into three conceptually-related pairs. Within each pair, every subject will perform one task in the control condition and one in the experimental, but the order is random.

One can look at the assignment of a particular subject as a three-digit binary number, where each digit represents whether he is performing the first or second task in the corresponding pair in the experimental condition. My initial intention was to use a sequential and wrapping binary count to ensure that each of the eight possible combinations would appear at a uniform distribution, but things turned out differently. First, I was not sure how many subjects would participate in the study (due to gradual response and a paper deadline). Second, one subject disrupted the count by performing only part of the study. Third, one of the tasks was eventually eliminated. Thus, I tried to follow such a count but had to do some manual balancing to ensure an equal or nearly equal number of subjects in each condition for each task. Nevertheless, the assignment was always done *before* the subject showed up, reducing potential assignment bias.

6.2.9 Choice of APIs

While the standard library of JAVA provides a wide range of capabilities, few users are familiar with all its intricacies. In addition, many libraries and APIs meet needs not supported by the standard libraries. For example SUN publishes the *J2EE* family of APIs to provide functionality necessary for large scale-enterprise applications, while *apache* provides a variety of components and functions to supplement things missed in the standard library.

Since my intended subject population was large and diverse, it seemed unlikely that I could find APIs and functions with which all subjects are equally familiar. Instead, I chose the opposite route: Within the standard library, I chose to use obscure or specialized classes which none of the subjects had used. I also selected APIs that are widely used in enterprises but which none of the subjects have used. I screened subjects for familiarity with these APIs and classes prior to beginning the study.

In many respects, my study evaluates the learning of a new API, rather than the use of a familiar one. However, since most APIs consist of many more classes and functions than those used in a single program, learning an API is a gradual process, over which certain directives and caveats are learned by experience. Even experienced developers can be surprised by the directives associated with a method that is less frequently used. In addition, APIs or parts of APIs are often learned from code examples, as they are in this study.

In selecting the actual APIs for my study, my first principle was to select high-quality and widely-used APIs. My second was to choose ones that are concerned with straightforward concepts that subjects can understand easily. A side benefit of these choices is that subjects are likely to encounter these facilities in their future, perhaps making them more likely to become involved in the tasks and learn from them.

I decided to use three different APIs, so that each pair of tasks would involve the same API. The first two tasks use the *Java Message Service* (JMS), an API by SUN for providing robust and reliable communications between JAVA processes at a much higher level of abstraction than sockets. The JMS API is today part of the *J2EE* standard and is widely used by many projects. The next two tasks use a less-

familiar class from the SWING API, the standard user interface toolkit distributed as part of the JAVA standard library. The last two tasks use the JAVA collections framework but also a set of additional collection classes from the *apache commons* project.

6.2.10 Decisions on amount of training

Another important decision had to do with the amount and nature of training provided to subjects on the use of *eMoose*, and on each of the involved APIs.

My decisions here were unfortunately motivated by time constraints. Since subject recruitment is more difficult for longer sessions, I capped sessions at two and a half hours. I had planned six tasks that would take about 20 minutes each, with sufficient time for reading task instructions. This only left up to 30 minutes for everything else, including time for filling a debriefing questionnaire, and a “safety margin” due to computer problems, interruptions, etc.

The main consequence of this time limit is that I decided not to have a “pilot task” before the actual tasks start. In other words, after subjects receive the initial training, they are immediately “thrown into” a task that counts. If they are performing the first task in the control condition, they may not truly understand the importance of reading documentation, and may approach the problem as they would in everyday work, for example with the integrated debugger. If they are in the experimental condition, they may ignore *eMoose* completely since they have not benefitted from it or used it directly in the past, or they may get overly distracted by it and follow every directive.

However, pilot tasks have some drawbacks, as users having to go through a series of pilot tasks may increase fatigue and a memory effect as described earlier. Also, if *eMoose* users are successful with only minimal training, that may bode well for adoption in the real world, where developers are less willing to spend effort on learning new tools.

All subjects were therefore given materials to read about *JavaDocs* and *eMoose*; these booklets are reproduced in Appendix A. These materials were supplemented by a verbal presentation from the experimenter, which covered: the notion of directives, examples of directives, the use of *eMoose*, and the use of *eMoose* in polymorphic situations. During the study, subjects received background information about each API when they first encountered it.

6.2.11 Decision not to use think-aloud or gaze tracking

A major goal of my study was to learn more about how subjects understand code in the presence of documentation. In particular, I wanted to learn how they make documentation reading decisions, when they become aware of directives, when they form certain hypotheses, and when they refute them. The only way to obtain the subjective facets of this knowledge is to have subjects provide it, typically by *thinking aloud*. Unfortunately, the process of vocalizing thoughts is known to have significant impact on subjects under certain circumstances [73].

In this study, a requirement to think aloud would have carried significant risks: First, subjects may spend a longer amount of time with each idea, call, or text line, as they would vocalize it. The additional time may lead them to recognize errors, mistakes, and deeper meanings. While this may increase success, it steers us away from the level of concentration these subjects would typically exhibit. Second, the need to think-aloud might steer subjects towards a more structured and less opportunistic investigation, since in many ways it is easier to vocalize and rationalize a process of elimination.

Since one of my primary goals was to evaluate whether directives are noticed, I could not accept the risk of influencing attention with a think-aloud and told subjects that they did not have to do so.

Note though that subjects were not prevented from subvocalizing as they read or thought, or even from vocalizing out of their own volition.

I decided not to use gaze-tracking equipment in my study, as the potential rewards were not sufficient to justify the logistic complexities. When developers examine code, gaze tracking would have indicated how frequently they examine a call without investigating its documentation. However, since many of the code fragments are very small, subjects can often “take the whole picture in”, and the data may be ambiguous. In addition, in my pilot sessions subjects often vocalized or used the mouse to trace code as they read, so that approximation is available. Similarly, when *JavaDocs* are inspected the font is relatively small so it is possible to absorb many lines at the same time. Use of subvocalization, mouse, and scrolling was deemed a sufficient approximation.

The main benefit of gaze tracking might have been in determining whether a subject is actually reading an open *JavaDoc* hover or staring at nearby code. However, since the hover is often overlaid on the code in close proximity to the call, it may be difficult to precisely establish focus. Therefore, I defined the “reading” of a *JavaDoc* to be the entire time that the hover is open and visible. The reading durations I report are therefore upper bounds, since in very long readings it is possible that some of the time is spent looking at the code or glancing elsewhere.

6.2.12 Policy on answering questions

Subjects performed all tasks in the physical presence of the experimenter (myself), who was there to handle the transition between tasks, monitor the time, and address issues.

Since subjects were working with unfamiliar codebases, APIs, and concepts, and since they did not have an execution context or debugger for some of the code fragments, I needed to allow subjects to ask questions. In conducting several early pilots, many of the questions repeated themselves and were straightforward to answer. In particular, there were many questions about the state of the system and certain conditions. For example, “is the broker running correctly” or “is the network address correct?”. Scripted answers were therefore created to many questions, to ensure that all subjects received equivalent assistance.

However, not all questions could be anticipated in advance, and the experimenter occasionally had to respond to a new question raised by a non-pilot subject. In those cases, the answer was added to the set to ensure that future answers to the same question, if it was asked again, would be identical.

To cope with those unanticipated questions that were not asked in the pilot sessions, I made three decisions: First, questions asking for help in finding the correct answer (e.g., “which statement fails?”) would not be answered. Second, questions which the subjects could answer with the means already available to them (e.g., “what does the function do?” when documentation was available) would also not be answered. Third, if a subject asked if a specific clause in the documentation text was the cause of the bug, a truthful answer would be given. Namely, if it was not, then the subject would be told so. If the clause exactly explains the error, the subject would be encouraged to try and fix the problem. If the clause was only part of the problem, the subject would be told so and encouraged to continue searching.

6.3 Study procedures and subjects

6.3.1 Subject recruitment

To have reasonable statistical power under the assumption that variance across conditions would be much larger than variation within conditions, I aimed to have at least 10 subjects perform each task in

each condition. Thus, I needed at least 20 subjects who would be available for a session lasting up to 2.5 hours.

To obtain these many subjects for such a long time commitment I chose for practical reasons to aim my recruiting efforts at students on the Carnegie Mellon University campus, even though most of them do not have significant industrial experience. Like many other studies that use students rather than very experienced practitioners, the results are limited by the nature of this population.

I advertised on the university's online bulletin board (Fig. 6.2), and on physical bulletin boards in the computer science building (Fig. 6.3). Participants were requested to have significant experience in JAVA development, at least with one internship, and some experience in using the *Eclipse* IDE which is uniformly used in the university. The study promised a flat compensation of \$25 USD, as well as raffle tickets corresponding to the number of successfully-completed tasks as a motivation to perform well.

For a lab study focused on how Java programmers use existing documentation and on the effects of a new Eclipse plugin, we are seeking participants with significant experience in Java programming. Participants should have at least half a year of routine experience in Java, preferably in the industry or at least via internships. Familiarity with IDEs and preferably Eclipse is necessary.

The study will take between 2 to 2.5 hours, in which you will perform several debugging, maintenance and code inspection tasks on existing code using Eclipse. A shorter 1.5-2 hour version of the study is available with less tasks; contact me for details. Slots are available weekdays and weekends in the next 10 days. Compensation is 25 USD for the full session, and you will also participate in a raffle of a prize where the number of tickets will be determined by the number of successfully completed tasks. If you are interested, please fill the screening questionnaire below. Note that you are not expected to be familiar with the mentioned APIs.

Figure 6.2: Text of online recruitment ad for lab study

For a lab study of documentation use in Java and Eclipse we are seeking developers with significant experience in Java programming (preferably industrial, at least one internship) and with Eclipse (at least one month). Earn 25 USD and participate in a prize raffle! Study lasts 2 - 2.5 hours where you will be asked to find errors in several Java programs with and without our eMoose tool.

Figure 6.3: Text of online recruitment ad for lab study

All applicants were requested to complete a detailed survey form, reproduced in Appendix A. The form asked subjects to list their development experience in several languages, and then asked specific questions about JAVA development, familiarity with specific APIs, and the use of the *Eclipse* IDE. The goal was primarily to filter out candidates who lacked the relevant background.

6.3.2 Subject characteristics

The eventual set of applicants was quite diverse, given the limitations of the academic environment and the constraints of the summer vacation during which this study took place. I accepted a total of 26 applicants to participate in the study, of which 24 were males.

16 subjects were students in several professional-masters programs in information management and information technology. These students had relatively limited industrial experience, and my impression was that while JAVA was the language they were strongest in, they were not very experienced development in general. However, they have participated in at least one major internship in the US or in their home country and were about to join the US job-market, so they were included in the study.

These relative novices were balanced by 10 applicants with a much stronger but diverse background: Two CS undergraduate students with significant development experience, three students from a prestigious software engineering masters program, one experienced masters student from electrical engineering and another from language technologies, and three Ph.D. candidates in the field of programming languages with significant experience.

Subject	Degree	JMS 1	JMS2	Swing 1	Swing 2	Collections1	Collections2
1	BS	ctl	exp	ctl	exp	ctl	exp
2	MS	exp	ctl	exp	ctl	exp	ctl
3	PHD	ctl	exp	ctl	exp	exp	ctl
4	MS	exp	ctl	-	-	-	exp
5	MS	exp	ctl	exp	ctl	ctl	exp
6	MS	ctl	exp	exp	ctl	ctl	exp
7	MS	exp	ctl	ctl	exp	exp	ctl
8	MS	exp	ctl	ctl	exp	ctl	exp
9	MS	ctl	exp	exp	ctl	exp	ctl
10	BS	ctl	exp	exp	ctl	exp	ctl
11	MS	exp	ctl	ctl	exp	ctl	exp
12	PhD	ctl	exp	exp	ctl	exp	ctl
13	MS	ctl	exp	ctl	exp	ctl	exp
14	MS	exp	ctl	exp	ctl	exp	ctl
15	MS	ctl	exp	ctl	exp	exp	ctl
16	MS	exp	ctl	exp	ctl	ctl	exp
17	MS	ctl	exp	exp	ctl	ctl	exp
18	MS	exp	ctl	ctl	exp	exp	ctl
19	MS	exp	ctl	ctl	exp	ctl	exp
20	PhD	ctl	exp	exp	ctl	exp	ctl
21	MS	ctl	exp	ctl	exp	ctl	exp
22	MS	exp	ctl	exp	ctl	exp	ctl
23	MS	ctl	exp	ctl	ctl	ctl	exp
24	MS	exp	ctl	exp	ctl	exp	ctl
25	MS	exp	ctl	ctl	exp	ctl	exp
26	MS	ctl	exp	ctl	exp	exp	ctl

Table 6.1: Assignment of lab study subjects into groups for each task

Table 6.1 lists the subjects in the study and their group assignment for each task. Note that subject S4 had to leave the study early due to personal reasons after having arrived late. He completed the first two tasks, and was instructed to perform the short sixth task. I kept the results for the tasks he performed and included them in the results.

Also note that the fourth task (*swing2*) was eliminated halfway through the study and the collected results were not formally analyzed. As shall be explained in Sec. 6.7, this task aimed to identify whether users become aware of directives in methods that they have just added, and was the only method where subjects were asked to create new code. However, I found that most subjects failed to add the relevant code, thus nullifying the purpose of the study.

6.3.3 Preliminary procedures

An exclusive time slot for the study was coordinated with every subject. On arrival, subjects signed a consent form and then the session began.

Setting up the work environment

The subject was seated in front of the computer used for the study. The computer was a quad-core PC with 4GB of RAM, significantly above the minimal requirements for *Eclipse*. This hardware ensured maximal responsiveness even with the screen recording in the background, as I was worried that

slow *Eclipse* response time may affect the subject's attention and willingness to explore calls. The computer was equipped with a 20" widescreen LCD, large enough for comfortable viewing from the subject's seat.

The computer was running *Windows XP*, with which all subjects had working experience. On the screen, already started was version 3.4 of *Eclipse*, running on *Java 6*, with *eMoose* installed (but turned off). The *Eclipse* workspace contained several projects which were used in the study. Also open but minimized was a *Firefox 2* web browser, which contained bookmarks for the main *JavaDocs* page for each of the APIs used in the study.

The subject was then invited to change the resolution of the computer and the font type and size in *Eclipse* to his level of comfort. A few subjects took advantage of this option, which made text more readable for them but slightly changed the amount of information visible to each subject.

Subjects were then invited to choose between a standard low-end keyboard and a higher-end split-keys ergonomic keyboard. The majority picked the standard keyboard.

In accordance with Fitts' law [32], the characteristics and sensitivity of the mouse can have significant effect on the subject's ability to scroll and use the hover mechanism of *Eclipse*. Subjects were given the choice of using a standard low-end mouse, or of using the default mouse, a *Microsoft Wireless Laser Mouse 6000* which is very sensitive, has a scroll wheel and is considered usable for left-handed individuals. All subjects chose the latter, but many of them accepted the invitation to change the mouse speed and sensitivity to their comfort through the operating system.

Tutorial

Subject were then instructed to read the introduction booklet, reproduced in Appendix A, which covered the procedures of the study, a short eclipse usage test, javadocs, and the use of *eMoose*. While reading, subjects performed the *Eclipse* test with the experimenter. After the subject finished reading, the experimenter repeated the rationale behind *eMoose*, and demonstrated to the user how the tool is used, including examples of polymorphism. The interaction around this tutorial took up to 10 minutes. Once the subject was ready, the experiment moved to the next stage.

6.3.4 Process for each task

For each task, the subject read some background materials and was then taken on a quick tour through the codebase, and when relevant also through the execution results. The subject was reminded of the goals and 15-minute time limit, a small stopwatch was placed on the screen, and screen recording was started. The subject was asked whether he was ready to start, and then the stopwatch was started, and *eMoose* decorations were activated if the subject was in the experimental condition for that task.

To avoid pressuring subjects while still offering them opportunity to ask questions, the experimenter was in the room but not sitting next to them. Subjects could ask questions whenever they wanted, and if a longer interaction took place the experimenter stood next to the subject for a while and if necessary used a second mouse to point at things. The experimenter sometimes asked questions, but tried to time them for situations when the subject did not appear to be too busy. In particular, when the subject appeared lost, the experimenter sometimes asked if the subject already had some theory in mind. Care was taken not to steer subjects towards answers. During all interactions, the stopwatch was not stopped.

Subjects typically announced when they found the problem, or asked for confirmation when they thought that they had. In the debugging tasks they were asked to fix the problem and run the program to demonstrate that their solution worked.

When subjects ran out of time, they were allowed to continue for a few more minutes, under an understanding that they would not be credited for the success. If they still could not find the problem or chose not to continue, they were shown the solution in an effort to put them on equal footings with successful subjects when they arrived at the next task. In the case of subjects in the control condition, the *eMoose* features were turned on and they were given a chance to try and find the problem with it, before the actual answer was given.

Once subjects finished the study, they were required to fill a debriefing questionnaire, presented and discussed in Sec. 6.11.

6.3.5 Analysis technique

During the study, the experimenter wrote down the outcome for each task, forming a simple measurement of subject success rates. To facilitate further analysis, however, all sessions were recorded using the *Camtasia 4* screen capture program. I therefore have a complete visual record of the subject's actions and interaction with the IDE. The recorded voice track also contains the subjects' interactions with the experimenter, and some of their subvocalization.

As a quantitative approximation for the duration of time that subjects focus on each method's documentation, I defined the entire period during which the *JavaDoc* of a method is visible in the *JavaDoc* hover as a "reading" of this method. This is possible since the hover mechanism only allows one *JavaDoc* to be presented at a time. I accommodate situations where subjects are reading *JavaDocs* on the web by arbitrarily determining which of the visible methods (if more than one) is being read. This determination is based on the location, the mouse pointer, and the subjects' speech and actions. A similar policy is taken in the rare cases where *JavaDocs* are inspected by examining the method declaration in the source code.

The problem with the above metric is that it is too conservative: it will include durations during which the *JavaDoc* was open but the subject was not actually reading it. For example, the subject may have finished reading the *JavaDoc* and is contemplating its implications, he could be interacting with the experimenter while the *JavaDoc* is open, or he might have been looking at the source code while keeping the last *JavaDoc* open. Nevertheless, I believe that this approximation is still meaningful since my impression during the study was that subjects were reading the *JavaDocs* when they were visible. They often traced sentences vocally, with their mouse, or with their fingers close to the screen. After finishing reading the text for the first time, they sometimes kept it open as reinforcement and read previous sentences again.

For each debugging task performed by a subject, I created a transcript of the subject's "reading actions" - times at which the *JavaDoc* was open. These measurements were taken at an accuracy of tenths of a second. However, due to limitations of the screen recording software and the way in which *Eclipse* renders *JavaDocs*, my measurement of when the hover becomes open and when it closes may be inaccurate by up to 3/10 of a second. This is not a problem since most actual readings are longer than a full second, making the impact of such inaccuracies minimal. The transcript also included data on other actions, such as scrolling, editing, and speaking. These are used to supplement detailed analysis in specific cases, but most of the analysis will focus on the subject's reading actions.

Next, I aggregated the data of each transcript by counting the number of readings of the same target, and adding up the total duration of these events. I adjusted my count so that two subsequent reads of the same target with a short pause between them and no other reads would be counted as one visit. This adjustment helped address the frequent situations where the hover was accidentally closed and had to be re-opened, for example when the mouse slipped beyond a certain threshold or when the users attempted to turn the hover into a floating window.

Task Name Condition, Outcome									
	S?	S?	S?	S?	S?	S?	Avg	Avg>1	Ct.>1
F1							####	####	??
F2							####	####	??
F3							####	####	??
Other							####	####	??
Read time							####		
Work time							####		
Read/work							####		
Ct. reads							####		
Avg read							####		

Figure 6.4: Template for timing data tables

Finally, I aggregated the data of all subjects performing a task in the same condition. These results will be presented in tables based on the template of Fig. 6.4. First, the table presents the total time spent by each subject on each function. Next, it presents two types of averages for each function: an average that includes all methods (treating unvisited methods as 0), and an average that only includes methods visited for a total of at least 1 second. The first value gives us a general idea about the amount of time spent on the method, but it can be greatly affected if few subjects read the method. The second average addresses this problem, but may be less meaningful when the method is read by few individuals. I chose to require a minimal total duration to account for accidental hovers: many subjects occasionally opened the hover by moving or leaving the mouse over a call or variable. They immediately moved the mouse away when the hover appeared. Therefore, if the sum of all such visits adds up to less than a second, we treat this target as if it had never been read.

In creating the transcript for each task, I included every use of the *Eclipse* hover mechanism, even when it was used over classes and variables. In the former case, *Eclipse* presents the *JavaDoc* of the target. In the latter case, *Eclipse* merely displays the type of the object, unless the program is stopped in the debugger, in which case it will provide the current value. In the aggregate tables presenting the results, all these will be considered as “others”.

All the reading data was summed into a total “reading time” that is presented along with the time the subject spent on the task and the ratio. I also counted the total number of read actions, and the average read length.

In every debugging task, I also aggregated task-specific data, such as the duration during which certain code or documentation fragments were visible.

Finally, note that due to crashes of the the recording software and the operating system, the recordings for several sessions were irreparably corrupted. This crashes were eventually traced to faulty memory. I therefore use the outcome recorded for these tasks, but they are not included in our timing analysis.

6.4 First debugging task, based on the JMS API

The first task in our study was designed to investigate the choices developers make about which *JavaDocs* to explore (read), rather than how they read each one. It presents subjects with a small code fragment whose last statement causes execution to hang, and requires subjects to fix the problem. The cause and solution for the hang are conveyed as a directive by a seemingly straightforward method that is invoked early in the fragment.

6.4.1 The JMS API

The first two debugging tasks are based on code and directives from the *Java Messaging Service (JMS)* API. This API is used to reliably send messages between processes in a synchronous or asynchronous manner. JMS supports point-to-point messaging via queues, and publish-subscribe messaging via topics. Processes communicate via a JMS broker process which can reside on any machine and is responsible for physically managing these data structures. In other words, senders and receivers do not have to discover each other or communicate directly, but rather have to contact the same broker and use abstracted medium identities such as queue or topic identities.

The JMS interfaces were initially published as standalone libraries, but are now published as part of the *J2EE* standard, and form the basis for more sophisticated *J2EE* technologies. Different vendors provide open-source and commercial implementations for JMS; in this study, I used *Apache ActiveMQ*. Note though our study is based on the first version of JMS, the documentation of relevant methods is the same even in the current *J2EE* version.

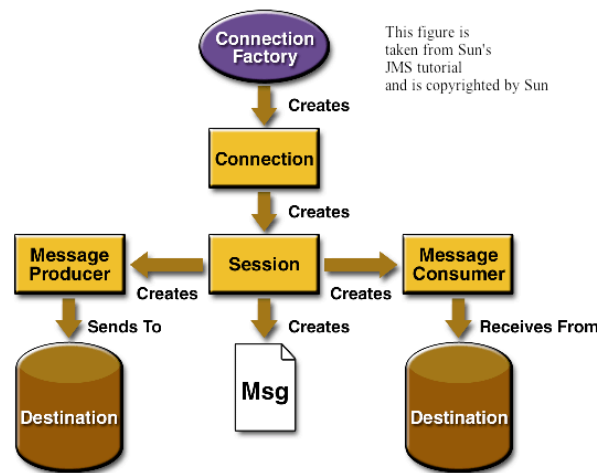


Figure 6.5: Sun's illustration of the JMS programming model

Clients usually follow a standard sequence of actions to communicate via JMS, as illustrated by Sun's diagram of the JMS programming model in Fig. 6.5. First, they instantiate a vendor-specific implementation of the JMS connection factory, and typically refer it to a specific broker. The factory object is then used to generate a connection. A connection can be used to generate multiple sessions, and each session object can be used to generate objects representing queues and topics, though these structures are physically maintained on the broker. The session object can also be used to create message producers or consumers that are bound to a specific queue or topic. The client can then create message objects, and send them via the producer; the consumers can receive messages and generate objects.

All subjects are given an overview booklet on the JMS API, which is reproduced in Appendix A.

6.4.2 The directive for this task

The first directive I chose for our study involves the peer-to-peer communications mechanism via queues. It is found in the `QueueConnectionFactory` interface, and specifically in its `createQueueConnection` method. The *JavaDoc* for this method, depicted in Fig. 6.6, states that the connection is created in a stopped mode, and that no messages will be delivered until the `start` method is invoked.

This kind of directive can be considered a protocol, since it clearly describes a sequence of invoca-

● **QueueConnection javax.jms.QueueConnectionFactory.createQueueConnection() throws JMSEException**

Creates a queue connection with the default user identity. The connection is created in stopped mode. No messages will be delivered until the `Connection.start` method is explicitly called.

Returns:
a newly created queue connection

Throws:
[JMSEException](#) – if the JMS provider fails to create the queue connection due to some internal error.
[JMSSecurityException](#) – if client authentication fails due to an invalid user name or password.

Figure 6.6: Documentation of `QueueConnectionFactory.createQueueConnection()` in JMS

tions. However, it refers to future errors but does not actually demand that the connection ever has to be started; after all, the process may send messages rather than receive them. A better classification of this directive may be a *return-value directive*. It serves an informational role, warning callers that the returned object may be considered incomplete or unfinished for their purposes.

Note that this directive meets the criteria I have set for the study as it is straightforward and unambiguous. It should also be relatively easy to find within the text of the documentation, as long as the subject reads beyond the first summary sentence.

The main reason I have chosen this directive, however, is that it appears in a method which I believe is not likely to be investigated by users. By its name and declaring class, I expect that developers will expect this to be a straightforward factory method for creating queue connection. They would therefore not expect to see additional directives or requirements placed here.

6.4.3 Codebase and task

The first task in our study was designed to investigate the choices developers make on which methods to explore. It focuses on exercising the decoration mechanism of *eMoose* and its potential ability to attract readers to methods which might not be investigated otherwise. Our choice of core directive, which appears in a relatively short *JavaDoc*, minimizes the importance of the augmented *JavaDoc* hover. While distractions by other decorated methods are possible, the relatively limited code size for this task should limit the impact of such distractions.

All subjects are first shown the `SenderToQueue` class, without *eMoose* annotations. This class consists of only a `main` method. The first part of its listing, depicted in Fig. 6.7, declares a variety of local variables. It makes sure that one or two parameters have been sent, and obtains a queue name and the number of messages to be sent.

Its second part, depicted in Fig. 6.8, begins with a `try` block responsible for initializing a queue. Specifically, a factory for queue connections is created with the hardcoded details of the broker. The factory is then used to create a connection, which is then used to create a session, which in turn is used to create the queue. Any exceptions will close the connection and quit from the program.

The next `try` block constitutes the heart of the test. The session and queue are used to manufacture a sender object and a text message object. A loop then uses the message object to send multiple distinct text messages through the sender object. Once this is done, an empty message object is created, and the queue is closed. Again, exceptions will close everything and terminate the program.

Subjects are next shown the `SynchQueueReceiver` class, which also consists of a single `main` function. The upper half of this class, depicted in Fig. 6.9, starts in a similar manner to `SenderToQueue`: a variety of objects are declared and the command parameters are parsed to identify a queue name (but not a number of messages).

```

1 package edu.cmu.contextualstudy.jms.jmssamples;
2 /**
3  * @(#)SenderToQueue.java 1.2 00/08/18
4  *
5  * Copyright (c) 2000 Sun Microsystems, Inc. All Rights Reserved.
6  *
7  * [ rest of license omitted to conserve space ]
8  */
9
10 import javax.jms.*;
11
12 import org.apache.activemq.ActiveMQConnectionFactory;
13
14 /**
15  * The SenderToQueue class consists only of a main method, which sends
16  * several messages to a queue.
17  * <p>
18  * Run this program in conjunction with either SynchQueueReceiver or
19  * AsynchQueueReceiver. Specify a queue name on the command line when you run
20  * the program. By default, the program sends one message. Specify a number
21  * after the queue name to send that number of messages.
22  *
23  * @author Kim Haase
24  * @version 1.2, 00/18/00
25  */
26 public class SenderToQueue {
27
28     /**
29      * Main method.
30      *
31      * @param args the queue used by the example and, optionally, the
32      *             number of messages to send
33      */
34     public static void main(String[] args) {
35         String queueName = null;
36         QueueConnectionFactory queueConnectionFactory = null;
37         QueueConnection queueConnection = null;
38         QueueSession queueSession = null;
39         Queue queue = null;
40         QueueSender queueSender = null;
41         TextMessage message = null;
42         final int NUM_MSGS;
43         final String MSG_TEXT = new String("Here is a message");
44         int exitResult = 0;
45
46         if ( (args.length < 1) || (args.length > 2) ) {
47             System.out.println("Usage: java SenderToQueue <queue_name> [<number_of_messages>"];
48             SampleUtilities.exit(1);
49         }
50         queueName = new String(args[0]);
51         System.out.println("Queue name is " + queueName);
52         if (args.length == 2){
53             NUM_MSGS = (new Integer(args[1])).intValue();
54         } else {
55             NUM_MSGS = 1;
56         }
57

```

Figure 6.7: Source code listings for SenderToQueue.java - upper half


```

57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
try {
    queueConnectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");
    queueConnection = queueConnectionFactory.createQueueConnection();
    queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
    queue = queueSession.createQueue(queueName);
} catch (Exception e) {
    System.out.println("Connection problem: " + e.toString());
    if (queueConnection != null) {
        try {
            queueConnection.close();
        } catch (JMSEException ee) {}
    }
    SampleUtilities.exit(1);
}

/*
 * Create sender.
 * Create text message.
 * Send five messages, varying text slightly.
 * Send end-of-messages message.
 * Finally, close connection.
 */
try {
    queueSender = queueSession.createSender(queue);
    message = queueSession.createTextMessage();

    for (int i = 0; i < NUM_MSGS; i++) {
        message.setText(MSG_TEXT + " " + (i + 1));
        System.out.println("Sending message: " + message.getText());
        queueSender.send(message);
    }

    // Send a non-text control message indicating end of messages.
    queueSender.send(queueSession.createMessage());
} catch (JMSEException e) {
    System.out.println("Exception occurred: " + e.toString());
    exitResult = 1;
} finally {
    if (queueConnection != null) {
        try {
            queueConnection.close();
        } catch (JMSEException e) {
            exitResult = 1;
        }
    }
}

SampleUtilities.exit(exitResult);
}

```

Figure 6.8: Source code listings for SenderToQueue.java - lower half

```

1 package edu.cmu.contextualstudy.jms.jmssamples;
2 /**
3  * @(#)SynchQueueReceiver.java 1.7 00/08/14
4  *
5  * Copyright (c) 2000 Sun Microsystems, Inc. All Rights Reserved.
6  *
7  * [Rest of license removed to conserve space]
8  */
9 import javax.jms.*;
10
11 import org.apache.activemq.ActiveMQConnectionFactory;
12
13 /**
14  * The SynchQueueReceiver class consists only of a main method, which fetches
15  * one or more messages from a queue using synchronous message delivery. Run
16  * this program in conjunction with SenderToQueue. Specify a queue name on the
17  * command line when you run the program.
18  * <p>
19  * The program calls methods in the SampleUtilities class.
20  *
21  * @author Kim Haase
22  * @version 1.7, 08/14/00
23  */
24 public class SynchQueueReceiver {
25
26     /**
27      * Main method.
28      *
29      * @param args the queue used by the example
30      */
31     public static void main(String[] args) {
32         String queueName = null;
33         QueueConnectionFactory queueConnectionFactory = null;
34         QueueConnection queueConnection = null;
35         QueueSession queueSession = null;
36         Queue queue = null;
37         QueueReceiver queueReceiver = null;
38         TextMessage message = null;
39         int exitResult = 0;
40
41         /*
42          * Read queue name from command line and display it.
43          */
44         if (args.length != 1) {
45             System.out.println("Usage: java SynchQueueReceiver <queue_name>");
46             SampleUtilities.exit(1);
47         }
48         queueName = new String(args[0]);
49         System.out.println("Queue name is " + queueName);
50

```

Figure 6.9: Source code listings for SynchQueueReceiver.java - upper half

The lower half of this class is depicted in Fig. 6.10, with the *eMoose* annotations that subjects in the experimental condition would see only once the timer is started. This part starts in a similar manner, with a `try` block that initializes a queue in an identical manner to that of `SenderToQueue`. The second `try` block, however, is different: instead of creating a sender object, a receiver is created. Then, a `while` loop attempts to receive a message using the receiver's `receive` method. If the received object is a text message, the result is printed, and the next iteration begins. Otherwise, the message indicates the end of the sequence, and the loop breaks and the program ends. It is important to note that in the original code, there was a call between lines 81 and 82, immediately after the creation of the receiver, which invoked the `start` method on the `queueConnection` object. As explained below, that call was deleted in the version seen by the users.

```

51      /*
52      * Obtain connection factory.
53      * Create connection.
54      * Create session from connection; false means session is not
55      * transacted.
56      * Obtain queue name.
57      */
58      try {
59          queueConnectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");
60          queueConnection = queueConnectionFactory.createQueueConnection();
61          queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
62          queue = queueSession.createQueue(queueName);
63      } catch (Exception e) {
64          System.out.println("Connection problem: " + e.toString());
65          if (queueConnection != null) {
66              try {
67                  queueConnection.close();
68              } catch (JMSException ee) {}
69          }
70          SampleUtilities.exit(1);
71      }
72
73      /*
74      * Create receiver, then start message delivery.
75      * Receive all text messages from queue until
76      * a non-text message is received indicating end of
77      * message stream.
78      * Close connection and exit.
79      */
80      try {
81          queueReceiver = queueSession.createReceiver(queue);
82          while (true) {
83              Message m = queueReceiver.receive();
84              if (m instanceof TextMessage) {
85                  message = (TextMessage) m;
86                  System.out.println("Reading message: " + message.getText());
87              } else if (m!=null)
88              {
89                  // Non-text control message indicates end of messages.
90                  break;
91              }
92          }
93      } catch (JMSException e) {
94          System.out.println("Exception occurred: " + e.toString());
95          exitResult = 1;
96      } finally {
97          if (queueConnection != null) {
98              try {
99                  queueConnection.close();
100             } catch (JMSException e) {
101                 exitResult = 1;
102             }
103         }
104     }
105     SampleUtilities.exit(exitResult);
106 }

```

Figure 6.10: Source code listings for `SynchQueueReceiver.java` - lower half with *eMoose* annotations

After subjects are shown a quick skim of both programs, they are shown a *JUnit* unit test and receive an explanation that it will run both programs together and indicate in green or red the results of execution. The experimenter starts the test, and shows that it does not terminate for a while, at which point the test times out and a failure indication appears. The experimenter shows them the console window, where the output of both programs shows that they are using the same queue. The remaining output then belongs to the sender program, which indeed sends a sequence of messages as expected. Subjects are assured that the sender works correctly, and that all messages are now waiting on the broker process.

Next, subjects are shown again the receiving program, and told to imagine that they have already debugged it and found that the first call to `receive` in line 83 causes the program to hang indefinitely. They are told that their role is to find and fix the problem, and that all changes would have to take place in

the receiver program. They are also assured that the problem occurs in all concrete JMS implementations and that the problem has to be fixed at the level of the receiver program rather than inside specific implementations.

I created this bug by deleting the call to `start` on `createQueueConnection`. Recall that I selected a directive stating that messages will not be delivered until the connection is explicitly started. As a result, the call to `receive` blocks on first entry. If the program was multithreaded, another thread could have started the connection at some point.

Interestingly, however, the *JavaDocs* for the `receive` method, depicted in Fig. 6.11, do not warn of this possibility. Rather, they are somewhat confusing. They state that the call will block until messages are produced or the connection is closed. The former is not applicable here since there are already messages on the broker waiting for delivery, and the latter is not applicable since the program is single-threaded (though the means to close the connection may not be clear to our subjects). Thus, the documentation of this method is actually misleading, as subjects are expected to eliminate these two causes and accept the fact that there must be a third, undocumented, reason. After being occupied by this for a while, they should eventually give up and begin exploring earlier statements as the potential source of failure.

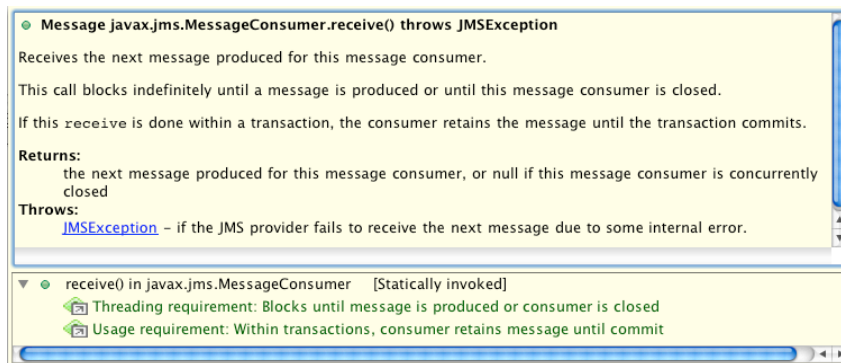


Figure 6.11: JavaDocs for the `MessageConsumer.receive()` method with *eMoose* directives

Note that within the *Eclipse* IDE as seen by virtually all subjects, the two `try` blocks never appear on the screen at the same time,. Thus, subjects focused on the `receive` call would not see the upper block until they start searching elsewhere or start suspecting that the problem actually lies in the initialization code. To be able to compare the time during which subjects have an opportunity to be exposed to our directive, I will measure the duration of time spent with the first block visible. I call this block our “core area”. Nevertheless, with so few statements up to and including the call to `receive`, I expect all subjects to at least be exposed to the initialization code.

The few other calls in this code are likely to pose some distractions to the users. The call to `createQueueSession`, which is not decorated in the experimental group, takes two parameters. However, as can be seen in Fig. 6.12, its documentation is fairly straightforward and the parameters themselves involve acknowledgements in transacted mode. Since messages are not received, that is likely not related.

The documentation for the decorated call to `createQueue` on the `queueSession` object, on the other hand, presents many details and directives. As can be seen in 6.13. it first states that the facility of creating queue identities is “meant for rare cases where clients need to manipulate queue identity”. The reason behind this obscure description has to do with the typical use of JMS, where specific destinations are requested via *JNDI* lookups. Nevertheless, the examples themselves and the API are meant exactly for this not-so-rare use. The fact that the client is not portable is concerning, but of little relevance here. Finally, the documentation states that the method is not meant for creating a physical queue.

```

● QueueSession javax.jms.QueueConnection.createQueueSession(boolean transacted, int acknowledgeMode) throws JMSException

Creates a QueueSession object.

Parameters:
    transacted indicates whether the session is transacted
    acknowledgeMode indicates whether the consumer or the client will acknowledge any messages it receives;
    ignored if the session is transacted. Legal values are Session.AUTO_ACKNOWLEDGE,
    Session.CLIENT_ACKNOWLEDGE, and Session.DUPS_OK_ACKNOWLEDGE.

Returns:
    a newly created queue session

Throws:
    JMSException - if the QueueConnection object fails to create a session due to some internal error or lack of
    support for the specific transaction and acknowledgement mode.

```

Figure 6.12: JavaDocs for the QueueConnection.createQueueSession() method

This is consistent with how JMS works, as the actual messages are stored on the broker process. The queue created in sending and receiving processes is merely a representation for the broker-based process. Nevertheless, this may confuse some of the subjects, especially those in the experimental condition.

```

● Queue javax.jms.QueueSession.createQueue(String queueName) throws JMSException

Creates a queue identity given a Queue name.

This facility is provided for the rare cases where clients need to dynamically manipulate queue identity. It allows the creation of a queue identity with a provider-specific name. Clients that depend on this ability are not portable.

Note that this method is not for creating the physical queue. The physical creation of queues is an administrative task and is not to be initiated by the JMS API. The one exception is the creation of temporary queues, which is accomplished with the createTemporaryQueue method.

Specified by: createQueue(...) in Session
Parameters:
    queueName the name of this Queue
Returns:
    a Queue with the given name
Throws:
    JMSException - if the session fails to create a queue due to some internal error.

▼ createQueue(String) in javax.jms.QueueSession [Statically invoked]
    Restriction: Provided for rare that clients dynamically manipulate queue identity. (13 Jul 20:46:16) [SELECTION]
    Limitation: Does not create physical queue, which is not initiated by JMS API. (13 Jul 20:46:17) [SELECTION]
▼ createQueue(String) in javax.jms.Session [Implemented]
    Restriction: Facility provided for rare cases when clients need to dynamically manipulate queue identity (13 Jul 20:48:02) [SELECTION]
    Usage requirement: Makes client not portable (13 Jul 20:48:03) [SELECTION]
    Limitation: Method does not create physical queues, which are not initiated by JMS API. (13 Jul 20:48:04) [SELECTION]

```

Figure 6.13: JavaDocs for the QueueSession.createQueue method with eMoose directives

Note that users in the experimental condition who examine createQueue will see a set of identical directives from the Session interface which QueueSession extends. For some unclear reason, SUN chose to present the method in both superinterface and subinterface, and to repeat the documentation (with a minor adjustment in the use of a @since tag rather than have the documentation inherited. Since the method and documentation are redefined, eMoose presents the directives in both versions (though the portability issue was accidentally not included in one of them). The duplication was expected to slightly perplex some users.

The call to createReceiver on queueSession, which is not decorated, presents very few details, as can be seen in Fig. 6.14.

```

● QueueReceiver javax.jms.QueueSession.createReceiver(Queue queue) throws JMSException

Creates a QueueReceiver object to receive messages from the specified queue.

Parameters:
    queue the Queue to access

Throws:
    JMSException - if the session fails to create a receiver due to some internal error.
    InvalidDestinationException - if an invalid queue is specified.

```

Figure 6.14: JavaDocs for the QueueSession.createReceiver method with eMoose directives

Finally, note that the code of the receiver after our change could have plausibly been written by

a developer who was exposed only to the sender program. After copying the first try block (with the initialization sequence) verbatim, that author could then create a receiver instead of a sender and begin receiving messages. Since the code for the queue initialization was copied, the author would not be likely to have read the directive on the call to `createQueueConnection`.

6.4.4 Results - Success rates

Success in this task required subjects to identify the need to call `start` to ensure that messages are delivered, and to do so in the code. Of 13 subjects who performed this task in the control condition, only 4 were successful and 9 were not. On the other hand, of the subjects who performed this task in the experimental condition, 10 were successful while 3 were not.

Since eMoose was expected to help performance, I used a one-tailed test to determine statistical significance. Given the small sample size, I used Fisher's exact test to test the independence of eMoose use and success, rejecting the null hypothesis in each case. For this task, the difference in success rate was statistically significant ($p = 0.024$). However, we must examine the behaviors in details to understand the reasons for this difference.

In the detailed narrative below, we refer to certain call targets by mnemonics. For ease of reference, a simplified version of our program is presented with these mnemonics in Fig. 6.15 below. The mnemonics are: FAC for the `ActiveMQConnectionFactory` constructor, CQUC for `createQueueConnection`, CQUS for `createQueueSession`, CQUE for `createQueue`, ST for `start`, CREC for `createReceiver`, and RECV for `receive`.

```

73 /*
74  * Obtain connection factory.
75  * Create connection.
76  * Create session from connection; false means session is not
77  * transacted.
78  * Obtain queue name.
79  */
80 try {
81     queueConnectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");
82     queueConnection = queueConnectionFactory.createQueueConnection();
83     queueSession = queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
84     queue = queueSession.createQueue(queueName);
85     queueConnection.start();
86 } catch (Exception e) {
87     [Error handling code, then exit]
88 }
89
90
91
92
93
94 }
95
96 /*
97  * Create receiver, then start message delivery.
98  * Receive all text messages from queue until
99  * a non-text message is received indicating end of
100 * message stream.
101 * Close connection and exit.
102 */
103 try {
104     queueReceiver = queueSession.createReceiver(queue);
105     while (true) {
106         Message m = queueReceiver.receive();
107         if (m instanceof TextMessage) {
108             [Long message handling code, never reached due to hang at RECV]
109         }
110     }
111 }

```

Figure 6.15: Method mnemonics in the codebase for Task 1

6.4.5 Results - Successful subjects in the control condition

Table. 6.2 presents the timing data for the four successful subjects in the control condition of this task. Unfortunately, the recording for subject S17 was corrupted, leaving us with data for only three subjects.

Before we proceed, a brief explanation of the structure of this table. Its first part refers to the time spent reading the documentation of each target method, using the mnemonics presented in Fig. 6.15. The second part presents the work and reading time.

The third part is unique to this task, and presents four rows: **Time in core** - The total time that the subject has spent with the core area visible. **Ratio in core** - The proportion of time (from the overall work time) that the subject has spent with the core area visible. **First visit core** - The first time in the study that the core area became visible. **First CQUC visit** - The first time that the subject has opened the documentation of the CQUC method.

Task 1, Control Condition, Successful Subjects							
	S6	S12	S17	S20	Avg	Avg>1	Ct. >1
FAC	0.0	0.0	V i d e o c o r r u p t e d	0.0	0.0	0.0	0 of 3
CQUC	10.3	2.9		52.5	21.9	21.9	3 of 3
CQUS	0.3	1.5		23.8	8.5	12.7	1 of 3
CQUE	15.0	0.0		75.4	30.1	45.2	2 of 3
CREC	5.7	0.0		22.0	9.2	13.9	2 of 3
RECV	0.2	0.0		21.3	7.2	21.3	1 of 3
Other	4.9	0.6		2.0	2.5	3.5	2 of 3
Read time	36.40	5.00		197.00	119.20		
Work time	194.20	96.00		460.00	375.10		
Read/Work	18.7%	5.2%		42.8%	31.8%		
Ct. Reads	12	3	18	11.0			
Avg read	3.0	1.7	10.9	5.2			
Time in core	104.6	57.7	270.3	144.2			
Ratio in core	53.9%	60.1%	58.8%	38.4%			
First visit core	5.0	8.0	151.0	54.7			
First CQUC visit	168.0	86.0	314.0	189.3			

Table 6.2: Timing data for successful controls in control condition of Task 1

Visits to CQUC

The three successful subjects for which we have data explored CQUC relatively early in the allocated 15 minute time window: S6 within 3 minutes, subject S12 in about 1.5 minutes, and subject S20 in a little over 5 minutes.

I examined the recordings to see what each subject did in this visit. Subject S6 took some time to absorb the directive and fix the problem, while subject S12 immediately spotted the directive and fixed the problem. Subject S20, however, read the *JavaDocs* of CQUC for a very long time, and then indicated that he is unclear about how JMS works in terms of where the queue is actually located. After receiving clarifications about how JMS operates, the user eventually announced the issue.

Actions before arrival at CQUC

Finding CQUC helped the subjects identify the problem, but what “wrong” choices had they followed first? According to the detailed logs, subject S6 initially swept the entire file, then spent some time examining the code in the receiving area, until migrating higher. Interestingly, he never examined RECV, and

spent very little time on CREC. Subject S12 worked extremely fast, and after a very brief visit to CQUS immediately visited CQUC. Subject S20 started with CREC, then spent about 20 seconds on RECV, and then spent significant time on CREC and CQUE until he reached CQUC. Thus, we have one subject who was instantaneously successful, and two others who followed a path through other methods before exploring CQUC.

An interesting fact about all successful subjects in the control condition is that they spent very little time exploring anything in the “other” category, such as variables, classes, and unrelated calls.

6.4.6 Results - Unsuccessful subjects in the control condition

The timing data for the 9 unsuccessful subjects in the control condition is presented in Table 6.3.

Task 1, Control Condition, Unsuccessful Subjects												
	S1	S3	S9	S10	S13	S15	S21	S23	S26	Avg	Avg>1	Ct.>1
FAC	3.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.4	3.6	1 of 9
CQUC	9.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	9.3	1 of 9
CQUS	38.9	137.2	0.0	0.0	0.0	13.0	0.0	0.0	0.0	21.0	63.0	3 of 9
CQUE	0.0	7.9	0.0	23.5	47.5	26.9	0.0	0.0	38.3	16.0	28.8	4 of 9
CREC	17.4	22.0	37.8	12.2	136.9	22.7	0.0	0.0	11.8	29.0	37.3	7 of 9
RECV	202.4	120.3	80.0	91.2	121.4	32.5	58.7	62.1	91.3	95.5	95.5	9 of 9
Other	90.4	110.6	107.7	106.8	4.8	158.3	9.7	37.6	22.0	72.0	72.0	9 of 9
Read time	362.0	398.0	225.5	233.7	310.6	253.4	68.4	99.7	163.4	235.0		
Work time	909.0	1044.0	900.0	900.0	909.0	930.0	900.0	920.0	900.0	923.6		
Ratio in reading	39.8%	38.1%	25.1%	26.0%	34.2%	27.2%	7.6%	10.8%	18.2%	25.2%		
Count of reads	27	27	24	15	17	22	9	17	10	18.7		
Mean read time	13.4	14.7	9.4	15.6	18.3	11.5	7.6	5.9	16.3	12.5		
Time in core area	338.0	443.0	16.5	101.5	93.3	180.4	164.0	134.3	88.0	173.2		
Ratio in core area	37.2%	42.4%	1.8%	11.3%	10.3%	19.4%	18.2%	14.6%	9.8%	18.8%		
First visit to core	174.0	96.0	390.0	569.0	705.0	136.0	22.0	45.0	60.0	244.1		
First CQUC visit	703.0	NA	NA	NA	NA	NA	NA	NA	NA	NA		

Table 6.3: Timing data for unsuccessful controls in control condition of Task 1

Where time was spent

Unsuccessful subjects spent an average of 4 minutes reading *JavaDocs*, while their successful peers spent only 2. However, they appeared to spend this extra time in the wrong locations. All of them read the RECV method, spending an average of 95 seconds, whereas this method was barely explored by their successful peers. Most of them also explored the other method in the receiving area, CREC, for an average of 37 seconds, while successful subjects only spent 14 seconds there.

Since each unsuccessful subject spent 15 minutes on the task, it is somewhat surprising that they spent only a fifth of their time, on average, with the core area. Nevertheless, these 173 seconds are actually more than the 144 spent by successful subjects. It is therefore striking that 3 of the 9 unsuccessful subjects did not explore any methods in the core area. Most importantly, of the other 6, only one explored CQUC, though he did so 10 minutes into the session, long after his first visit to the core area. In other words, even given ample time in the core area, all but one of the unsuccessful subjects did not explore CQUC. Among all controls in general, only 5 of 13 explored *CQUC*.

Why subject S1 did not find the directive in CQUC

While it is not surprising that the 8 controls who did not examine CQUC did not succeed in the task, it is peculiar that subject S1, who did examine it, was not successful. In the recording transcripts, the subject

did not use the mouse pointer or selections while the *JavaDoc* for CQUC was open, so it is not clear whether he read the directive. However, soon after reading CQUC he was starting to wonder if there is an exception thrown, and received an assurance that none were, after which the subject explored other calls and did not return to CQUC. It is possible that the presence of the `throws` block attracted the subject's attention.

Proportion of subjects examining calls

Interestingly, the portion of unsuccessful subjects who explored a particular call tended to decrease with distance from RECV. All 9 examined RECV, 7 examined CREC, 4 examined CQUE, 3 examined CQUS, and only 1 explored CQUC and FAC. In fact, for all these subjects with the exception of S1, it holds that if a call was explored so were all the calls that were closer to RECV. However, an examination of the actual transcripts shows that despite this pattern, subjects did not necessarily examine calls in a reverse order, and that sometimes they returned to calls that they have previously explored after examining ones farther away from RECV.

Reading the “other” category

Before we proceed, the issue of the “other” category must be addressed. The timing data shows that most unsuccessful subjects spent an average of more than a minute on items in this category, compared to only a few seconds among successful ones. Time spent in this category is aggregated from a large variety of items on which subjects could hover to reveal a *JavaDoc* window. These items generally fall into four groups:

- Classes, for which the class-level *JavaDoc* is displayed.
- Alternative versions for some operations (`receiveNoWait` instead of `receive`), which the subjects typically examine in the web-based version or via the auto-completion mechanism.
- Variables and objects, for which all they received is an indication of type.
- Classes, objects, and methods that are not relevant to the task, such as standard library classes (e.g., `String`), or that appear in areas that are not relevant (e.g., within exception blocks, after the blocked call).

While an exploration of the first two types of items makes sense as subjects are trying to gather an understanding of the problem or find a solution, the next two are perplexing. Note that a similar behavior will appear in all other tasks. As a result, we will ignore this category in the subsequent tasks and then discuss this issue in general in Sec. 6.10.

6.4.7 Results - All subjects in the experimental condition

We now turn to the experimental condition. The timing results for the 10 successful subjects is presented in Table 6.4, while the data for the 3 unsuccessful ones appears in Table 6.5. Surprisingly, beyond the expected differences in work time, there are no obvious differences between the two conditions, although the small sample size for unsuccessful subjects makes comparisons difficult.

We see that all subjects in the experimental condition explored CQUC. The time spent on it, however, varied greatly between subjects. 6 subjects spent 10 seconds or less on CQUC, while 7 spent significantly more. Interestingly, two of those who actually spent more time did not fix the problem, suggesting that

Task 1, Experimental Condition, Successful Subjects													
	S2	S5	S7	S14	S16	S18	S19	S22	S24	S25	Avg	Avg>1	Ct.>1
FAC	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0 of 0
CQUC	9.5	47.5	10.2	4.3	30.6	34.7	57.2	8.6	1.5	64.5	26.9	26.9	10 of 10
CQUS	32.9	0.0	0.0	0.0	28.6	65.8	0.4	0.0	7.2	0.0	13.5	33.6	4 of 10
CQUE	3.3	0.0	22.9	26.9	43.5	78.0	60.4	3.0	8.0	88.6	33.5	37.2	9 of 10
CREC	16.2	4.7	0.3	0.0	0.0	28.4	0.0	0.0	15.5	8.4	7.4	14.6	5 of 10
RECV	206.7	234.8	138.9	219.0	131.5	83.7	128.0	175.1	5.1	19.6	134.2	134.2	10 of 10
Other	1.1	52.7	30.6	18.7	54.6	29.0	0.0	0.0	111.8	25.6	32.4	40.5	8 of 10
Read time	269.7	339.7	202.9	268.9	288.8	319.6	246.0	186.7	149.1	206.7	247.8		
Work time	763.9	653.0	604.0	758.0	624.3	834.0	476.0	842.0	640.0	615.0	681.0		
Ratio in reading	35.3%	52.0%	33.6%	35.5%	46.3%	38.3%	51.7%	22.2%	23.3%	33.6%	37.2%		
Count of reads	22	25	25	28	18	24	15	14	25	25	22.1		
Mean read time	12.3	13.6	8.1	9.6	16.0	13.3	16.4	13.3	6.0	8.3	11.7		
Time in core area	201.0	220.0	123.0	94.2	217.9	481.6	167.6	80.1	107.7	268.5	196.2		
Ratio in core area	26.3%	33.7%	20.4%	12.4%	34.9%	57.7%	35.2%	9.5%	16.8%	43.7%	28.8%		
First visit to core	20.0	333.0	10.0	4.0	23.0	85.0	69.0	286.0	180.0	169.0	117.9		
First CQUC visit	747.0	580.0	490.0	720.0	23.0	805.0	387.0	834.0	632.0	476.0	569.4		

Table 6.4: Timing data for successful subjects in experimental condition of Task 1

Task 1, EXP Condition, Unsuccessful Subjects						
	S4	S8	S11	Avg	Avg>1	Ct.>1
FAC	2.5	0.0	0.8	1.1	2.5	1 of 3
CQUC	17.3	36.5	5.3	19.7	19.7	3 of 3
CQUS	42.7	6.0	24.1	24.3	24.3	3 of 3
CQUE	7.3	53.7	52.5	37.8	37.8	3 of 3
CREC	0.0	2.7	15.5	6.1	9.1	2 of 3
RECV	60.7	322.2	257.8	213.6	213.6	3 of 3
Other	39.0	48.7	26.3	38.0	38.0	3 of 3
Read time	169.5	469.8	382.3	340.5		
Work time	923.0	900.0	940.0	921.0		
Ratio in reading	18.4%	52.2%	40.7%	37.0%		
Count of reads	24	28	29	27.0		
Mean read time	7.1	16.8	13.2	12.3		
Time in core area	536.0	144.0	273.5	317.8		
Ratio in core area	58.1%	16.0%	29.1%	34.5%		
First time in core	370.0	24.0	60.0	270.3		
First CQUC visit	543.0	41.0	584.0	389.3		

Table 6.5: Timing data for unsuccessful subjects in experimental condition of Task 1

the delay may be attributed to a misunderstanding of the directive or to something else that attracted the reader's attention. All the subjects in the experimental condition also explored RECV and all but one explored CQUE, so together with CQUC we can say that they explored all of the decorated methods.

Why some subjects did not find the directive

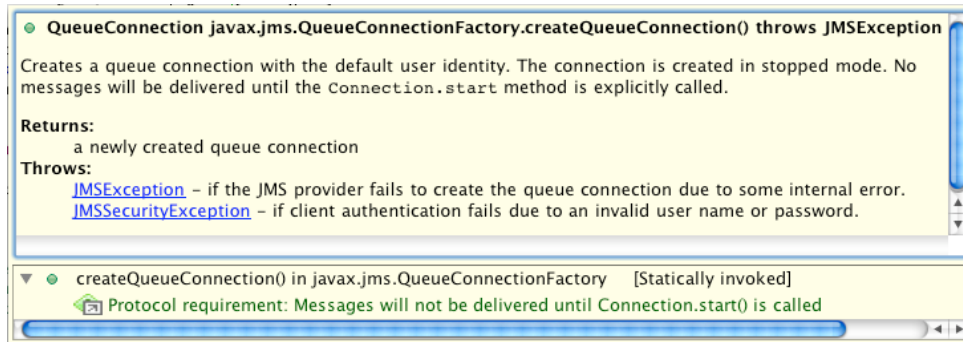


Figure 6.16: JavaDocs for the `QueueConnectionFactory.createQueueConnection()` method with *eMoose* directives

To investigate why three subjects failed to fix the problem despite being exposed to the directive, I reexamined the recordings. I found that these subjects seemed to ignore the lower pane. For reference, Fig. 6.16 presents the augmented hover for CQUC. Subject S4 viewed the *JavaDocs* of CQUC with the autocompletion mechanism, which is not augmented by *eMoose*. He was therefore not exposed to the explicit highlighting of the directive, although it is not clear why he did not grasp it from the text. Subject S8 read the text in the upper pane, and after the first statement skipped to the return clause and vocalized it; he focused on this line again in his two visits towards the end of the session. When asked afterwards about this behavior, he said that he did not look at the lower part at all. Subject S11 made two very brief visits as part of a systematic scan of methods in the core area, and appeared to focus on the upper pane.

Note that it is not clear whether the successful subjects identified the directive from the lower pane, which explicitly lists the directives, or from the full text in the upper pane. The entire window was visible at the same time, allowing users to see both. The second JMS task, discussed later, investigates this issue with a hover in which seeing the directive in the text would require scrolling.

6.4.8 Results - Contrasting results for the two conditions

Our attention in this study is primarily focused on the differences between the two conditions. To facilitate this discussion, Table 6.6 compares the values for specific rows from the previous table.

Treatment of CQUC

The most important difference between the conditions is that all 13 subjects in the experimental condition read CQUC, compared to only 5 in the control condition. This accounts for the significant difference in success rates between the groups. The statistically significant difference suggests that the *eMoose* decorations may have had a significant impact on directive awareness.

Examining the rest of the results related to CQUC, however, highlights several additional and puzzling differences.

	CTL-Success	CTL-Fail	EXP-Success	EXP-Fail	Total CTL	Total EXP
Subjects with data	3	9	10	3	12	13
Read/Work	31.80%	25.20%	37.20%	37.00%	26.85%	37.15%
Core/Work	38.40%	18.80%	28.80%	34.50%	23.70%	30.12%
AVG 1st core visit	54.00	244.00	117.00	270.00	196.50	152.31
CQUC Visitors	3/3	1/9	10/10	3/3	4/12	13/13
CQUC AVG>1	21.90	9.30	26.90	19.70	12.45	25.24
AVG 1st CQUC visit	189.00	703.00	569.00	389.00	574.50	527.46
RECV Visitors	1/3	9/9	10/10	3/3	10/12	13/13
RECV AVG >1	21.30	95.50	134.20	213.60	76.95	152.52

Table 6.6: Comparison of certain behaviors across both conditions

First, the first visit to CQUC by subjects in the experimental condition was on average much later than the first visit by those in the control condition who did visit it. Specifically, successful subjects in the experimental condition took on average nearly 9.5 minutes before their first visit, and unsuccessful subjects took 6 minutes. For successful controls, it took only 3 minutes, though the unsuccessful control who examined CQUC took 11. In other words, while the subjects in the experimental condition for which the method CQUC was decorated were more likely to eventually visit it, they did so much later than those who did happen to examine it.

Second, subjects in the experimental condition tended to explore CQUC significantly later after their first visit to the core area than those in the control condition who explored CQUC. Successful controls took 2 minutes, on average, after the first visit, while successful *eMoose* users took 7 minutes and unsuccessful ones took 2 minutes. Thus, even after the core was visible, CQUC was not immediately examined.

Third, the average time spent reading CQUC in the experimental group was actually higher than that in the control group, despite the availability of the lower pane.

Other methods in the core area

Subjects in the experimental condition spent more time (in absolute terms and relative to the total work time) in the core area than those in the control condition. Perhaps as a result, a greater portion of them explored each method in the core area. For example, the decorated method CQUE, was explored by 6 controls and 12 *eMoose* users, for a similar amount of time. Interestingly, the undecorated method CQUS was also investigated by 7 *eMoose* users vs. at least 3 controls.

Time reading RECV

A curious finding is that despite the increased time spent in the core area, almost all subjects in the experimental condition spent, on average, significantly more time than those in the control condition reading the decorated RECV method. Specifically, successful and unsuccessful subjects in the experimental condition spent 2 and 3 minutes respectively, while controls spent 20 seconds and 1.5 minutes respectively.

This finding is particularly interesting because it cannot directly be explained by the *eMoose* interventions and their potential for distractions. Since this method was known to be the point where the program hangs, subjects did not need to rely on the decorations to know that they may want to explore it. Once they opened the *JavaDoc* hover, the contents of the upper and lower pane were very similar as the text primary conveyed the directives.

6.4.9 Discussion - Explaining the difficulties of controls

The results for the control group demonstrate the reality and potential seriousness of the problem of directive awareness. Out of 13 subjects, 8 never explored the CQUC method despite spending a significant amount of time in the core area and in many cases reading some of its other methods. Understanding this inconsistency is critical, and while we lack “think aloud” data to determine the cause, the data can support several interpretations.

We saw that among unsuccessful control subjects, methods were less likely to be read as distance from the call to RECV increased. This is perhaps not surprising, as calls in the core area were never visible at the same time as those in the receiving area on which subjects initially focused. I suspect that had both parts been on the screen at the same time, earlier and more extensive attention would have been given to the core area and its methods. It would be interesting to repeat this experiment with a larger viewport and also with code that splits the blocks into two separate functions. Nevertheless, this separation does not explain why the call to CQUC was explored much later than the first visit to the core area and significantly less frequently than the calls in the two lines that follow it.

My interpretation of the results offers a tentative answer for the study’s research question, which aimed to determine why subjects in the control condition failed to identify the directive. It is inspired by recent applications [56] of *information foraging* [70] theory. That model suggests that information exploration decisions are based on *scents* that help identify targets relevant to goals and estimate the profitability of exploring them. I argue that decisions on method exploration are influenced by a “scent” given by the method’s likely role and by the apparent relation of its name to the developers’ goals. In this case, CQUC may have seemed like a trivial factory method that generated a remote ancestor of the object whose method causes the failure. This may have constituted a negative scent that made its exploration less attractive even as options were running out.

Regardless of the reasons for missing CQUC, these results carry implications for my study’s second research question, about the implications for documentation writers. I argue that with standard IDE support, authors must take into account the significant possibility that the documentation of their methods would not be read at all by users. My survey of APIs suggests that at present, many methods merely present a contract in the documentation and perform no runtime “sanity” checks. While this policy improves performance, it can also result in difficult-to-trace problems that defeat the goals of API providers and their clients. In the future, it is possible that design by contract with static analysis may offer a solution, but at present the specification and checking effort is not yet practical. API authors may therefore wish to err on the side of caution and include additional safety checks when possible, and avoid surprising users with caveats and side effects.

The limited exploration of method CQUC reinforces the need for means to signal the availability of important information to clients who may not otherwise explore the documentation of an invoked method. We now turn to examining the impact of *eMoose* in this task.

6.4.10 Discussion - Explaining the impact of *eMoose*

Impact of *eMoose* on exploration of CQUC

The most notable difference between the experimental and control conditions is that every *eMoose* user explored the decorated methods CQUC and RECV, with all but one also exploring the decorated CQUE. Thus, in this small fragment the call decorations had the expected effect of increasing the chances of finding the important directive. On the other hand, these findings also indicate a potential for disruptive distractions, as subjects explored decorated methods whose directives were not relevant to the problem.

This is wasteful and could have sent them down costly “dead-ends”.

It turns out, however, that though *eMoose* users explored more decorated methods, the tool did not seem to lead them to examine everything. In addition, closer scrutiny of the data and recordings shows that the *JavaDocs* for decorated calls were also not necessarily explored as soon as they were first visible. Subjects frequently did so only later, as their goal changed or when other options were less promising. This was evident with CQUC, which was initially ignored by most subjects. In many cases, subjects also explored undecorated calls before exploring decorated ones. In other words, *eMoose* does not seem to force users to examine everything, at least not immediately.

My interpretation of this behavior is once again based on the model of information foraging. I suspect that the presence of an *eMoose* decoration on a method contributes another type of positive scent to the call; conversely, the lack of a decoration contributes a negative one. However, these *eMoose* scents are factors together with the other scents based on location, role, and naming, into a decision whether and when to explore the call.

Applying this interpretation to this case, the call to CQUC still emitted the same negative scents that warded off our control subjects, but also a positive scent from the *eMoose* decoration. Once other targets seemed less promising, this positive scent may have been sufficient to overcome the negative scents and push the perceived potential profitability of an exploration past the threshold. While the calls to RECV and CQUE were also decorated, the effect of the positive scent was not as dramatic here, as these already emitted positive scents due to their location and name relevancy. Similarly, the negative scent from a lack of decorations on the calls to CQUS and CREC was not sufficient to prevent their exploration.

If my interpretation is accurate, then it implies a limitation on the potential benefits of *eMoose*, since it is still possible for targets with important directives never to be explored if they emit too many negative scents. However, it also limits the disruptive impact of too many decorated methods, which would have rendered the tool frustrating to use and thus impractical. Note that in the third task, subjects will be debugging a much larger program with many decorated calls, so we will have a chance to examine whether this expected behavior holds.

Based on the increased time spent in the core area and on its undecorated method CQUS, I also suspect that the presence of multiple decorated calls in a small area may increase the perceived scent of the entire area and even of undecorated items within it. However, more studies are needed to evaluate this possibility.

Impact of *eMoose* on exploration of RECV

My finding that *eMoose* users spent significantly more time on RECV than controls was surprising because its directives closely resembled its documentation, which stated the two blocking conditions, as was seen in Fig. 6.11. Subjects in both conditions tended to focus on these two conditions in a cycle of “inquiry episodes” [75] until they accepted that other blocking conditions are possible, and essentially acknowledged the imperfection of the text.

Previous studies [54] hinted that developers treat authors of respectable programs as “correct until proven otherwise” and are reluctant to second-guess their work. The decorations and presence of explicit directives may have somehow lent credibility and authority to the blocking conditions, making users more reluctant to distrust them and begin exploring other options. The potential effect of tools on perceived credibility and thus on performance deserves further study. Another possibility, however, is that users assumed that a decorated call is more likely to contain the solution.

Impact of the *eMoose* lower pane

The discussion here has focused on the impact of the decorations on function calls, as the task was concerned only with investigating which invoked methods are explored. The impact of the augmented *JavaDoc* hover was not expected to be significant since the documentation of most invoked methods was short and paralleled the presented directives.

In fact, as we have seen, several subjects ignored or bypassed that pane and read only the documentation text, and this may have contributed to their difficulties. Additional training and experience may bring more attention to this feature. Nevertheless, mouse movements and speech in the recordings suggest that most other subjects did focus on the *eMoose* pane first, and often quickly screened the method by systematically going over the listed directives. *eMoose* users may therefore be able to quickly determine the reasons that a call is decorated and decide whether to investigate further, without paying the cost of a full text read. However, more evidence is needed for this from subsequent tasks.

The second task, to which we now turn, is designed to explore in depth the impact of the augmented hover on long *JavaDocs*, while minimizing the impact of method decorations.

6.5 Second debugging task, based on the JMS API

Whereas the first task of our study was designed to investigate how developers choose to read the documentation of certain call targets, our second task is designed to study how they find directives in the text of such documentation. This task uses a small number of calls, offering sufficient opportunity to explore all of them, but the key directive is hidden deep in the verbose documentation of one of the targets.

6.5.1 The directive for this task

Like the preceding task, the directive and codebase for this task is also based on the JMS API. This time, however, the focus is not on the peer-to-peer communication facilities via queues, but rather on the publish-subscribe facilities which revolve around *topics*. JMS allows a program to announce itself as a publisher to a topic, to which it sends messages. JMS allows multiple clients to register themselves as subscribers to these topics, so they can receive any messages that are being published. However, the connections are typically not durable. In other words, clients only receive messages that are being published while they are active.

The second directive comes from the `setClientId` method in the `Connection` class. The documentation for this method was previously shown, and is reproduced in Fig. 6.17. As stated in the next-to-last paragraph, client identifiers can be used by JMS providers (typically broker processes) to maintain state information about the client. At present, the only permitted use is for *durable subscriptions*. This mechanism allows a subscriber in a particular process to pick up messages that were not read by a subscriber carrying the same identifier in a previous process that has already ended. This mechanism can be used, for example, to ensure reliable message delivery if a user switches between machines. The relevant bookkeeping takes place on the broker, which remains active even as the client switches processes.

As can be seen in Fig. 6.17, the documentation is quite verbose (277 words) and includes a variety of directives. The first paragraph indicates that there are better ways to configure connections, although this is merely a recommendation. The last paragraph indicates that it is an error to have two connections with the same client identifier active at the same time.

● **void javax.jms.Connection.setClientID(String clientID) throws JMSEException**

Sets the client identifier for this connection.

The preferred way to assign a JMS client's client identifier is for it to be configured in a client-specific `ConnectionFactory` object and transparently assigned to the `Connection` object it creates.

Alternatively, a client can set a connection's client identifier using a provider-specific value. The facility to set a connection's client identifier explicitly is not a mechanism for overriding the identifier that has been administratively configured. It is provided for the case where no administratively specified identifier exists. If one does exist, an attempt to change it by setting it must throw an `IllegalStateException`. **If a client sets the client identifier explicitly, it must do so immediately after it creates the connection and before any other action on the connection is taken. After this point, setting the client identifier is a programming error that should throw an `IllegalStateException`.**

The purpose of the client identifier is to associate a connection and its objects with a state maintained on behalf of the client by a provider. The only such state identified by the JMS API is that required to support durable subscriptions.

If another connection with the same `clientID` is already running when this method is called, the JMS provider should detect the duplicate ID and throw an `InvalidClientIDException`.

Parameters:
clientID the unique client identifier

Throws:
[JMSEException](#) – if the JMS provider fails to set the client ID for this connection due to some internal error.
[InvalidClientIDException](#) – if the JMS client specifies an invalid or duplicate client ID.
[IllegalStateException](#) – if the JMS client attempts to set a connection's client ID at the wrong time or when it has been administratively configured.

Figure 6.17: Javadocs for the `Connection.setClientID` method in JAVA JMS (Reproduced with added emphasis)

Our focus, however, is on the first highlighted directive, which states that a call to this function must come after the connection object has been created but before any other operations on the connection. An exception is “promised” if this obvious protocol is violated. This kind of protocol makes sense since connections in JMS are used to generate sessions, which are in turn used to create other communication related objects. If the client identifier has an effect or propagates to any of these objects, it becomes necessary to set it up before any of them are created or the connection is started. The interesting point about this directive is that it is placed in a paragraph deep within the verbose documentation, and starts deep inside that paragraph. I conjecture that readers would be more likely to miss it, albeit for a very different reason from the one in the first task.

Also note that the highlighted directive has a corresponding redundancy in the `throws` clause at the end of the *JavaDoc*, which states that an `IllegalStateException` may be thrown if the identifier is set at “the wrong time” or if it is administratively configured. This may confuse users, or actually lead them towards the full directive since they could skim the text in search of more details on the `IllegalStateException`.

Finally, note that because of the length of this *JavaDoc*, it may not be fully visible when developers view it using the *JavaDoc* hover. In fact, in the study, the default viewport extended only up to the middle of the highlighted directive. Users in the experimental condition, however, also saw a lower pane with three directives: **1) Client ID must be unique among running clients, 2) If ID is set explicitly, it must be done immediately after creating connection and before any other action, 3) Preferred way is via configuration in a `ConnectionFactory` which is then used to create connection.**

6.5.2 Codebase and task

The codebase for this task is taken from the sample file `DurableSubscriberExample`. In this file, a main class uses several inner classes to create a publisher and several subscribers, which are then disconnected and reconnected with the same client identifier to demonstrate the concept of durable subscribers. Since our focus is on reading long *JavaDocs* rather than on which calls are read, we immediately take the subjects to a specific fragment, shown in Fig. 6.18, and specifically to lines 126–131 in the constructor of the `DurableSubscriber` inner class.

```
38⊕ * The DurableSubscriberExample class demonstrates that a durable subscription.
57 public class DurableSubscriberExample {
58     String     topicName = null;
59     int        exitResult = 0;
60     static int startindex = 0;
61
62     String BROKER_ADDRESS = "Address";
63
64     * The DurableSubscriber class contains a constructor, a startSubscriber
65⊕ public class DurableSubscriber {
66     TopicConnection topicConnection = null;
67     TopicSession    topicSession = null;
68     Topic           topic = null;
69     TopicSubscriber topicSubscriber = null;
70     TextListener    topicListener = null;
71
72     * The TextListener class implements the MessageListener interface by
73⊕ private class TextListener implements MessageListener {
74
75
76
77
78
79
80
81
82⊕ * Constructor: looks up a connection factory and topic and creates a
83⊕ public DurableSubscriber() {
84     TopicConnectionFactory topicConnectionFactory = null;
85
86     try {
87
88         topicConnectionFactory = new ActiveMQConnectionFactory(BROKER_ADDRESS);
89         topicConnection = topicConnectionFactory.createTopicConnection();
90         topicSession = topicConnection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
91         topicConnection.setClientID("DurableSubscriberExample");
92         topic = topicSession.createTopic(topicName);
93         topicConnection.start();
94
95     } catch (Exception e) {
96         e.printStackTrace();
97         System.out.println("Connection problem: " + e.toString());
98         if (topicConnection != null) {
99             try {
100                 topicConnection.close();
101             } catch (JMSEException ee) {}
102         }
103         System.err.println("Exception: " + e);
104         SampleUtilities.exit(1);
105     }
106 }
107
108 }
```

Figure 6.18: Constructor for the `DurableSubscriber` inner class

Subjects are told that when the test program executes, there is an error or exception which can be immediate or delayed. They are not told what the error is, but are instead told to imagine that they have already narrowed down the problem down to this constructor. They are assured that the system state and parameters are all correct, but that there is some problem in those six lines. To avoid revealing the exact call responsible for the error and the resulting exception, which would turn the task into a simple search, subjects are not allowed to run the program and are asked to rely solely on the code and documentation.

The problem in this code fragment was caused by a reversion of the order between line 128 and 129,

so that now the call to `setClientId` comes after, rather than before, the call to `createTopicSession`. This will cause the program to throw an exception, as there is now an action (setting session) on the connection object prior to setting the client identifier. Note that such an inversion is quite plausible for a developer to make when the dependencies between lines are not fully clear.

The code fragment is very similar to the queue initialization sequence of the previous task but uses analogous operations that are specific to publish-subscribe topics rather than peer-to-peer queues. One difference in this task is that the call to `start`, which was eliminated in the previous task, is visible in this codebase. The only truly new method in this sequence is `setClientId`. Its presence should steer users towards investigating it, though the short size of the fragment is likely to ensure that anyway.

6.5.3 Results - Success rates

Success in this task required subjects to identify the directive stating that no other calls on the connection are allowed to take place before the call to `setClientId`, and to fix it by accordingly swapping two calls. Of 13 subjects who performed this task in the control condition, only 7 were successful and 6 were not. On the other hand, all the 13 subjects who performed this task in the experimental condition were successful.

Once again I applied a *Fisher's exact test* to test the independence of eMoose use and success, rejecting the null hypothesis. For this task, the difference in success rate was also significant ($p = 0.007$). However, we must once again examine the behaviors in details to understand the reasons for this difference.

In the detailed narrative below, we refer to certain call targets by acronyms. For ease of reference, a simplified version of the code section is presented with this mnemonics in Fig. 6.19 below. The acronyms are: FAC for the `ActiveMQConnectionFactory` constructor, CTOC for `createTopicConnection`, CTOS for `createTopicSession`, SCID for `setClientId`, CTOP for `createTopic`, and ST for `start`.

```

L1 topicConnectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");
                                     FAC
L2 topicConnection = topicConnectionFactory.createTopicConnection();
                                               CTOC
L3 topicSession = topicConnection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
                                               CTOS
L4 topicConnection.setClientId("DurableSubscriberExample");
                                               SCID
L5 topic = topicSession.createTopic(topicName);
                                               CTOP
L6 topicConnection.start();ST

```

Figure 6.19: Method mnemonics in the codebase for Task 1

On the machine used in the study, the default size of the *JavaDoc* hover typically only covered the text up to the middle of our directive, and it had to be manually resized or scrolled to reveal more of the text, including our key directive. To facilitate analysis, I created a detailed log containing the subject's interactions with the method and the visible viewport into the text at each point in time.

6.5.4 Results - Successful subjects in the control condition

Table. 6.7 presents the timing data for the seven successful subjects in the control condition of this task. Unfortunately, the recording for subject S22 was corrupted, leaving us with data for six subjects.

Task 2, Control Condition, Successful Subjects										
	S2	S8	S11	S14	S16	S19	S22	Avg	Avg>1	Ct.>1
FAC	3.2	0	0.8	1.8	5.0	2.6	C o r r u p t e d	2.2	3.2	4 of 6
CTOC	21.0	45	26.7	11.8	25.6	28.0		26.4	26.4	6 of 6
CTOS	22.5	88.8	85.5	15.4	43.0	41.0		49.4	49.4	6 of 6
SCID	125.3	243	381.3	61.0	125.0	70.8		167.7	167.7	6 of 6
CTOP	0.0	16.7	30.9	0.0	0.0	0.0		7.9	23.8	2 of 6
ST	0.0	0	12.1	0.0	0.0	0.0		2.0	12.1	1 of 6
Other	0.0	48.4	4.9	0.0	31.2	29.2		19.0	22.7	4 of 6
Read time	172.0	441.9	542.2	90.0	229.8	171.6		274.6		
Work time	234.0	600.0	560.0	120.0	285.0	257.0		342.7		
Read/work	73.5%	73.7%	96.8%	75.0%	80.6%	66.8%		80.1%		
Ct. reads	5	17	21	4	7	11		11		
Avg read	34.4	26.0	25.8	22.5	32.8	15.6	25.3			

Table 6.7: Timing data for successful subjects in control condition of Task 2

The 6 successful controls for which we have data spent on average 80% of their work time (4.5 minutes) reading *JavaDocs*. Nearly 3 minutes of that time, on average, was devoted to the SCID method, with the rest spent primarily on CTOC and CTOS.

To understand what reading choices they made, I investigated the detailed transcripts. I found that subjects S2, S14, S16, and S19 essentially followed a systematic top-down approach in which they explored every call and attempted to eliminate every clause in its *JavaDocs* until they reached the directive. Since they found the problem in SCID in one pass, they never continued past it to CTOP and ST. Note that even during this systematic exploration, they sometimes closed the hover, examined other objects (but not methods), and hovered again on the same item.

The other two successful controls behaved differently. Subject S8 started out with a top-down approach, but appeared to miss the directive in SCID or not recognize its importance. He continued, reached the end of the code section, and came to the wrong conclusion that the call to `start` had to be moved. He began “thrashing” and exploring other options, until eventually returning to SCID and finding the directive. Overall, he made 17 *JavaDoc* reads. Subject S11 also read SCID, but was jumping around its text and missed the directive. He only found it after a total of 22 *JavaDoc* reads.

6.5.5 Results - Unsuccessful subjects in the control condition

Table 6.8 presents the timing data for the 6 unsuccessful controls. Five of them did not find the cause of the problem nor fixed it, and one (S7) was able to fix the problem but not explain what the problem was.

Task 2, Control Condition, Unsuccessful subjects										
	S4	S5	S7	S18	S24	S25	Avg	Avg>1	Ct.>1	
FAC	1.8	4.1	60.0	0.6	8.9	0.0	12.6	18.7	4 of 6	
CTOC	39.3	77.1	38.1	24.6	62.6	0.1	40.3	48.3	5 of 6	
CTOS	94.9	119.0	63.1	124.3	137.2	64.0	100.4	100.4	6 of 6	
SCID	144.5	297.7	289.5	16.6	204.0	97.3	174.9	174.9	6 of 6	
CTOP	154.2	86.3	37.4	41.3	56.9	27.9	67.3	67.3	6 of 6	
ST	0.4	57.3	0.0	4.5	27.5	0.0	15.0	29.8	3 of 6	
Other	24.5	2.5	22.2	29.2	10.8	23.2	18.7	18.7	6 of 6	
Read time	459.6	644.0	510.3	241.1	507.9	212.5	429.2			
Work time	922.0	936.7	985.0	949.0	900.0	900.0	932.1			
Read/work	49.8%	68.8%	51.8%	25.4%	56.4%	23.6%	46.0%			
Ct. reads	35	31	22	41	47	11	31.2			
Avg read	13.1	20.8	23.2	5.9	10.8	19.3	13.8			

Table 6.8: Timing data for unsuccessful subjects in control condition of Task 2

The unsuccessful subjects spent more time in absolute terms reading *JavaDocs* than their successful peers, although this accounted for a much smaller portion of their work time. They also spent slightly

more time, on average and in absolute terms, reading SCID, with only S18 spending very little time there. The major difference, however, is that unsuccessful controls spent significantly more time on other calls beside SCID. For example, they spent twice as much time than controls on CTOC and CTOS, and more than twice on CTOP and ST. The total number of reading actions, due to repeat visits, was also significantly higher. This is consistent with the behavior of subjects S8 and S11 among the successful controls, who did not find the directive and therefore searched in more places.

Since almost all unsuccessful controls spent significant amounts of time on SCID, we need to understand whether they saw the directive and not realized its importance, or whether they had missed it. As mentioned above, I created a detailed log of the areas of the *JavaDoc* of SCID which were visible at each point in time. For ease of reference, I marked each clause in the *JavaDoc* of SCID with a serial number, presented in Fig. 6.20. The tables that shall now be presented reflect the intervals during which specific portions, highlighted in light green, were visible. They use darker green to indicate that the user was moving the mouse over the clause, and an even darker shade to indicate when the mouse was explicitly used to select the text of the clause or when the user was reading the text out loud.

void javax.jms.Connection.setClientID(String clientID) throws JMSEException

Sets the client identifier for this connection.

P1 The preferred way to assign a JMS client's client identifier is for it to be configured in a client-specific `ConnectionFactory` object and transparently assigned to the `Connection` object it creates.

P2 Alternatively, a client can set a connection's client identifier using a provider-specific value. **P3** The facility to set a connection's client identifier explicitly is not a mechanism for overriding the identifier that has been administratively configured. It is provided for the case where no administratively specified identifier exists. **P4** If one does exist, an attempt to change it by setting it must throw an `IllegalStateException`. **P5** If a client sets the client identifier explicitly, it must do so immediately after it creates the connection and before any other action on the connection is taken. **P6** After this point, setting the client identifier is a programming error that should throw an `IllegalStateException`.

P7 The purpose of the client identifier is to associate a connection and its objects with a state maintained on behalf of the client by a provider. **P8** The only such state identified by the JMS API is that required to support durable subscriptions.

P9 If another connection with the same `clientID` is already running when this method is called, the JMS provider should detect the duplicate ID and throw an `InvalidClientIDException`.

Parameters: **P10**
`clientID` the unique client identifier

Throws:

P11 `JMSEException` - if the JMS provider fails to set the client ID for this connection due to some internal error.

P12 `InvalidClientIDException` - if the JMS client specifies an invalid or duplicate client ID.

P13 `IllegalStateException` - if the JMS client attempts to set a connection's client ID at the wrong time or when it has been administratively configured.

Figure 6.20: Javadocs of setClientId with enumerated clauses

Fig. 6.21 presents the view table for subject S4. As we can see, the subject made 8 visits to SCID and still missed the directive or glossed over it. In visits 1, 2, 5, 6 and 7 he only revealed the default viewport, so that our directive in (in P5–P6) was not fully exposed. In his third visit, which was the first in which he clicked on the *JavaDoc* and explored lower sections, and on the eighth visit, he scrolled lower so that our directive was visible, but his focus was on the later directives. In his fourth visit, he focused on the throws clause.

SCID Viewports for Subject S4																
Start	End	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	
01:03:90																
01:06:01																
01:06:30	01:06:80															
03:50:90																
04:10:00	04:23:80															
04:42:90																
04:48:10																
04:50:00																
05:02:00																
05:05:00	05:06:80															
06:14:70																
06:25:00																
06:29:00	06:32:00															
13:02:00																
13:07:00	13:17:02															
13:28:00																
13:32:80																
13:35:50																
13:36:00																
13:40:00	13:48:00															
13:59:00	14:06:60															
15:16:90																
15:21:40																
15:26:00																
15:32:00																
15:36:00	Time ends															

Figure 6.21: Log of visible regions of SCID documentation for subject 4

SCID Viewports for Subject S5																
Start	End	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	
03:00:00	03:07:00															
03:09:00																
03:24:00																
03:31:40	04:06:60															
06:49:30																
06:52:70																
06:59:50																
07:19:19	07:22:00															
07:32:00																
07:36:00																
07:38:00																
07:48:00																
08:57:00																
09:29:50	10:44:00															
10:50:70																
11:00:00																
11:43:50	11:57:40															
11:59:60																
12:04:40																
12:06:00	12:14:50															
14:56:00																
15:12:00	15:36:70															

Figure 6.22: Log of visible regions of SCID documentation for subject 5

Fig. 6.22 presents the view table for subject S5. As can be seen, the subject made 10 visits to the SCID method. In the first four, he clicked and scrolled the view to the last part, which described the later directives and exception; our core directive was outside his view. On the next few visits, he resized the hover window to show everything, making it difficult to determine the focus as he did not use the mouse or vocalize. He did another systematic scan and a last-minute visit to the top part, but never seem to have encountered or paid attention to our core directive.

SCID Viewports for Subject S7																
Start	End	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	
02:36:00	02:40:00															
06:01:00																
06:18:00																
06:24:00																
06:34:00																
07:15:00																
07:31:00																
07:47:00																
07:57:00																
08:04:00																
08:17:50																
08:25:00																
08:33:00																
08:39:00																
08:50:00																
09:06:00	09:13:00															
15:11:00																
15:29:00																
15:32:00																
15:39:00																
15:44:00																
15:56:00																
16:08:00																
16:22:00	Time ends															

Figure 6.23: Log of visible regions of SCID documentation for subject 7

Fig. 6.22 presents the view table for subject S7. This subject is different than the other unsuccessful controls in that he did fix the problem, but could not determine precisely why the fix had worked. What happened is that he followed a systematic approach in exploring functions, and then systematically went over all the text in the hover window that he had maximized, vocalizing or using the mouse and selections as he covered each clause. He eventually reached our core directive, but then had a momentary lapse of concentration. He read out the directive, but then wanted to explore the list of exceptions and quickly moved to the end. From then on, he focused on the exceptions, and started pondering what “wrong time” could mean. He conjectured that the two functions needed to swap and did so, thus fixing the bug, but when prompted to explain why he kept looking for a reason, he never recalled the directive which had lead him there.

Subject S18 spent so little time on SCID, that it is perhaps not surprising that he never encountered the directive.

Fig. 6.24 presents the view table for subject S24. As can be seen, this subject made several scans of the documentation but never seemed to focus on the core directive, although it was occasionally visible. It appears that he may have read the beginning of the paragraph, but ignored its ending, including our core directive.

SCID Viewports for Subject S24															
Start	End	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13
01:21:00															
01:27:50															
01:35:40															
01:40:00															
01:58:00															
03:55:00															
04:14:00															
04:21:00															
04:39:00															
04:41:60															
04:44:70															
04:59:00															
R07:10															
R07:24															
R07:47															
R07:57															
R08:09															
12:00															
13:50:00															
14:03:00															

Figure 6.24: Log of visible regions of SCID documentation for subject 24

Subject S25 scanned SCID once in a systematic manner, but his questions suggest that he may have been distracted by the issues of duplicate identifiers in clause P9.

My general impression from examining the recordings for the control group is that the chances of reading a particular clause in the main body of the long *JavaDoc* text roughly decreases with distance from the top and with distance from the beginning of the visually distinct paragraph. In their initial visits to SCID, most subjects also seemed reluctant to click the hover to create a standalone window and then scroll the text beyond the initial viewport.

6.5.6 Results for successful subjects in the experimental condition

As previously indicated, all 13 subjects in the experimental condition were successful in identifying the problem and completing the task. Their timing data is presented in Fig. 6.25. Once again, the recording for one subject, S15, was corrupted.

Their average work time, *JavaDoc* reading time, and the time spent reading SCID were all roughly half of those of successful controls, indicating that the subjects were significantly more effective in their work. Note, however, that they spent significant amount of time, comparable to that of controls, on the decorated CTOC method. In addition, several of them spent significant amount of time on the other decorated method, CTOP. On the other hand, only half of subjects in this condition read CTOS, which was not decorated and which was read by everyone in the control group and significantly distracted the unsuccessful controls.

Since all subjects in both conditions read SCID, the difference might be in how it is read. Subjects in the experimental condition could see the directive in the lower pane without any need for scrolling.

Task 2, Experimental Condition, Successful Subjects																
	S1	S3	S6	S9	S10	S12	S13	S15	S17	S20	S21	S23	S26	Avg	Avg>1	Ct.>1
FAC	2.0	0.9	0.0	8.2	0.0	0.0	0.0	C o n t r o l s	10.6	0.0	0.0	0.6	0.0	1.9	6.9	3 of 12
CTOC	3.1	10.9	30.2	0.0	16.9	7.2	91.7		0.0	21.8	79.6	34.5	43.0	28.2	33.9	10 of 12
CTOS	0.0	0.2	2.6	0.0	0.0	7.4	12.5		0.0	0.0	12.1	4.3	5.6	3.7	7.4	6 of 12
SCID	18.4	45	85.7	35.6	53.2	51.8	68.2		41.6	36.2	200.1	64.9	48.3	62.4	62.4	12 of 12
CTOP	0.0	0	47.6	0.0	33.8	0.0	0.0		0.0	0.0	101.5	44.9	0.0	19.0	57.0	4 of 12
ST	0.0	1.8	9.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	7.0	0.0	1.5	5.9	3 of 12
Other	8.1	8.2	1.2	0.0	4.1	0.0	0.0		0.9	0.0	7.1	11.4	3.7	3.7	5.9	7 of 12
Read time	31.6	67.0	176.3	43.8	108.0	66.4	172.4		53.1	58.0	400.4	167.6	100.6	120.4		
Work time	165.0	110.0	230.0	60.0	163.1	203.8	205.0		112.0	165.1	498.0	292.6	155.0	196.6		
Read/Work	19.2%	60.9%	76.7%	73.0%	66.2%	32.6%	84.1%		47.4%	35.1%	80.4%	57.3%	64.9%	61.2%		
Ct. reads	9	7	13	4	14	3	4		5	10	17	16	6	9		
Avg read	3.5	9.6	13.6	11.0	7.7	22.1	43.1		10.6	5.8	23.6	10.5	16.8	13.4		

Figure 6.25: Timing data for successful subjects in experimental condition of Task 2

Indeed, the recordings revealed that only three subjects: S10, S13, and S23 scrolled the text viewport in a way that could have revealed the entire directive. Thus, for all those other subjects the directive in the lower pane was sufficient to identify and fix these problems. However, even those subjects who never needed to scroll did not necessarily realize the implication of the directive immediately. Subjects S6 and S21 visited SCID several times before identifying the directive. It appears that the implications of the directive were not clear, or that subjects were distracted by the first directive. The three subjects who did scroll also made several visits, so they may have faced the same challenge.

6.5.7 Discussion

The limited scale of this task, which is even smaller than that of the first task, lent itself to a systematic exploration of each call and of the documentation of each call. Such behavior, which is of course not practical for larger programs [75], appeared to account for the success of many of the controls. However, the small scale also meant that there was sufficient time for everyone to explore the SCID method and become aware of the directive.

Failures among controls

Since everyone explored SCID, I argue that the 7 of 13 success rate among controls is low considering that certain conditions were favorable compared to real world situations. The code fragment was extremely short and known to contain a bug whose solution was likely to appear in the documentation. One may even argue that the gap between uses of the connection and session objects looked suspicious. This demonstrates the serious difficulties that developers may face in becoming aware of important directives in verbose text. I acknowledge, though, that many subjects were relative novices, worked with limited context, and had to cope with documentation of unfamiliar concepts.

Since all subjects in the control group, with the exception of S18, spent a significant amount of time in SCID, failure in the task likely resulted from a failure to notice the directive or to realize its importance. My analysis of the viewports shows that users tend to read the materials at the beginning of the documentation, especially as that is what is shown to them without additional effort.

Once scrolling begins, it appears that reading is not necessarily systematic, but rather opportunistic: these subjects sometimes focused on materials towards the end of the text, missing materials in the middle. My interpretation is that within the verbose text, certain visual or textual cues attract more attention than anything exhibited by our directive. One such cue is the explicit listing of thrown exceptions, which are easy to identify and seemed to attract significant attention. Another type of cue is the presence of a relatively short paragraph with distinct vertical space around it, as was the case for the directive in P7-P8

and the directive in P9.

Conversely, our directive was often not noticed even when it was visible on the screen, regardless of its relative position in the viewport at the time. It appears that several factors contributed to missing it or not realizing its importance, including: its placement late in the text and deep within another paragraph, the lack of keywords to attract a reader's attention, and a phrasing that was not sufficiently clear or powerful. Even those who noticed it might have considered it unremarkable and unmemorable, as was apparently the case with subject S7, who used it to find the relevant exception but could not recall having read it previously.

Since documentation writers cannot assume that all readers would be thorough, the key to directive awareness, then, is in making directives more salient through an explicit effort by the author or via tools like *eMoose*.

Experimental group

Since the SCID method was explored by all subjects in the control condition, it is unlikely that method decorations had significant impact on success in the experimental condition. A possible impact may have been in increasing the perception that the solution to the problem may lie in SCID, which could lead to closer scrutiny among those in the experimental condition. However, since CTOC and CTOP were also decorated, whereas three other methods were not, these methods provided sufficient distraction to offset this effect, and indeed attracted the attention of many subjects in the experimental condition.

The much higher success rates and shorter time spans in this condition are therefore likely to result from the presence of the lower pane, since directives can be missed within the verbose text of SCID. This is particularly likely since many subjects in the experimental condition were able to solve the problem without ever scrolling the *JavaDoc*.

Even with this presentation, however, identifying the problem often took time and repeated visits. This may be indicative of the risk of stating directives in unclear ways, or of having one directive distract readers from subsequent ones.

Note that the fact that many subjects relied on the explicit list of directives and never explored the text raises the risk that information would be missed if it were not explicitly tagged as directives. Whoever tags the documentation must therefore be careful to ensure that no directives or critical information are left untagged.

Improving the usability of the documentation text

While API authors can do little (without tools) to attract readers to the documentation of their methods, how they write this documentation can have significant impact on its utility to potential readers. The reading behaviors observed in this task, but also in other tasks in this study, suggest ways in which documentation can be made more usable.

First, authors should not assume that the entire text would be read thoroughly in a linear order, and must write accordingly. In particular, authors should determine what details are the most critical and present them first. Each clause should also describe the instruction or required action before describing the rationale or the qualifying issues. In my survey of APIs, I found many descriptions that began with a long line of reasoning, and then concluded with an instruction to the caller. Since readers skimming the text tend to focus on the beginning of each paragraph, they are more likely to miss the directive. Also, the phrasing of directives should be more forceful and direct, and should use stronger phrases such as “warning”, “note that”, etc.

Second, authors should review their documentation and consider whether some of the information can effectively be pushed into the signature. For example, renaming a method of the form `getValue` to `getValueBlocking` is a more effective way to convey that it may block if the value is not available than merely documenting that fact. Similarly, calling it `tryAcquireValue` is more effective than documenting that it would return immediately if there is failure or that it would return a null.

Improving documentation generation tools

My findings indicate that current documentation support tools, which present the same text to all users, should be expanded to allow the creation of different documentation for different audiences.

A single medium forces authors to pick between creating detailed documentation that can be read in the browser, and shorter documentation for reading via IDE tooltips. Because *JavaDocs* were originally distributed in HTML form, developers were encouraged to provide a detailed narrative similar to the contents of a web page or a blog post. In many ways, however, the tooltip window is more suited to “tweet”-like materials than to complete “blog posts”. The text should be much more concise, with the most important details clearly visible and separate from one another. In particular, while lengthy paragraphs may fit well in a narrative, multiple shorter structures that convey a single idea are easier to skim.

A separate presentation could also be useful for the common scenario where a method’s author needs to convey some information to callers, and different information to those who implement or override it.

In addition, I believe that the technique of presenting a method’s “summary”, as is the case with web-based *JavaDocs*, does not improve usability. Good API design practices should focus on making method signatures self explanatory, so that the summary is left redundant. Throughout this study, subjects often read the summary sentence, were reassured that they understood the method’s purpose, and did not read further. As evidenced by the behavior of some subjects, this may have led them to miss important directives.

The results of the first task demonstrated that *eMoose* decorations may help lead developers to explore the documentation of call targets with relevant directives. The results of the second task showed that explicitly highlighting these directives in verbose text makes them more likely to be noticed. While these impacts are beneficial when the directives are truly relevant, there is a risk that developers may be distracted by many irrelevant calls and directives. We evaluate this risk in the third task, which uses the SWING API and the code of an entire program.

6.6 First Swing Task

The third task is primarily designed to investigate how developers choose what *JavaDocs* to read and how much attention to devote to them when confronted with a larger code base. I was particularly interested in investigating whether the presence of decorations on many unrelated calls would significantly distract *eMoose* users.

6.6.1 The directive for this task

The next directives comes from the standard SWING GUI toolkits packaged with JAVA. Initially, JAVA was released with the *Abstract Window Toolkit* as a platform-independent API for building user interfaces. This toolkit is relatively *thin*, and relies on the native UI implementation in the hosting operating

system. Since every OS has different widgets and they are not always compatible, the resulting programs are not truly portable. To overcome these limitations, later versions of JAVA included the SWING widget toolkit. SWING offered a variety of features such as a consistent *look and feel* with custom-drawn widgets, a rudimentary model-view-client architecture, and many others. However, many parts of SWING, including its component widgets, are based on AWT classes and services. Today, SWING is the most widely used GUI toolkit for JAVA.

The directive for this task is taken from a relatively obscure but highly useful SWING class called `JLayeredPane`. It is a subclass of the standard SWING `JComponent`, which is in turn a subclass of the AWT `Container`. Its benefit is that it allows contained items to be organized into *layers*, and its rendering functionality handles the hiding of elements in lower layers by elements in higher layers. Every object also has a position within a layer, so that it can be hidden by other items in the same layer and hide others, while still conforming to the hiding implied by the layer numbering.

```
void javax.swing.JLayeredPane.putLayer(JComponent c, int layer)
```

Sets the layer property on a JComponent. This method does not cause any side effects like `setLayer()` (painting, add/remove, etc). Normally you should use the instance method `setLayer()`, in order to get the desired side-effects (like repainting).

See Also:
[setLayer](#)

Parameters:
`c` the JComponent to move
`layer` an int specifying the layer to move it to

Figure 6.26: Javadocs for the `JLayeredPane.putLayer` method in SWING

```
void javax.swing.JLayeredPane.setLayer(Component c, int layer)
```

Sets the layer attribute on the specified component, making it the bottommost component in that layer. Should be called before adding to parent.

Parameters:
`c` the Component to set the layer for
`layer` an int specifying the layer to set, where lower numbers are closer to the bottom

Figure 6.27: Javadocs for the `JLayeredPane.setLayer` method in SWING

The directive I picked is in the method `putLayer`. This method is used to assign an object (that is usually already in the container) to a different layer. However, this method appears to merely change the layer assignment in the internal data structures maintained by the pane object, without actually causing any visual effect. Its documentation, depicted in Fig. 6.26, explicitly states that to get “side-effects” like repainting, one must call the method `setLayer` instead. This is an example of an *alternative directive*; it may appear informational, but one can easily foresee errors that can occur if it is ignored in the wrong context. Interestingly, the documentation of `setLayer`, depicted in Fig. 6.27, is much more terse and indicates nothing about refreshing.

Though I have frequently noticed these kinds of *alternative directives*, this specific case is particularly interesting. We have two methods that superficially appear almost the same and each may meet the expectations of a developer seeking to reassign a contained object to a different layer. When developers try to figure out how to accomplish this goal, they may automatically think of the *set* verb because of the setting of property, or they may actually think of the *put* verb if they think about the act of putting an object in a particular layer. If the developer explores the summary descriptions in the web-based *JavaDoc*, as depicted in Fig. 6.28, the difference between the methods is not evident, though if one goes lexically, `putLayer` may be found first and the search might not continue. The only way to truly distinguish between the two is to read the documentation.

Note that the API is not wrong in providing two ways to set a layer, though a better effort at disambiguation would have been beneficial. If every change to the model was immediately reflected in the

	returns a string representation of this <code>JLayeredPane</code> and.
static void	<code>putLayer(JComponent c, int layer)</code> Sets the layer property on a <code>JComponent</code> .
void	<code>remove(int index)</code> Remove the indexed component from this pane.
void	<code>setLayer(Component c, int layer)</code> Sets the layer attribute on the specified component, making it the bottommost component in that layer.
void	<code>setLayer(Component c, int layer, int position)</code> Sets the layer attribute for the specified component and also sets its position within that layer.
void	<code>setPosition(Component c, int position)</code> Moves the component to <code>position</code> within its current layer, where 0 is the topmost position within the layer and -1 is the bottommost position.

Figure 6.28: Outline area in the Web-based JavaDocs for the `JLayeredPane` class

UI, this would be computationally intensive and cause flicker. Since some changes occur in *batches*, the availability of a method that does not cause a refresh is important. Support for batching is a common idiom in toolkits like SWING.

6.6.2 Task description

Since our directive comes from the `JLayeredPane` class, I chose to base this task on Sun's official demo for this class which is part of the official Swing tutorial.¹ The demo class, called `JLayeredPaneDemo`² consists of 220 lines.

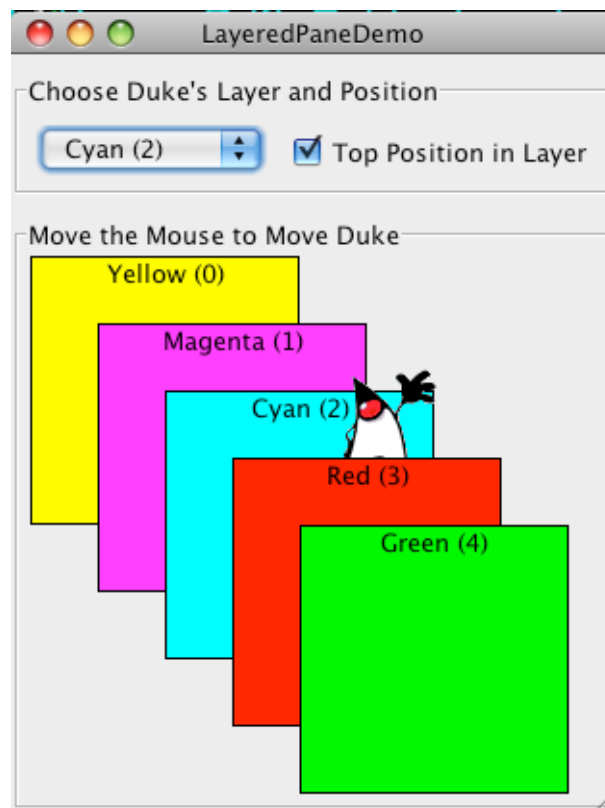


Figure 6.29: Initial state of the `JLayeredPaneDemo` program

When executed, it presents on the screen a Swing window with a control panel on top and a `JLayeredPane`

¹The demo can be found at <http://java.sun.com/docs/books/tutorial/uiswing/components/layeredpane.html>.

²<http://java.sun.com/docs/books/tutorial/uiswing/examples/components/LayeredPaneDemoProject/src/components/LayeredPaneDemo.java>

on the bottom, as can be seen in Fig. 6.29. The layered pane has five predefined layers, ranging from 0 to 4. To illustrate them, a colored square (actually a `JLabel` widget) is placed in each layer with a caption indicating the layer number. A graphical image (Java's mascot *Duke*) is then placed in layer 2, so that it is in front of the purple box of layer 1 but behind the red box of layer 3. Within layer 2, however, its position is assigned to be in front of the cyan box. Moving the mouse within the lower pane moves Duke while still respecting the layer ordering.

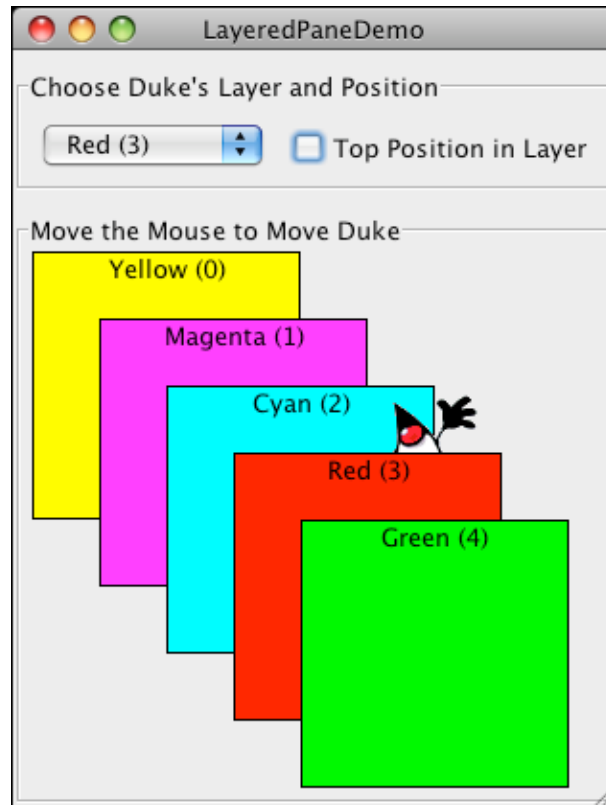


Figure 6.30: State of the `JLayeredPaneDemo` program after Duke is moved to layer 3

The upper pane offers two ways for clients to change the location of Duke. By unchecking the `top position in layer` checkbox, Duke will be moved to a lower position within his current layer. As a result he will appear behind the corresponding box (e.g., the cyan) while still appearing in front of lower-ranked layers but behind higher-ranked ones. The user can also move duke to another layer using the list box, affecting the way he is displayed. In this case, however, his position relating to the box representing the new layer depends on the state of the checkbox. For example, Fig. 6.30 depicts Duke after he had been moved to layer 3 but behind its box.

As with previous tasks, however, I have broken the program. In this case, when the user moves Duke to certain layers by making a selection in the list box, it has no effect. When the user moves Duke with the mouse, Duke stays in the previous layer. For example, Fig. 6.31 shows how Duke is still in front of the box of layer 3 even though it had been moved to layer 1. Subjects are shown such scenarios, and then also shown that checking and unchecking the checkbox results in Duke being assigned to the appropriate layer.

The problem has been generated by making a change to the method `actionPerformed`, which occupies lines 173-188 and is depicted with *eMoose* annotations in Fig. 6.32. The change that I have made is that initially, the lower conditional (for `LAYER_COMMAND`) did not have the call to `putLayer` and `setPosition`. Instead, it had a single call to `setLayer`:

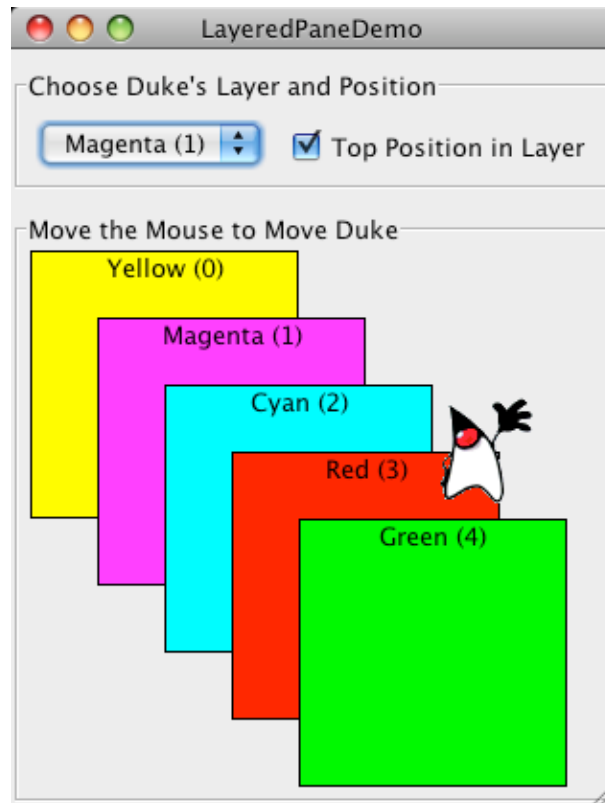


Figure 6.31: Error state of the JLayeredPaneDemo program after Duke is moved to layer 1

```

175 public void actionPerformed(ActionEvent e) {
176     String cmd = e.getActionCommand();
177
178     if (ON_TOP_COMMAND.equals(cmd)) {
179         if (onTop.isSelected())
180             layeredPane.moveToFront(dukeLabel);
181         else
182             layeredPane.moveToBack(dukeLabel);
183
184     } else if (LAYER_COMMAND.equals(cmd)) {
185         int position = onTop.isSelected() ? 0 : 1;
186         layeredPane.putLayer(dukeLabel,
187                             layerList.getSelectedIndex());
188         layeredPane.setPosition(dukeLabel, position);
189     }
190 }

```

Figure 6.32: Source code of the actionPerformed method in the JLayeredPaneDemo class

`layeredPane.setLayer(dukeLabel, layerList.getSelectedIndex(), position);` To break the program, I split this call into a call to `setLayer` with two parameters, and a separate call to `setPosition`. I then replaced the call to `setLayer` with the call to `putLayer`, thus preventing the refresh that would move Duke. When the user changes the checkbox, the calls to `moveToFront` and `moveToBack` implicitly cause a refresh. If the user replaces the call to `putLayer` with the call to `setLayer`, the problem is fixed.

Note that the program itself is quite long and the reader is encouraged to consult the source code. Fig. 6.33 presents a “folded” version, which only shows the declarations of fields and methods.

Since we will be investigating the distraction by decorated calls outside `actionPerformed`, we present the source code for methods that contain them and count instances. The class constructor, depicted in Fig. 6.34, contains 6 calls to `add`, a single call to `setBorder` and another call to `setBackground`. The methods `createColoredLabel` and `createControlPanel`, depicted in Fig. 6.35, add 2 more calls each to `add` and to `setBorder`, and a single call to `setBackground`. The method `createAndShowGUI` (in Fig. 6.36) has a call to `setDefaultCloseOperation`, while `main` (also in Fig. 6.36) has a call to `invokeLater`. In summary, outside the core area we have 8 calls to `add`, 3 calls to `setBorder`, 2 calls to `setBackground`, and single calls to `setDefaultCloseOperation` and `invokeLater`.

6.6.3 Results - Success rates

Of 13 subjects who performed this task in the control condition, only 5 were successful and 8 were not. On the other hand, all of the 12 subjects who performed this task in the experimental condition were successful. Note that group sizes are unbalanced since one additional subject who was assigned to perform this task in the experimental condition left the study early for personal reasons and did not even start this task.

Once again I applied a *Fisher’s exact test* to test the independence of `eMoose` use and success, rejecting the null hypothesis in each case. For this task, the difference in success rate was also significant ($p = .001$). However, we must once again examine the behavior of subjects in detail to understand the reasons for these differences and the impact of *eMoose*.

To understand the differences between subjects in each condition and outcome group, I created comprehensive logs of the time spent reading *JavaDocs*. For each element, I counted the total time spent on that element as well as the number of times it was read, and this will be presented in the form $tnt - nx$ where tnt represents the time in seconds and n represents the number of reads.

Rather than present detailed timing data for every method and element in the program, most values are aggregated into five categories. First, we distinguish between the reading of calls to methods outside the current program, which will be reflected in our first four categories, and the reading of “other” elements, including: fields, variables, classes, declarations, and methods within the program. In our tables, all method names will be followed by parentheses. Second, we distinguish between elements within the `actionPerformed` method, which we term “core”, and elements outside it (“other”). Finally, we also distinguish between methods which are not decorated in the experimental condition, and those that are, which we indicate with an asterisk after their name. We make this distinction to enable an analysis of the impact of *eMoose* on this subset of methods. However, while we will discuss how subjects in the control condition read these decorated methods, it is important to remember that these subjects do not see these decorations.

In the tables presented below, our categories are: `DecoratedCoreMethod` and `UndecoratedCoreMethod` for `actionPerformed`, `DecoratedOtherMethod` and `UndecoratedOtherMethod` for other locations, and `Other` for non-methods (which will be broken into specific subtypes). In addition, details will be included for

```

LayeredPaneDemo.java
4+ * Redistribution and use in source and binary forms, with or without.
31
32 package edu.cmu.contextstudy.jdk;
33
34+ import javax.swing.*;
40
42+ * LayeredPaneDemo.java requires
45 @SuppressWarnings("static-access")
46 public class LayeredPaneDemo extends JPanel
47         implements ActionListener,
48                 MouseMotionListener {
49+ private String[] layerStrings = { "Yellow (0)", "Magenta (1)",
52+ private Color[] layerColors = { Color.yellow, Color.magenta,
55
56 private JLayeredPane layeredPane;
57 private JLabel dukeLabel;
58 private JCheckBox onTop;
59 private JComboBox layerList;
60
61 //Action commands
62 private static String ON_TOP_COMMAND = "ontop";
63 private static String LAYER_COMMAND = "layer";
64
65 //Adjustments to put Duke's toe at the cursor's tip.
66 private static final int XFUDGE = 40;
67 private static final int YFUDGE = 57;
68
69+ public LayeredPaneDemo() {}
118
119 /** Returns an ImageIcon, or null if the path was invalid. */
120+ protected static ImageIcon createImageIcon(String path) {}
129
130 //Create and set up a colored label.
131+ private JLabel createColoredLabel(String text,
144
145 //Create the control pane for the top of the frame.
146+ private JPanel createControlPanel() {}
164
165 //Make Duke follow the cursor.
166+ public void mouseMoved(MouseEvent e) {}
169 public void mouseDragged(MouseEvent e) {} //do nothing
170
171 //Handle user interaction with the check box and combo box.
172
173+ public void actionPerformed(ActionEvent e) {}
189
191+ * Create the GUI and show it. For thread safety,
195+ private static void createAndShowGUI() {}
209
210+ public static void main(String[] args) {}
219 }
220

```

Figure 6.33: Folded code outline for the JLayeredPaneDemo class

```

69  public LayeredPaneDemo()    {
70      setLayout(new BorderLayout(this, BorderLayout
71          .PAGE_AXIS));
72
73      //Create and load the duke icon.
74      final ImageIcon icon = createImageIcon("images/dukeWaveRed.gif");
75
76      //Create and set up the layered pane.
77      layeredPane = new JLayeredPane();
78      layeredPane.setPreferredSize(new Dimension(300, 310));
79      layeredPane.setBorder(BorderFactory.createTitledBorder(
80          "Move the Mouse to Move Duke"));
81      layeredPane.addMouseMotionListener(this);
82
83      //This is the origin of the first label added.
84      Point origin = new Point(10, 20);
85
86      //This is the offset for computing the origin for the next label.
87      int offset = 35;
88
89      //Add several overlapping, colored labels to the layered pane
90      //using absolute positioning/sizing.
91      for (int i = 0; i < layerStrings.length; i++) {
92          JLabel label = createColoredLabel(layerStrings[i],
93              layerColors[i], origin);
94          layeredPane.add(label, new Integer(i));
95          origin.x += offset;
96          origin.y += offset;
97      }
98
99      //Create and add the Duke label to the layered pane.
100     dukeLabel = new JLabel(icon);
101     if (icon != null) {
102         dukeLabel.setBounds(15, 225,
103             icon.getIconWidth(),
104             icon.getIconHeight());
105     } else {
106         System.err.println("Duke icon not found; using black square instead.");
107         dukeLabel.setBounds(15, 225, 30, 30);
108         dukeLabel.setOpaque(true);
109         dukeLabel.setBackground(Color.BLACK);
110     }
111     layeredPane.add(dukeLabel, new Integer(2), 0);
112
113     //Add control pane and layered pane to this JPanel.
114     add(Box.createRigidArea(new Dimension(0, 10)));
115     add(createControlPanel());
116     add(Box.createRigidArea(new Dimension(0, 10)));
117     add(layeredPane);
118
119 }

```

Figure 6.34: Source code of the constructor in the JLayeredPaneDemo class

```

132 //Create and set up a colored label.
133 private JLabel createColoredLabel(String text,
134                                 Color color,
135                                 Point origin) {
136     JLabel label = new JLabel(text);
137     label.setVerticalAlignment(JLabel.TOP);
138     label.setHorizontalAlignment(JLabel.CENTER);
139     label.setOpaque(true);
140     label.setBackground(color);
141     label.setForeground(Color.black);
142     label.setBorder(BorderFactory.createLineBorder(Color.black));
143     label.setBounds(origin.x, origin.y, 140, 140);
144     return label;
145 }
146
147 //Create the control pane for the top of the frame.
148 private JPanel createControlPanel() {
149     onTop = new JCheckBox("Top Position in Layer");
150     onTop.setSelected(true);
151     onTop.setActionCommand(ON_TOP_COMMAND);
152     onTop.addActionListener(this);
153
154     layerList = new JComboBox(layerStrings);
155     layerList.setSelectedIndex(2); //cyan layer
156     layerList.setActionCommand(LAYER_COMMAND);
157     layerList.addActionListener(this);
158
159     JPanel controls = new JPanel();
160     controls.add(layerList);
161     controls.add(onTop);
162     controls.setBorder(BorderFactory.createTitledBorder(
163         "Choose Duke's Layer and Position"));
164     return controls;
165 }

```

Figure 6.35: Source code of the `createColoredLabel` and `createControlPanel` methods in the `JLayeredPaneDemo` class

```

192  /**
193   * Create the GUI and show it. For thread safety,
194   * this method should be invoked from the
195   * event-dispatching thread.
196   */
197  private static void createAndShowGUI() {
198      //Create and set up the window.
199      JFrame frame = new JFrame("LayeredPaneDemo");
200      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
201
202      //Create and set up the content pane.
203      JComponent newContentPane = new LayeredPaneDemo();
204      newContentPane.setOpaque(true); //content panes must be opaque
205      frame.setContentPane(newContentPane);
206
207      //Display the window.
208      frame.pack();
209      frame.setVisible(true);
210  }
211
212  public static void main(String[] args) {
213      //Schedule a job for the event-dispatching thread:
214      //creating and showing this application's GUI.
215      javax.swing.SwingUtilities.invokeLater(new Runnable() {
216          public void run() {
217              createAndShowGUI();
218          }
219      });
220  }

```

Figure 6.36: Source code of the `createAndShowGUI` and `main` methods in the `JLayeredPaneDemo` class

every element in the `actionPerformed` method, and these numbers make up part of the aggregate values in the above categories.

The complete timing data for successful controls are presented in Table 6.9, and for unsuccessful ones in Table 6.10. Both tables are split into four parts. First, we present aggregate timing data for each specific type of element in the “other” category. Second, we present the data for each of the method categories. Third, we present the general statistics on reading time and work time seen in the previous tasks. Finally, we present the detailed timing for every element in the core area. Methods calls in the core area are followed by the parenthesis, and an asterisk indicates that they were decorated. Also note that `setLayer` is included in this table, since it could be viewed in different ways.

6.6.4 Results - Control condition

Behavior of successful subjects

We begin by examining the behavior of the 4 successful control subjects for which we have data; the recording for subject 15 was unfortunately corrupted, leaving us with a very small sample. The detailed timing data for these subjects is presented in Table 6.9.

		3	15	19	23	26				
		CTL	CTL	CTL	CTL	CTL				
		Pass	Pass	Pass	Pass	PASS				
		220	x	718.3	332.1	641.20	AVG	AVG >1	Portion	Users >1
Aggregated	ConstantField	0		0	0	0	0.0	0.0	0.0%	0 of 4
	Class	0		4.4 - 3x	5.5 - 1x	0	2.5	5.0	1.7%	2 of 4
	Constant	0		2.8 - 3x	0	1.3 - 1x	1.0	2.1	0.7%	2 of 4
	Variable	0		0	0.2 - 1x	0	0.1	0.2	0.0%	0 of 4
	Fields	0		2.2 - 2x	5.9 - 3x	9 - 3x	4.3	5.7	2.9%	3 of 4
	DeclarationMethod	0		5.1 - 2x	0	0	1.3	5.1	0.9%	1 of 4
	InternalMethod	0		3.6 - 5x	0	0	0.9	3.6	0.6%	1 of 4
	DecoratedCoreMethod	13.4 - 1x		43.8 - 2x	136.7 - 10x	73.3 - 4x	66.8	66.8	44.8%	4 of 4
	DecoratedOtherMethod	0		198.2 - 8x	0	0	49.6	198.2	33.2%	1 of 4
	UndecoratedCoreMethod	0		8.4 - 2x	4.7 - 1x	36.7 - 2x	12.5	16.6	8.3%	3 of 4
UndecoratedOtherMethod	0		19.8 - 8x	0.8 - 1x	20.9 - 1x	10.4	20.4	7.0%	2 of 4	
Stats	Read time	13.4		288.3	153.8	141.2	149.2			
	Work time	220		718.3	332.1	641.2	477.9			
	Read/Work	6.09%		40.14%	46.31%	22.02%	30.0%			
	Ct. reads	1		35	17	11	16.0			
Core	ON_TOP_COMMAND			0.8 - 1x						
	LAYER_COMMAND					1.3 - 1x				
	position									
	cmd				0.2 - 1x					
	dukeLabel				5.9 - 3x					
	layeredPane					1.2 - 1x				
	onTop					7.8 - 2x	2.0	7.8		1 of 4
	putLayer()*	13.4 - 1x		44.2 - 2x	116.2 - 8x	31.4 - 2x	51.3	51.3		4 of 4
	setLayer()*				20.5 - 2x		5.1	20.5		1 of 4
	setPosition()*					0.9 - 1x	0.2	0.0		0 of 4
	getActionCommand()*					41 - 1x	10.3	41.0		1 of 4
	isSelected()			3.2 - 1x		1.5 - 1x	1.2	2.4		2 of 4
	moveToFront()			5.2 - 1x			1.3	5.2		1 of 4
	moveToBack()						0.0	0.0		0 of 4
	getSelectedIndex()				4.7 - 1x	35.2 - 1x	10.0	20.0		2 of 4

Table 6.9: Timing data for successful subjects in control condition of Swing 1

Successful subjects in the control condition spent about a third of their time, on average, reading *JavaDocs*. An average of 44% of this read time was spent on the methods which would have been decorated for *eMoose* users in the core area, and 33% on decorated methods in other areas. Less than a quarter of the reading time is spent on undecorated methods, suggesting that the would-be decorated methods somehow attract attention even without visible decorations. Very little time was spent on elements which were not methods.

As we can see, subject S3 exhibited a very different behavior from his peers, having spent most of his four minutes of work examining the code without actually inspecting anything. Then, with “surgical precision”, he made a single read, that of `putLayer`, and fixed the problem. This subject was a Ph.D. student with significant programming experience and his skill may be an outlier. Although he was not very familiar with SWING, he appeared able to spot the `actionPerformed` method quickly, identify that the relevant condition involves the list box, and apparently suspect that something was amiss in how the icon was assigned to a label.

Our three other successful control subjects, S19, S23, and S26 took significantly more time and made more reading actions, for a greater portion of their work time. Subject S23 spent almost all his time on the decorated methods in the core area, and primarily on `putLayer`. He appeared to become aware of the directive, but spent time investigating the alternative by reading its web-based *JavaDocs*. Subject S26 got distracted by many decorated and undecorated methods and elements in the core area, and spent significant amounts of time on `getActionCommand` and `getSelectedIndex`. He also spent about 20 seconds on a completely unrelated method outside the core area, `setMouseMotionListener`. Subject S19 realized the importance of `actionPerformed` and focused primarily on `putLayer`. Earlier, however, he got bogged down for more than 3 minutes by the `invokeLater` method in `main` which merely activates the main thread of the program.

Behavior of unsuccessful subjects

We now turn to the unsuccessful subjects in the control condition, whose detailed data is presented in Table. 6.10, and try to understand what could have caused the failures.

	1	7	8	11	13	18	21	25					
	CTL	CTL	CTL	CTL	CTL	CTL	CTL	CTL					
	FAIL	FAIL	Fail	Fail	Fail	Fail	FAIL	FAIL					
	900	900	900	900	900	900	940	943	AVG	AVG >1	Portion	Users >1	
Aggregated	ConstantField	0.5 - 1x	0	0	0	0.4 - 1x	0	2 - 1x	0	0.4	2.0	0.2%	1 of 8
	Class	0.9 - 2x	1.6 - 1x	15.7 - 1x	1.4 - 1x	3.3 - 4x	0	0.6 - 1x	3.6 - 2x	3.4	4.4	2.0%	6 of 8
	Constant	4.8 - 4x	3.1 - 3x	8.6 - 4x	0.1 - 1x	7.7 - 8x	6.9 - 3x	0	0.4 - 1x	4.0	6.2	2.3%	5 of 8
	Variable	5.5 - 4x	20.3 - 3x	0.9 - 1x	1.8 - 3x	17.3 - 11x	1.5 - 1x	2.9 - 5x	13.7 - 2x	7.9	8.0	4.5%	8 of 8
	Fields	10.3 - 7x	9.7 - 4x	16.9 - 8x	34.2 - 12x	55.2 - 26x	26.4 - 10x	12.6 - 7x	6.9 - 7x	21.5	21.5	12.4%	8 of 8
	DeclarationMethod	2.1 - 2x	0	11.1 - 2x	0	2.2 - 1x	0	0	0	2.0	5.1	1.1%	3 of 8
	InternalMethod	0	0	15.6 - 4x	0	0	3.4 - 2x	1.7 - 1x	0	2.6	6.9	1.5%	3 of 8
	DecoratedCoreMethod	9.1 - 4x	219.9 - 10x	34.9 - 5x	30.3 - 8x	103.2 - 7x	66.7 - 9x	7.2 - 4x	5.7 - 3x	59.6	59.6	34.5%	8 of 8
	DecoratedOtherMethod	0.8 - 1x	0	35.4 - 2x	0	1 - 1x	0	0	0	4.7	18.2	2.7%	2 of 8
	UndecoratedCoreMethod	21.9 - 5x	38.4 - 6x	72.8 - 4x	65.6 - 22x	44.9 - 8x	17.3 - 6x	27.7 - 4x	129.2 - 12x	52.2	52.2	30.2%	8 of 8
UndecoratedOtherMethod	17.5 - 5x	0	46.5 - 11x	1.4 - 2x	0	0.7 - 1x	53.6 - 5x	0	15.0	29.9	8.6%	4 of 8	
Stats	Read time	73.8	293	258.4	134.8	235.2	122.9	108.3	159.5	173.0			
	Work time	900	900	900	900	900	900	940	943	910.4			
	Read/Work	8.20%	32.56%	28.71%	14.98%	26.13%	13.66%	11.52%	16.91%	19.1%			
	Ct. reads	35	27	42	49	67	32	28	27	38.4			
Core	ON_TOP_COMMAND	2.1 - 2x	2.8 - 2x	6.7 - 3x	0	4.5 - 4x	4.1 - 2x	0	0				
	LAYER_COMMAND	1.2 - 1x	0.3 - 1x	1.9 - 1x	0.1 - 1x	3.1 - 3x	2.8 - 1x	0	0.4 - 1x				
	position	1 - 1x	20.3 - 3x	0.9 - 1x	1 - 2x	16.4 - 10x	0	0.5 - 1x	13.7 - 2x				
	cmd	0.4 - 1x			0.8 - 1x	0	1.5 - 1x	0	0				
	dukelLabel	5.1 - 2x	4.9 - 2x	3.7 - 3x	19.3 - 7x	21.1 - 5x	1.1 - 2x	6.3 - 1x	3.3 - 3x				
	layeredPane	2.6 - 3x		4.5 - 3x		22.4 - 15x	1.9 - 2x	4.8 - 5x	0.9 - 1x				
	onTop	0.9 - 1x	3.6 - 1x	2.7 - 1x	5.2 - 2x	8.1 - 3x	7.1 - 4x	1.5 - 1x	2.7 - 3x				
	putLayer()*	8.5 - 3x	45.2 - 4x	27.1 - 4x	16.8 - 3x	61.8 - 3x	31.9 - 5x	0.5 - 1x	0	24.0	27.4		6 of 8
	setLayer()*									0.0	0.0		0 of 8
	setPosition()*	0.6 - 1x	140.1 - 4x		10.3 - 3x	41.4 - 4x	8.1 - 2x	0.7 - 1x	5.4 - 2x	25.8	41.0		5 of 8
	getActionCommand()*		34.6 - 2x	7.8 - 1x	3.2 - 2x	26.7 - 2x	6 - 2x	0.3 - 1x	9.8	15.7			5 of 8
	isSelected()	1 - 1x	5.5 - 2x		9.7 - 3x	33.8 - 4x	3.3 - 3x	26.4 - 3x	0.1 - 1x	10.0	13.3		6 OF 8
	moveToFront()	11.7 - 2x	11.5 - 2x	52.3 - 3x	1.6 - 4x		0.3 - 1x	1.3 - 1x	86.7 - 7x	20.7	27.6		6 OF 8
	moveToBack()	9.2 - 2x			9.7 - 1x	0.4 - 1x	12.1 - 1x		42.4 - 4x	9.2	18.5		4 OF 8
	getSelectedIndex()		21.4 - 2x	20.5 - 1x	44.6 - 14x	10.7 - 3x	1.6 - 1x			12.4	19.8		5 OF 8

Table 6.10: Timing data for unsuccessful subjects in control condition of Swing 1

Since success in this task depended on an awareness of the directive in `putLayer`, our attention turns first to this method. Surprisingly, 7 of the 8 unsuccessful subjects actually examined this method, and 6 examined it for a significant amount of time (more than 24 seconds on average). In other words, unlike our first task (the one based on JMS), the problem is not that subjects did not get to the documentation, but rather that they did not benefit from it. While 24 seconds is about half the time spent by the successful subjects, it should still have been sufficient to allow subjects to notice a directive that occupies most of

the two lines of documentation (see Fig. 6.26).

Like their successful peers, unsuccessful subjects appeared to realize the importance of `actionPerformed`, and spent relatively little time reading methods outside the core area, regardless of decoration status. However, they spent significantly more time examining other calls and elements in the core area. Consider the striking visual difference in the lower third of the table (“Core”) between the sparse access of successful subjects (Fig. 6.9) and the dense access of unsuccessful ones (Fig. 6.10). Every unsuccessful subject explored the majority of elements and calls in the core area; this accounts for the significant amount of time spent on undecorated core methods, and on fields in the “other” category. It is quite possible that the access to `putLayer` was “accidental” rather than intentional when subjects followed a systematic or random examination of everything in the `actionPerformed` function.

When we examine the functions investigated within the core area, we see that most of the unsuccessful subjects were distracted for a significant amount of time and multiple visits by one or two specific functions, but the identity of these functions differs from subject to subject. For example, subjects S8 and S25 spent a lot of time on the undecorated `moveToFront` and `moveToBack` combo, which are located in the other control path for `actionPerformed`. It is possible that these subjects did not fully understand the structure of the action handler. Subjects S7 and S13 spent significant amounts of time on `setPosition`, which presents a short but very perplexing documentation (Fig. 6.37). Subjects S8, S11 were captivated by `getSelectedIndex` while subject S13 focused on `isSelected`.

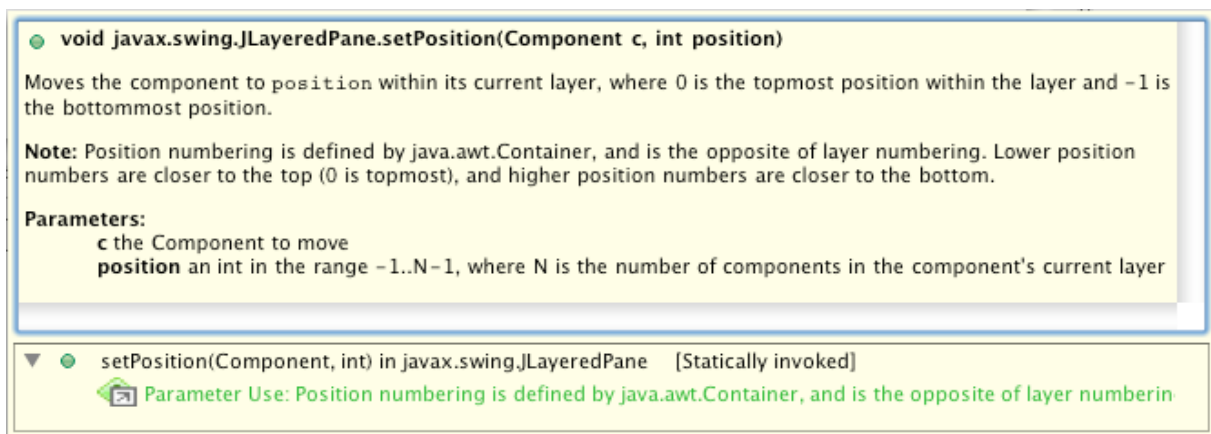


Figure 6.37: Javadocs for the `JLayeredPane.setPosition` method with *eMoose* directives

Unlike successful subjects, however, unsuccessful ones spent significant amount of time examining elements that are not calls, although most of these elements tended to be in `actionPerformed`. Much of the time in the `Other` category (about 20 seconds on average) was spent examining fields, and some time was spent on variables. Subjects appeared less interested in classes and constants.

6.6.5 Results - Experimental condition

All 12 subjects in the experimental condition were successful. We move to compare their behavior to that of the 5 successful control subjects and the 8 unsuccessful ones.

Aggregate time comparisons

Before we dive into the detailed timing data, let us first compare some aggregate data between the three groups. Tables 6.11 and 6.12 respectively present the proportion of reading time and the total reading

time spent on average on each of the categories. The average is calculated for each subject group and treats subjects who did not participate in a particular category as 0.

Proportion of reading time (AVG all)			
	CTL-PASS	CTL-FAIL	EXP-PASS
TOTAL DecoratedCoreMethod	44.8%	34.5%	57.1%
TOTAL DecoratedOtherMethod	33.2%	2.7%	24.3%
TOTAL UndecoratedCoreMethod	8.4%	30.2%	2.2%
TOTAL UndecoratedOtherMethod	7.0%	8.7%	7.8%
Total Other	6.7%	24.0%	8.6%

Table 6.11: Comparison of proportion of reading time in each category in first Swing task

Total reading time (AVG all)			
	CTL-PASS	CTL-FAIL	EXP-PASS
TOTAL DecoratedCoreMethod	66.8	59.6	42.2
TOTAL DecoratedOtherMethod	49.6	4.7	18.0
TOTAL UndecoratedCoreMethod	12.5	52.2	1.6
TOTAL UndecoratedOtherMethod	10.4	15.0	5.8
Total Other	10.0	41.6	6.7

Table 6.12: Comparison of total reading time in each category in first Swing task

As can be seen from the proportions table (Tab. 6.11), the behavior of subjects in the experimental condition is more similar to that of their successful peers in the control condition than to that of the unsuccessful ones. They spent most of their time reading decorated methods in the core area, and spent relatively little time reading undecorated methods or elements in the “Other” category. Note that while there is also an apparent similarity in the reading of decorated methods outside the core area, this may be misleading as the average for successful controls is biased by a single subject who spent a lot of time on such methods, while others barely spent any time. Therefore, only subjects in the experimental condition spent a significant portion of their time on decorated methods outside the core area. This could indicate a distraction caused by *eMoose*.

When we turn to the absolute times in Table. 6.12, we see that subjects in the experimental condition actually spent a lot less time reading the decorated core methods than their peers in the control condition regardless of success. They also spent significantly less time reading undecorated methods or elements in the “other category”. The time spent on decorated calls outside the core area is actually not that high, though still higher than for unsuccessful controls.

Detailed timing data

Now that we understand the major differences, it is time to examine the detailed timing data for subjects in the experimental condition, presented in Table. 6.13.

As we have seen in the aggregate tables, subjects in the experimental condition spent very little time with undecorated methods, leading to sparseness in the last four rows of the table.

Of the decorated methods in the core area, only `putLayer` was explored by everyone, and it accounted for most of the time spent on the core area. However, awareness and understanding of the directive in `putLayer` was not always immediate, as 7 of the 12 subjects needed more than one visit, and two needed more than a minute of total time. Because so few of the subjects in the control condition were successful, it is hard to determine whether they were slower or faster to identify the directive.

Interestingly, of the two other decorated methods in the core area, `setPosition` and `getActionCommand`,

		2	5	6	9	10	12	14	16	17	20	22	24				
		EXP	EXP	EXP	EXP	EXP	EXP	EXP	EXP	EXP	EXP	EXP	EXP	AVG	AVG >1	Portion	Users >1
		PASS	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	PASS	PASS	PASS				
Aggregated	ConstantField	0.2 - 1x	0	0	0	0	0	0	0.6 - 1x	0	0	0	0	0.6	0.0	0.8%	0 of 12
	Class	0	2.1 - 1x	0	0	0	0.3 - 1x	0	0	3.7 - 2x	0	0	29.3 - 3x	3.0	11.7	4.0%	2 of 12
	Constant	0.5 - 1x	0	0	0	0	1.6 - 2x	0	0	1 - 1x	2 - 1x	0	0	0.4	1.7	0.6%	3 of 12
	Variable	0	0	1.4 - 2x	0	0	0	0	1 - 3x	2.1 - 2x	2.1 - 2x	0.6 - 1x	0	0.6	1.8	0.7%	4 of 12
	Fields	3.1 - 2x	0	1.9 - 2x	0	0	0	0.9 - 1x	0.7 - 2x	2.8 - 3x	0.9 - 1x	1 - 2x	12.4 - 1x	2.0	4.2	2.4%	5 of 12
	DeclarationMethod	0	0	0	0	0	0	0	0	0	0	0	0.7 - 1x	0.1	0.0	0.1%	0 of 12
	InternalMethod	0.1 - 1x	0	0	0	0	0.3 - 1x	0	0.2 - 1x	0	0	0	0	0.1	0.1	0.1%	0 of 12
	DecoratedCoreMethod	16.9 - 3x	101.3 - 6x	34.7 - 2x	22.4 - 2x	10.6 - 2x	5.8 - 1x	58.4 - 2x	93.2 - 8x	32 - 2x	25.7 - 2x	68 - 1x	37.1 - 5x	42.2	42.2	57.1%	12 of 12
	DecoratedOtherMethod	37.4 - 1x	12 - 1x	0	0	37.6 - 3x	0	10 - 1x	75.4 - 8x	15.9 - 3x	14.9 - 2x	0	12.2 - 2x	18.0	26.9	24.3%	8 of 12
	UndecoratedCoreMethod	0	1.6 - 1x	1.4 - 1x	0	0	0	0	10.3 - 3x	0	0.7 - 1x	2.1 - 2x	3.4 - 1x	1.6	3.9	2.2%	5 of 12
UndecoratedOtherMethod	13.7 - 2x	0	0	0	1.4 - 1x	0	18.9 - 3x	19.3 - 3x	13.7 - 6x	2.3 - 2x	0	0	5.8	11.6	7.8%	6 of 12	
Stats	Read time	71.9	119	39.4	22.4	49.6	7.7	88.2	200.7	71.2	48.6	71.7	95.1	73.8			
	Work time	480	450	105	161	321	90	104.2	510	335	214	178	480	128.3			
	Read/Work	14.98%	26.44%	37.52%	13.91%	15.45%	8.56%	84.64%	39.35%	21.25%	22.71%	40.28%	19.81%	28.7%			
	Ct. reads	11	9	7	2	6	5	6	29	19	11	6	13	10.3			
Core	ON_TOP_COMMAND	0.5 - 1x					0.5 - 1x			1 - 1x	2 - 1x						
	LAYER_COMMAND						1.1 - 1x										
	position			0.2 - 1x					0.2 - 1x	2 - 1x	1 - 1x						
	cmd			1.2 - 1x						0.1 - 1x		0.6 - 1x					
	dukeLabel	2.5 - 1x		0.7 - 1x					0.5 - 1x	0.5 - 1x		1 - 2x					
	layeredPane	0.6 - 1x								1 - 1x	0.9 - 1x						
	onTop			1.2 - 1x				0.9 - 1x					12.4 - 1x				
	putLayer()*	8.2 - 2x	76.3 - 4x	34.7 - 2x	22.4 - 2x	8.2 - 1x	5.8 - 1x	51.2 - 1x	64.4 - 6x	32 - 2x	18.7 - 1x	68 - 1x	14.2 - 3x	33.7	33.7		12 of 12
	setLayer()*													0.0	0.0		0 OF 12
	setPosition()*	8.7 - 1x	25 - 2x			2.4 - 1x		7.2 - 1x	28.8 - 2x		7 - 1x		17.5 - 2x	8.1	13.8		7 OF 12
	getActionCommand()*													0.0	0.0		0 OF 12
	isSelected()		1.6 - 1x								0.7 - 1x	2.1 - 2x		0.4	1.5		3 OF 12
	moveToFront()								10 - 2x					0.8	10.0		1 OF 12
moveToBack()								0.3 - 1x					0.0	0.0		0 OF 12	
getSelectedIndex()			1.40 - 1x									3.4 - 1x	0.4	2.4		2 OF 12	

Table 6.13: Timing data for successful subjects in experimental condition of Swing 1

only the former was explored, and by only half the subjects. Something led everyone in the experimental condition to avoid exploring the latter even though it was heavily explored in the control condition.

Investigation of decorated methods outside core area

This task was different from the other tasks in our study in that it involved the code of an entire program, and thus offered ample opportunities for distraction. An important question is whether *eMoose* users were distracted by decorated calls outside `actionPerformed` more than their peers in the control condition.

The data shows that 8 of 12 subjects in the experimental condition explored at least one decorated method outside the core area, compared to 4 of 12 controls. In that sense, more *eMoose* users were distracted. We examine the specific methods to understand the source of the differences.

Outside of `actionPerformed`, there were a total of five distinct invoked methods which were decorated (some were invoked multiple times). The method `setBackground` was explored by a single *eMoose* user. The method `setBorder` was explored by two controls and three *eMoose* users. The method `setDefaultCloseOperation` was briefly explored by a single control and two *eMoose* users. The method `invokeLater` was explored for a long time by one control, and briefly by two *eMoose* users. Finally, the `add` method (on various widget containers) was explored for moderately long periods by a single control and four *eMoose* users.

The first four functions are clearly not related to the problem, and were generally explored for short periods. While it seems that *eMoose* users were slightly more likely to read them, there is no clear pattern. The last method `add` method received more attention and was clearly explored more frequently by *eMoose* users, although they were still a minority from among the participants.

6.6.6 Discussion

Unlike the previous two tasks which involved short code segments, our third task more closely resembles a real-world software maintenance problem. It involves a sizable and non-trivial code base which cannot practically be approached with a systematic brute-force investigation. Identifying and isolating the bug requires subjects to understand and map complex GUI behavior into actual source code statements.

Explaining failure to notice directive in `putLayer`

The fact that all subjects were able to identify and focus on the relatively short `actionPerformed` method is testimony to their ability to understand, at least superficially, the structure of the program and its control flow. However, the low success rate (5 of 13) among controls is alarming, as it indicates that even exposure to an explicit directive in a short *JavaDoc* does not guarantee absorption and understanding. Unlike the first task where many subjects did not even find the relevant method or the second task where the directive was hidden deep inside the text, the `putLayer` method was explored by almost everyone and the directive occupied most of its text. This leads us to question why *eMoose* users tended to notice the directive while controls did not?

I suspect that the difference in the prospects of noticing and recognizing the importance of the directive in `putLayer` are related to the level of attention paid to that specific call.

As previously hinted, I suspect that the presence of a decoration lends the call an “aura” of importance. The fact that the directive is repeated via the lower pane may further increase this perception. In this case, however, the text was actually somewhat misleading as it didn’t mention what the “desired side effects” would be, requiring subjects to read the original text.

A more central factor, however, could be that the reading of other elements in the proximity of `putLayer`, either systematically or randomly, reduced the attention paid to that method. In other words, by reading so many *JavaDocs* in the `actionPerformed` area, unsuccessful controls may have been exposed to the important directive in `putLayer` but failed to recognize it as a directive or as a more important issue than the information they absorbed from the other elements. For example, a developer perplexed by the complex definition of positions for `setPosition` could have failed to pay sufficient attention to the issue of side effects. Since many controls investigated so many elements in `actionPerformed`, the the reading of `putLayer` might have been “accidental”. I believe that developers may pay more attention to a method’s documentation when it stands out among other calls and a conscious and deliberate decision to explore it specifically is taken.

If the above conjectures are the explanation for the difference, then they may indicate additional mechanisms of impact for *eMoose* beyond drawing subjects to a specific call and letting them quickly identify its directives. First, the the presence of a decoration may increase the perceived reliability of the information it conveys. Second, the absence of decorations on other methods indirectly increases the attention given to decorated ones. While these effects are positive, they carry significant risks if the set of tagged directives is incomplete or inaccurate.

Even if the above conjectures are correct and explain why the directive was not immediately apparent, it is not clear why subjects did not identify it even after repeated readings. My impression from hearing subjects subvocalize and from talking to them is that some focused only on the first sentence and did not read beyond it. Others read the next sentence but did not necessarily connect the bug to refreshing because they may have attributed it to a logic problem. When they decided that the side-effect issue was not relevant, they may have stopped reading and never noticed the subsequent explicit instruction about using `setLayer`.

Distraction in the control group

The behavior of subjects in the control group demonstrates that developers attempting to understand or debug a program can be significantly distracted by methods even without the “help” of *eMoose* decorations. The reasons for these distractions are not always clear though they seem to be associated a perception that a particular call could conceivably be related to the error.

It is likely that in many cases subjects focused for long periods of time on seemingly unrelated

methods simply because they misunderstood the mechanisms of the program or of the error. For example, of the undecorated calls outside the core area, the one explored for the longest time was actually `addMouseMotionListener`. which was explored for about 20 seconds by two controls and two subjects in the experimental condition. Even though the bug stems from an error in `actionPerformed` which is invoked when the user interacts with the list boxes, it is only manifested when users move the mouse,. This may have lead subjects to explore the point closest to the point of error. The next most explored undecorated method, `addActionListener`, may have been explored when subjects began attributing the error to interaction with the list boxes, but before finding out that the handler is in `actionPerformed`.

Of course, the amount of distraction and actual exploration choices may depend on familiarity with the API. Experienced SWING developers are likely to be quite familiar with `addActionListener`, and are less likely to investigate it.

Avoiding distraction in the *eMoose* group

One of the potential risks of using *eMoose* on large code bases is the that of significant distraction that could be brought on by decorated methods. To confirm that such distractions exist, it is not enough to show that *eMoose* users were spending significant time on decorated methods. Rather, it is necessary to show that it happens more frequently than for subjects who were not exposed to the decorations. Unfortunately, determining this is difficult with the limited sample size that we have.

As we have seen, subjects in the control condition spent significant time exploring everything in the `actionPerformed` method, including undecorated methods but also many of the the decorated methods such as `setPosition` and `getActionCommand`. Subjects in the experimental condition, on the other hand, spent very little time on the undecorated methods, which would offset the cost of distraction by decorated methods. In addition, at least within `actionPerformed` they also spent less time on the decorated methods except for `putLayer`.

It is also interesting that while many subjects in the experimental condition spent time on the perplexing `setPosition` method, nobody spent time on `getActionCommand`. This latter method was used to obtain the identity of the UI widget with which the user interacted, and serves as the basis for the switch statement. By not examining it despite the presence of decorations, subjects showed that they were capable of applying discretion and an evaluation of a call's potential relevance prior to making an exploration decision.

Outside `actionPerformed`, we saw that *eMoose* users were slightly more likely to explore the unrelated decorated methods. However, they seemed significantly more likely to explore `add` although they were still a minority. This focus is perhaps not surprising because this method could conceivably have been the source of the problem and is therefore an important avenue for exploration. As can be seen in Fig. 6.38, its documentation states that changes may not be seen until the `validate` method is called. While the implementation in this program merely reassigns the icon to a layer with `putLayer`, it could have also changed the display by removing and reinserting the object. In fact, this directive is the basis for our second SWING-based task.

Nevertheless, we see that subjects are not compelled to investigate every decorated call and were not significantly more distracted than their peers. I believe that these results offer evidence that developers can cope with a large number of decorated methods over large code ranges without having to explore all of them.

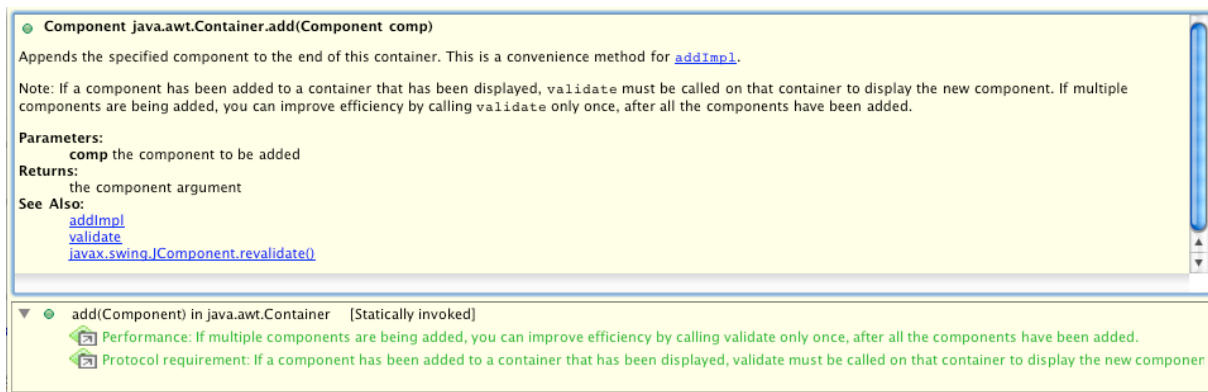


Figure 6.38: Javadocs for the `Container.add` method in AWT with *eMoose* directives

Examining non-methods

As we have found in the previous tasks, subjects spent significant amounts of time examining via the hover mechanisms things that are not method calls and which may not have *JavaDocs*. This behavior was quite noticeable in this task, but surprisingly affected mostly unsuccessful subjects in the control condition.

One explanation is that these subjects might have realized the importance of `actionPerformed`, and then systematically or randomly examined every element in it, even if that element was a field or a constant.

However, I have also seen frequent examinations of elements outside this function. The fact that such examinations focused on fields and variables suggests that the developers may have been trying to use the debugger's variable inspection feature. Note that unlike the previous tasks, where the use of the debugger was prohibited to avoid pinpointing the exact point of failure in a small code section, subjects in this task were allowed to run the program and even to use the debugger, as it does not directly reveal the point of failure. Many subjects did so, placing breakpoints and using the hover to examine variables.

6.7 Second Swing Task

The first three tasks aimed to evaluate the decisions that developers make when confronted with decorated methods in existing code. The fourth task aimed to investigate whether developers investigate and benefit from the highlighting of newly-added calls. Recall that at present *eMoose* does not augment the code-completion mechanism, so a developer is made aware of a directive in a newly-added call via a decoration that appears several seconds after its addition.

This task builds upon the code base from the previous SWING task, and asked subjects to write the code for an empty function. My expectation was that in doing so they will make use of the `add` method, whose documentation appears in Fig. 6.38, and fail to become aware of the need to call `validate` to obtain a refresh behavior.

Unfortunately, subjects who were not previously familiar with SWING struggled to write new code using this API, and in many cases did not even make the call to `add`. In addition, there was a risk that some subjects would have encountered this call while working on the first SWING tasks, or that they would already be familiar with it if they had SWING experience. As a result, halfway through the study I decided to scrap this task. I leave here the description of the task since it could be useful if the experiment

is ever carried out with more experienced SWING developers.

6.7.1 Directive for this task

The fourth directive comes from the standard `Container` class in AWT, and is thus inherited by SWING's `Component`, and `JLayeredPane`. Objects are added to containers and components using the `add` method and removed using the `remove` method. As can be seen in the documentation in Fig. 6.38, these methods indicate that to cause the actual refresh, one must explicitly call `validate`. Failure to do so may cause actions to have no visible effects until something else, such as a window resize, causes the container to be repainted.

6.7.2 Task Description

The codebase for this task has been modified from the official `LayeredPaneDemo` class used in the previous task. The new class, called `LayeredPaneDemoDual` now contains two layered panes that appear in a vertical order below the control panel. They are assigned to the fields `topPane` and `bottomPane`. In the top pane, the `Duke` icon always appears “on top” of the current layer, as if the checkbox is checked. In the bottom pane, he appears below everything in the current layer, as if the checkbox is unchecked.

The revised control panel no longer includes the checkbox. Instead, there are three new checkboxes. The first two boxes controls whether each of the panes appears. The third box controls whether the entire control pane appears before (higher than) the layered panes or below it. By interacting with these boxes, users should be able to hide one or both of the panes, and move the control panel if at least one pane is visible. However, that functionality is not implemented, and the event handler simply calls a function that is supposed to update the view. The subjects are taken to this function, which has an empty body, and asked to implement it.

A straightforward way to implement this function in about 10 lines of code is to first remove all components, then add them in the appropriate order based on the checkboxes, calling `validate` at the end. Users were expected to write the code but probably miss the call to `validate`, leading to failures that they would fix by immediately or eventually investigating `add`. My plans were to measure the time until this was noticed. Unfortunately, users who were not familiar with SWING did not attempt a removal and addition based solution. Rather, most looked for a solution that involved reordering the elements, or changing the visibility.

6.8 First Collections Task

Liskov's principle of substitution [59, 66] states that the behavior of a subtype must conform to that of the supertype so that it can safely be used in its place. For example, if a method can operate on a certain parameter value, any overriding version should do the same, although it could potentially support additional values. Unfortunately, substantial conformance violations exist even in well-respected APIs. In addition, some overridden versions may not be violating their contract in the strict sense, but may introduce caveats such as performance issues or side effects. A difference in implementation may be accompanied by different documentation, and thus a different set of directives.

One of the goals of this study was to evaluate the directive awareness challenges presented by polymorphic code in which the documentation of an overriding version of a method conflicts with the overridden. The goal of the first collections task is to examine the developers' ability to become aware of

a conflicting directive when they do not expect it, and the second task will examine this when they do. This behavior will then be compared to that of *eMoose* users.

The challenges of directive awareness in polymorphism

Before turning to the task, it is important to understand the nature of the challenge to directive awareness presented by polymorphic code.

In JAVA, the documentation of a method is inherited by an overriding versions unless it specifies its own version. In the same way that the code of an overriding method can violate the behavior of the overridden, the documentation of an overridden method can conflict with that of the overridden, or even just add to it.

In most IDEs, however, the documentation presented via the hover mechanism corresponds to the static (declared) type of the object on which the method is invoked. In such situations, a developer confronted with a call to a method which can be overridden will explicitly have to seek the documentation of that version. As described below, doing so in a consistent manner can be a tedious and time-consuming process.

First, the developer must become aware of the fact that the variable on which the call is invoked, and which is declared with a certain static type, may contain a different dynamic type at runtime. While this is trivial for very familiar types like standard JAVA collections, in many cases the developer may explicitly need to check the static type by hovering over it or examining its declaration. If the declared type is an interface or an abstract class (which cannot be instantiated), then it is clear that at runtime the variable will contain a different concrete dynamic type. The greatest challenge, however, is when the type is concrete but could potentially be subtyped and its methods overridden. In those situations, the dynamic type could be identical to the static type, but there is a chance that it is not.

Once the developer has become aware of the possibility of different dynamic types, he must identify all of them. The *Eclipse* IDE offers a convenient tree presentation for the subtypes of a given type; however, this tree can contain duplicates. While JAVA does not support multiple inheritance of classes, it allows a class to implement multiple interfaces. Thus, while class hierarchies have the shape of a tree, the typing graph has a lattice structure. The developer must examine the entire tree and identify duplicates. Another option is to use the subtypes list in the *JavaDoc*, but this list is not transitive, thus requiring a tedious scan of the lattice. In any case, once the set of subtypes have been collected, it must be further filtered to contain only concrete types which can actually be instantiated as the dynamic type.

Once all possible dynamic types have been identified, one must proceed to identify the exact version of the method that would be invoked for each of them. If the method is defined in that specific type, then this is straightforward since it is now the “lowest overriding version”. Otherwise, however, one must trace up the lattice following class inheritance nodes until an overriding versions is found.

It is important to note that while given a dynamic type it is always possible to determine which version of the method would be invoked, there are situations where it is not clear which documentation should apply. One of the reasons that JAVA, unlike C ++, forbids multiple inheritance of classes is to avoid the difficulties in determining what code should execute when the inheritance takes a diamond form or simply when the method is inherited from two parents. With interfaces there is no such problem because they consist only of method declarations rather than actual definitions. However, since interface methods are documented, it is possible for a method to inherit several versions with conflicting documentation, with no way to determine which one should apply.

6.8.1 API and directive for this task

To evaluate directive awareness, I chose to have subjects work against a very familiar static type but with a less-familiar subtype that presents conflicting directives. As the static type, I chose to use the `Collection` interface from the `java.util` package of the standard library. This interface is the base for all specific collection types in the standard libraries and in supporting collections libraries such as the *apache-collections* library.

Despite its central status in the language, the collections framework contains conformance violations that are very familiar to most programmers. For example, the interfaces `set` and `map` have `add` and `put` methods that can potentially accept everything. However, if the chosen implementation is a `TreeSet` or `TreeMap`, and the added object does not implement the `Comparable` interface, an exception will be thrown at runtime, a clear conformance violation. To prepare subjects for these sort of violations, I used this example during the initial *eMoose* tutorial, and showed how *eMoose* will present the decoration and the directives for the overriding version.

Two characteristics that can be used to broadly distinguish between collections are the handling of element ordering and of duplicates. For example, a linked list or array preserves order and duplicates, while a set keeps neither.

Our task is focused specifically on two methods: `containsAll` and `retainAll`. Both are invoked on a collection (to which we shall refer as the left-hand-side LHS) and receive a second collection (the right-hand-side RHS). As can be seen in Fig. 6.39, `containsAll` returns `true` if LHS contains all of the elements in RHS. As can be seen in Fig. 6.40, `retainAll` removes all elements from LHS that are not contained in RHS.

```
● boolean java.util.Collection.containsAll(Collection c)
Returns true if this collection contains all of the elements in the specified collection.
Parameters:
  c collection to be checked for containment in this collection.
Returns:
  true if this collection contains all of the elements in the specified collection
Throws:
  ClassCastException - if the types of one or more elements in the specified collection are incompatible with this collection (optional).
  NullPointerException - if the specified collection contains one or more null elements and this collection does not support null elements (optional).
  NullPointerException - if the specified collection is null.
See Also:
  contains(Object)
```

Figure 6.39: Javadocs for the `Collection.containsAll` in standard JAVA library

```
● boolean java.util.Collection.retainAll(Collection c)
Retains only the elements in this collection that are contained in the specified collection (optional operation). In other words, removes from this collection all of its elements that are not contained in the specified collection.
Parameters:
  c elements to be retained in this collection.
Returns:
  true if this collection changed as a result of the call
Throws:
  UnsupportedOperationException - if the retainAll method is not supported by this Collection.
  ClassCastException - if the types of one or more elements in this collection are incompatible with the specified collection (optional).
  NullPointerException - if this collection contains one or more null elements and the specified collection does not support null elements (optional).
  NullPointerException - if the specified collection is null.
See Also:
  remove(Object)
  contains(Object)
```

Figure 6.40: Javadocs for the `Collection.retainAll` in standard JAVA library

Interestingly, the documentation of neither of these functions states how cardinality (the number of

copies of identical objects) is treated. For example, if one collection contains X instances of an object and the other contains Y , what constitutes containment and how many are retained?

All collections in the standard library extend the abstract `AbstractCollection` class, which includes the implementation of many higher-level convenience methods. As a result, the implementations in `AbstractCollection` can be considered as the default behavior of the `Collection` interface. The implementations of `containsAll` and `retainAll` in `AbstractCollection` provide a different documentation from that of `Collection` and directly address cardinality.

The documentation of `containsAll` in `AbstractCollection` states:

Returns true if this collection contains all of the elements in the specified collection. This implementation iterates over the specified collection, checking each element returned by the iterator in turn to see if it's contained in this collection. If all elements are so contained true is returned, otherwise false.

In other words, the default behavior is that every element in RHS is separately checked against LHS. Thus, if every unique element in RHS has at least one copy in LHS, containment exists.

The documentation of `retainAll` in `AbstractCollection` states:

Retains only the elements in this collection that are contained in the specified collection (optional operation). In other words, removes from this collection all of its elements that are not contained in the specified collection.

This implementation iterates over this collection, checking each element returned by the iterator in turn to see if it's contained in the specified collection. If it's not so contained, it's removed from this collection with the iterator's `remove` method.

Note that this implementation will throw an `UnsupportedOperationException` if the iterator returned by the iterator method does not implement the `remove` method and this collection contains one or more elements not present in the specified collection.

In other words, the default behavior is that every element in LHS is separately checked against RHS and if it does not exist at least once in RHS it will be removed.

The above behaviors are followed by all collections in the standard JAVA library. Because the breadth of this library is limited, many projects in the Apache foundation make use of a project called `Apache Commons`, which includes a smaller library called *apache-collections*. The `Apache Commons` libraries are widely used by developers. Among the many collections in *apache-collections* is a collection representing a `Bag` data structure. A bag is a collection that respects cardinality but ignores order. In other words, it counts the number of instances for each element.

Because `Bag` is an interface rather than a class, it does not extend `AbstractCollection` but rather directly extends the `Collection` interface. Its class documentation states:

NOTE: This interface violates the `Collection` contract. The behavior specified in many of these methods is not the same as the behavior specified by `Collection`. The noncompliant methods are clearly marked with "(Violation)". Exercise caution when using a bag as a `Collection`. This violation resulted from the original specification of this interface. In an ideal world, the interface would be changed to fix the problems, however it has been decided to maintain backwards compatibility instead.

While this decision is understandable, it would not be known to users who do not explicitly choose to investigate the interface and could have unexpected consequences. Indeed, as can be seen in Fig. 6.41,

the `containsAll` method of `Bag` does respect cardinality. Its documentation states that if LHS is a bag (and thus this method is activated), and RHS contains n copies of some value x , then the bag must contain at least n copies too for `containsAll` to be true. Note that this is not symmetric, since if `containsAll` is invoked on another type of collection and passed a `Bag`, the version from `AbstractCollection` will be invoked. The documentation also explicitly states that this violates the description from `Collection` which does respect cardinality. This is actually an error, since this is only specified in `AbstractCollection`.

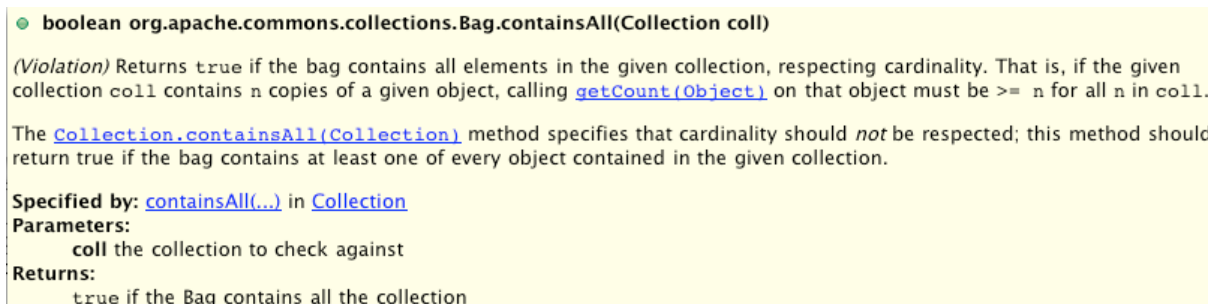


Figure 6.41: Javadocs for the `Bag.containsAll` in `apache-collections`

The description for `retainAll`, shown in Fig. 6.42 is similar. Cardinality is respected once again, meaning that if the LHS bag has more copies of an element than the RHS collection, the difference will be eliminated.

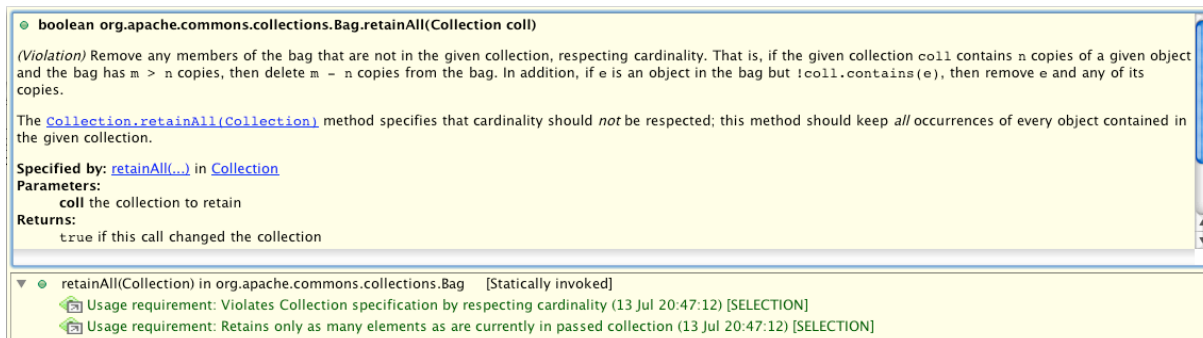


Figure 6.42: Javadocs for the `Bag.retainAll` in `apache-collections`

The difference in treating cardinality raises the risk of program error. If the user owns a reference to a `Collection` object that actually contains a `Bag`, the results of invoking `containsAll` and `retainAll` can therefore be unexpected.

For instance, suppose that we have a bag containing two copies of x , a list containing three copies of x , and a set containing a single instance of x . Calls on collections that are not the bag, such as `list.containsAll(bag)` or `list.containsAll(set)`, will return `true` since every object in the bag or set appears at least once in the LHS collection. Similarly, calls to `retainAll` would not change the LHS collection since there is at least one instance of every value.

However, if the LHS is the bag, things are different. The call `bag.containsAll(set)` would return `true` since the bag would have more instances of x than the set. On the other hand, the call to `bag.containsAll(list)` would return `false` since it has fewer instances of x . Similarly, the call `bag.retainAll(set)` would actually remove one copy of x from the bag since there are fewer copies in the set.

As directives for this task, I chose the warnings about cardinality in the overriding version of `containsAll`

and `retainAll` in `Bag`. This directive is difficult to classify, but can be thought of as a limitation since it informs users that the behavior in certain cases may not conform to expectations.

6.8.2 Task Description

Unlike previous tasks in this study which used existing sample code, the code base for this task was created from scratch with a goal of placing subjects in the situation where an awareness and exploration of the subtype would be necessary.

```
package edu.cmu.contextstudy.apachecommons.testsuite;

import java.util.Collection;
import java.util.HashSet;
import java.util.Vector;

import org.apache.commons.collections.buffer.PriorityBuffer;
import org.apache.commons.collections.list.TreeList;

import org.apache.commons.collections.bag.HashBag;

import junit.framework.Assert;
import junit.framework.TestCase;

@SuppressWarnings("unchecked")
public class StandardCollectionsTest extends TestCase
{
    final int SIZE = 200, RANGE = 10;

    Collection collections[] = null;

    public void setUp()
    {
        this.collections = new Collection[] { new HashSet(), new PriorityBuffer(), new TreeList(), new HashBag(), new Vector() };
        for(int i=0; i<SIZE; ++i) {
            int num = (int)(Math.random()*RANGE);

            for(Collection collection: collections) {
                collection.add(num);
            }
        }
    }

    public void testMutualRetainment() throws Exception
    {
        for(Collection c1: collections) {
            for(Collection c2: collections) {
                if(c1==c2) continue;

                Assert.assertTrue(c1.containsAll(c2) && c2.containsAll(c1));

                c1.retainAll(c2);
                c2.retainAll(c1);

                Assert.assertTrue(c1.containsAll(c2) && c2.containsAll(c1));
            }
        }
    }
}
```

Figure 6.43: Source code for first collections task

Fig. 6.43 presents the entire code file used in this task. The code defines two constants: `SIZE`, which is made to be very large, and `RANGE`, which is meant to be smaller by an order of magnitude. The `setUp` function initializes the `collections` array field with a group of newly created collections. It then randomizes `SIZE` numbers, each within `RANGE`, and adds them using the `add` method to each collection. Note that since the range is smaller than the size, the existence of duplicates is guaranteed.

The collections themselves are a blend of very familiar ones (`HashSet` and `Vector`) and less familiar ones. `PriorityBuffer` is essentially a heap implementation from *apache-collections*. A `TreeList` is an *apache-collections* list that has fast insertions and removals. A `HashBag` is an implementation of the `Bag` interface, and uses hashing to look up specific values.

The heart of the code is the `testMutualRetainment` function. This function essentially runs over every permutation of two of the collections. In each iteration, it checks whether both contain each other. It then calls `retainAll` on both in both directions, and runs the assert again. Subjects are told that this fails, and are asked to find why without using the debugger. Note that they are only asked to demonstrate the mechanism of failure, not to actually fix it.

Under the standard collection contract (or more correctly, under that of `AbstractCollection`), cardinality is ignored. Since the same value was added to every collection and appears in it at least one, calls to `containsAll` should always return true, and calls to `retainAll` should never change the collections. What breaks this, however, is the presence of the bag. At some point, the bag will serve as LHS while the set would be RHS, so there will be some value that appears more than once in the bag. When we call `retainAll`, the implementation in bag would remove all but one instance of that value from the bag, making it identical to the set. Later on, when we run `containsAll` of the bag as LHS and one of the other structures as the RHS, the bag would have less instances of the value than RHS, and the call would return `false`, leading to a failure of the first assertion statement.

Goals

This task is different from previous one in that we are not comparing the success rates between subjects. I do not necessarily expect subjects to successfully solve this problem and identify the bug, since the error mechanism is complex. Rather, my goal is mainly to observe how subjects in the control condition would handle a situation where they observe an unexpected behavior. Specifically, I seek to determine whether these subjects would explore possible overriding versions, and whether they would be successful in identifying the directives for `containsAll` and `retainAll` in `Bag`.

Since I am only checking whether subjects became aware of these directives, I will not be analyzing the behavior of *eMoose* users, as they were exposed to the directive as soon as they hovered over the calls to `containsAll` and `retainAll`.

6.8.3 Results

Twelve subjects attempted to perform this task in the control condition: S1, S5, S6, S8, S11, S13, S15, S16, S17, S23, S24, and S25. Of these twelve, only one subject, S6, fully succeeded in completing the task and identifying the exact scenario of failure.

The focus, however, is not on success in the task, but rather on awareness of the directive. How many subjects were exposed to the documentation of `containsAll` or `retainAll` in the control condition?

To answer this question and better understand the actions of the subjects, I created transcripts, counted the time spent on different elements, and aggregated it. Unfortunately, three recordings were corrupted: S1, S24 and S25. This leaves us with 8 unsuccessful subjects and one successful subject.

The tabulations are presented in Table. 6.14. They are organized into three groups: Elements that are present in the `setup` function and whose documentation can be read through it, elements that are present and readable via the `TestMutualRetainment` function, and elements that are not present in the original source code. To assist the reader, the table highlights in bold fonts every element whose reading required more than a simple hover in the existing code.

The first question that we have to ask is how many subjects explored the web-based documentation. It turns out that all but subject S13 used the web based *JavaDocs*, even though the same information could be accessed via the IDE.

A second question is whether those who utilized the web-based based documentation examined any

			S5 Fail	S6 Success	S8 Fail	S11 Fail	S13 Fail	S16 Fail	S17 Fail	S23 Fail	S25 Fail	
V i s i b l e i n s e t u p ()	HashSet	HashSet class in code	48 - 4x	8		4.2		13.2 - 2x				
	PriorityBuffer	Priority buffer in code	15.7 - 2x	5.5				22.8				
	TreeList	TreeList class in code			1.3		12.8		1.7			
		TreeList on the web									11.6	
		TreeList.add() on the web									20.2	
	HashBag	HashBag class in code		3.1 - 2x	1		23.9		4.3			
		Hashbag on web							21.9		6.4	56.9 - 2x
	Vector	Vector class in code		5.5	2.8		5.6		23.4 - 2x			
		Vector on the web							9.2			
		Vector.containsAll() on the web							23.8			
Vector.add() on the web								16.2				
Other	SIZE constant					5.6						
	RANGE					1.2		0.3				
	Math.random()					24.5 - 2x		40.4	0.8			
V i s i b l e i n t e s t ()	Collection	Collection class in code	4.9 - 2x		1.3		0.4					
		Collection on web	92.4								15.9 - 3x	
		Collection.containsAll() in code	18.2 - 1x	33 - 5x	198.4 - 5x	104.2 - 8x		3.4	149.3 - 3x		52.6 - 4x	
		Collection.containsAll() on web						31.7				
		Collection.retainAll() in code	12.5 - 3x	64.4 - 3x	75.2	175.4 - 6x	56			221.1 - 6x		
		Collection.retainAll() on web	117.5 - 1x									
		Collection.add() in code						67.9	30.3 - 2x			
	Collection web various			145 - 1x								
	Variables and constants	collections variable					1.4 - 2x		3.9			
		c1					2.5		1.8			
equals								0.7		42.5		
Asserts	assert			45 - 3x						26.2 - 2x		
	assertTrue	6.9 - 1x		10.4		2.2		0.3	1	9.7 - 1x		
	assertFalse							2.1	??			
B a g s	Bag	Bag class on web		27.6				4.4			115	
		Bag package web				12.5					33.3 - 2x	
		Bag.containsAll() web							291.2 - 3x		249.2	
		Bag.retainAll() web		4.7								
		Bag.getCount() on web									16.7	
		Bag.add() on web							94 - 3x		3.3	
	Bag.uniqueSet() on web									8		
AbstractMapBag	AbstractMapBag.add() web								18.8			
TreeBag	TreeBag on the web	14	1									
AbsCols	AbstractCollection	AbstractCollection.containsAll web							25.4			
		AbstractCollection.contains web							20.7			
		AbstractCollection.retainAll web							34.4			
		AbstractCollection.remove web							19.4 - 2x			
		AbstractCollection.add web							24.7			

Table 6.14: Timing data for controls in first collections task

of the subtypes of the collections listed in the code, as this may indicate that they might have been considering an overridden version. With the exception of subject S8, who only used the web-based documentation for the `Collection` interface, this was generally the case. Specifically, subject S5 spent 14 seconds on `TreeBag`, one of the concrete implementations of the `Bag` interface, to which he got almost randomly when looking at the list of classes in the environment. He did not examine `Bag`. Similarly, subject S23 spent 18 seconds exploring `AbstractMapBag`, which is another subtype of `Bag`, but did not explore `Bag`. Subject S17 devoted his attention to `AbstractCollection`, the topmost base class for all collections. Interestingly, this class is the only one which lists the standard policy about cardinality in the collections framework, as the methods in the interface do not. All other subjects, S6, S11, S16 and S25, explored at least one location related to the `Bag` interface, although S11 merely explored the package.

This finally brings us to the `Bag` interface. Only three of the nine subjects for which we have data were exposed to either `containsAll` or `retainAll` in `Bag` and thus had a chance of becoming aware of the directive. If we treat such exposure as our measure of success, then we have a 33% success rate. Subject S6, who was the only one to actually identify the problem, spent relatively little time (about 5 seconds) on `retainAll`) but spent nearly 30 seconds on the introduction text for the `Bag` class, which details the cardinality issue in depth.

It is interesting that not all subjects examined `containsAll` and `retainAll` on the `Collection` class in the given code. Subject S13 examined `retainAll` but not `containsAll`, while subject S16 (who did explore `Bag`) and S23 explored `containsAll` but not `retainAll`. Most notably, subject S25 did not explore either in `Collection` but spent considerable time on `Bag`.

6.8.4 Discussion

The results above, although based on a limited sample of 9 controls, show that developers face difficulties in becoming aware of directives in overridden methods. This task was built in such a way that the subjects would essentially be examining calls on a single interface for which they know that many implementations exist. They should therefore suspect that inconsistencies may be due to inheritance. Nevertheless, some subjects did not appear to consider this possibility, while others faced challenges in searching for potential conflicts. I argue that in a situation where subjects are not expected to focus on a single call, the chances of becoming aware of a conflict might be even lower.

The fact that many subjects turned to the web based documentation and explored subtypes shows that at least in this case they were willing to invest time in the search, although the search was not necessarily fruitful. Since the failure was attributed to a higher type than the concrete object instantiated in the program, subjects searched for the relevant types, and sometimes headed in the wrong directions.

The fact that subjects relied on web-based documentation suggests that current support via the IDE is lacking when supporting the exploration of documentation that is not directly associated with code entity. *Eclipse*, for example, offers a separate `Java browsing` perspective that can be used to investigate types and methods, but that involves a significant shift of the user interface. Options are more limited from within the standard `JAVA` perspective. While *Eclipse* offers a way to browse the type hierarchy and the declared methods of a class, to see the documentation one must typically click a method to open its source code.

It is important to note though that the process of using the web-based documentation was far from efficient. Even when subjects knew exactly what they were looking for, they had to navigate several clicks to find the relevant package or class, and then to the relevant type and eventually the function. They frequently followed hyperlinks, which can lead to disorientation. It was extremely easy for them to get distracted by other types and methods, and to scroll around until they found the relevant materials.

The last task, to which we now turn, focused on the effectiveness by which developers find directives in subtypes when they are expecting them.

Before we proceed, however, we briefly note that all subjects performing this task in the experimental investigated either `containsAll` or `retainAll`, and were therefore exposed to the relevant directives. However, such exposure did not always translate to success in the task. Nevertheless, it had made all subjects aware of the fact that there is a violation in the subtype.

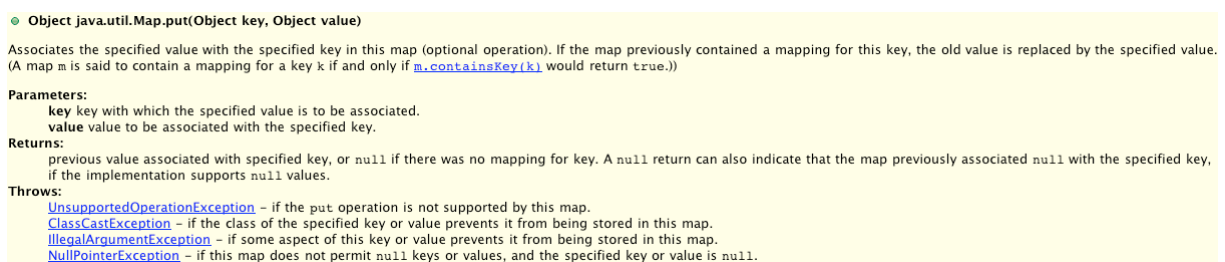
6.9 Second Collections Task

Our second collections task, the last task in this study, is less focused on awareness and more focused on exploration. As we described earlier, the process of identifying all the overriding versions of a method and specifically those that contain conflicting directives could be tedious. *eMoose* may have an impact on this process thanks to the augmented *JavaDoc* hover, which presents directives from a variety of overriding methods. To this end, subjects were presented with code similar to that of the first task, in which they were made aware of the existence of conflicting overriding versions. Their role was to identify the relevant directives that explain the errors. Controls had to cope with complex hierarchies of inheritance and interfaces. *eMoose* users, on the other hand, were presented with the correct directives but also with directives for unrelated types.

6.9.1 Directive for this task

Our directive for this task comes from the `put` method that is supported by collections that extend or implement the `Map` interface. The documentation for this method is presented in Fig. 6.44, as it was revealed to subjects in the study who used *eMoose*.

Recall that during the introduction of the study and the *eMoose* tutorial, subjects were reminded that the standard `TreeMap` implementation can fail if the passed key does not implement the `Comparable` interface. However, violations are possible in other less familiar maps, including those in *apache-collections*.



The image shows a screenshot of the Javadoc for the `put` method in the `java.util.Map` interface. The text is as follows:

```
Object java.util.Map.put(Object key, Object value)
Associates the specified value with the specified key in this map (optional operation). If the map previously contained a mapping for this key, the old value is replaced by the specified value. (A map m is said to contain a mapping for a key k if and only if m.containsKey(k) would return true.)
Parameters:
  key key with which the specified value is to be associated.
  value value to be associated with the specified key.
Returns:
  previous value associated with specified key, or null if there was no mapping for key. A null return can also indicate that the map previously associated null with the specified key, if the implementation supports null values.
Throws:
  UnsupportedOperationException - if the put operation is not supported by this map.
  ClassCastException - if the class of the specified key or value prevents it from being stored in this map.
  IllegalArgumentException - if some aspect of this key or value prevents it from being stored in this map.
  NullPointerException - if this map does not permit null keys or values, and the specified key or value is null.
```

Figure 6.44: Javadocs for the `Map.put` in the standard JAVA library

For our first overriding version, we picked the version of `put` from the bidirectional mapping class `BidiMap`. Its documentation (Fig. 6.45) states that adding a new mapping from a key to a certain object that is already part of a different mapping (from another key), would eliminate the original mapping. We consider this as an informative side effect directive.

We also picked a directive from `DoubleOrderedMap`, which manages a mapping using a red-black tree, and requires that all keys and values be unique. Its documentation (Fig. 6.46) states that an exception will be thrown if the key or value are already in the mapping. This is clearly a *parameter directive*.

● Object `org.apache.commons.collections.BidiMap.put(Object key, Object value)`

Puts the key-value pair into the map, replacing any previous pair.

When adding a key-value pair, the value may already exist in the map against a different key. That mapping is removed, to ensure that the value only occurs once in the inverse map.

```
BidiMap map1 = new DualHashBidiMap();
map.put("A", "B"); // contains A mapped to B, as per Map
map.put("A", "C"); // contains A mapped to C, as per Map

BidiMap map2 = new DualHashBidiMap();
map.put("A", "B"); // contains A mapped to B, as per Map
map.put("C", "B"); // contains C mapped to B, key A is removed
```

Specified by: [put\(...\)](#) in [Map](#)

Parameters:

key the key to store
value the value to store

Returns:

the previous value mapped to this key

Throws:

[UnsupportedOperationException](#) – if the `put` method is not supported
[ClassCastException](#) – (optional) if the map limits the type of the value and the specified value is inappropriate
[IllegalArgumentException](#) – (optional) if the map limits the values in some way and the value was invalid
[NullPointerException](#) – (optional) if the map limits the values to non-null and null was specified

Figure 6.45: Javadocs for the `BidiMap.put` in the standard JAVA library

● Object `org.apache.commons.collections.DoubleOrderedMap.put(Object key, Object value)` throws `ClassCastException`, `NullPointerException`, `IllegalArgumentException`

Associates the specified value with the specified key in this map.

Overrides: [put\(...\)](#) in [AbstractMap](#)

Parameters:

key key with which the specified value is to be associated.
value value to be associated with the specified key.

Returns:

null

Throws:

[ClassCastException](#) – if the class of the specified key or value prevents it from being stored in this map.
[NullPointerException](#) – if the specified key or value is null
[IllegalArgumentException](#) – if the key duplicates an existing key, or if the value duplicates an existing value

Figure 6.46: Javadocs for the `DoubleOrderedMap.put` in the standard JAVA library

6.9.2 Task description

Subjects are presented with the source code of Fig. 6.47. Once again, a size constant is defined and so is a range constant of a lower magnitude. This time, the array of collections is replaced by an array of maps. In addition, two separate arrays are declared - one for labels, and one for numbers.

The `setUp` method begins by allocating six maps into the array. It then allocates `SIZE` slots in the labels and numbers array. In a loop of size `SIZE`, a new number is randomly generated in `RANGE` during every iteration and is assigned to the corresponding slot in the numbers array. A label describing the iteration number is similarly assigned to the labels array. Thus, we have an array of distinct labels but of values with guaranteed duplicates.

The actual method `testMapPopulation` iterates over all the maps in the collection, operating on each separately. For each, it goes over the array and adds using the `put` method a mapping from the corresponding label to the corresponding value. Then, a second loop iterates again over the array and asserts that each label is mapped to the appropriate value.

Subjects are explicitly asked to identify two of the maps for whom the iteration would fail, and to offer evidence or demonstrate why.

This program will fail for the bidirectional map since when a mapping is added to a value that already has a key mapped to it, the first mapping is removed so that the original key is no longer mapped at that value. The program will also fail for the double ordered map as it does not support duplicates.

6.9.3 Results

As with the previous collections class, we are only interested in the performance of controls. Subjects in the experimental group who hover over the problematic call are presented with a hover that includes the directives for all overriding versions of the method, including the ones that constitute the answer. The challenge for *eMoose* users is therefore minimal, and all of them were unsurprisingly successful within a relatively short time.

The behavior of controls, however, can provide valuable insight on how developers access the documentation of subtypes. Since hovering over `put` merely provides the general documentation of the `Map` interface, they have no choice but to investigate the documentation of subtypes. They can do so via code, web, or IDE operations, but could become entangled in the complex hierarchy of JAVA collections.

Of the 13 subjects in the control condition, 8 were able to find both the problem in `DoubleOrderedMap` (“DOM”) and the one in `DualTreeBidiMap` (“BIDI”). Three others found only the problem in `DoubleOrderedMap`, and one did not find either. The timing data for all controls is presented in Table. 6.15. For each class, the table presents the time spent on specific methods (if any), and the time spent on everything else (labeled as “any”). For locations that were visited multiple time, the count appears after the total time in seconds. Note that the recording for subject S22 was corrupted.

Interestingly, while all subjects used the *JavaDoc* hover for investigating calls on the `Map`, all but one used the web-based *JavaDocs* to investigate the subtypes. The only exception is S7, who explored source code directly in the IDE, and stuck to the concrete classes instantiated in the main code, without exploring any of their supertypes.

We also see that all subjects explored both `DoubleOrderedMap`, which constituted one solution to the problem. All but two (S15, S26) examined `DualTreeBidiMap`, the former finding the solution from the supertype `AbstractTreeBidiMap` and the latter guessing the solution and the issue from the problem in `DoubleOrderedMap` without looking for confirmation.

```

import java.util.*;
import org.apache.commons.collections.*;
import org.apache.commons.collections.bidimap.DualTreeBidiMap;
import org.apache.commons.collections.map.ListOrderedMap;

import junit.framework.Assert;
import junit.framework.TestCase;

@SuppressWarnings("unchecked")
public class StandardMapsTest extends TestCase
{
    public final static int SIZE = 200;
    public final static int RANGE = 30;

    Map maps[] = null;
    String labelsArray[];
    Integer numbersArray[];

    @SuppressWarnings("deprecation")
    public void setUp()
    {
        this.maps = new Map[]{ new HashMap(), new TreeMap(), new DualTreeBidiMap(), new DoubleOrderedMap(),
            new MultiHashMap(), new ListOrderedMap()};

        labelsArray = new String[SIZE];
        numbersArray = new Integer[SIZE];
        for(int i=0; i<SIZE; ++i)
        {
            int value = (int)(Math.random()*RANGE);
            labelsArray[i] = "Label#" + i;
            numbersArray[i] = value;
        }
    }

    public void testMapPopulation()
    {
        for(Map map: maps)
        {
            for(int i=0; i<SIZE; ++i)
            {
                map.put(labelsArray[i], numbersArray[i]);

                // Make sure all labels and values are in there were added
                for(int i=0; i<SIZE; ++i)
                {
                    Assert.assertTrue(map.containsKey(labelsArray[i]));
                    Assert.assertTrue(map.containsValue(numbersArray[i]));
                }
            }
        }
    }
}

```

Figure 6.47: Source code for second collections task

Class	Method	S2	S3	S7	S9	S10	S12	S14	S15	S18	S20	S22	S24	S26
Map	Any put()						9.4						2	89 - 2x
	containsKey()			16.3 - 2x			4 - 2x	11 - 2x						
	containsValue()							13 - 2x						
TreeMap	Any put()	23.7 - 2x				18.9 - 3x		37 - 2x						41 - 2x
	containsValue()													21
HashMap	Any put()		2.1 - 1x	14 - 2x		13.3 - 2x		54 - 3x			9		63 - 2x	
	containsValue()										12			
AbstractHashMap	Any containsKey()						6				5			
ConcurrentHashMap	Any										4			
MultiMap	Any containsValue()				23.2 - 1x			17						
	remove()				32.2	19								
MultiHashMap	any put()	47.8 - 2x	40.5 - 1x	34.8 - 3x		31 - 2x	49.4 - 2x	58 - 3x	65.8	27	1		57	
	containsValue		5 - 1x					38	22					
OrderedMap	any			5.8 - 1x			14							
DoubleOrderedMap	any constructor	127.1 - 1x	62.1 - 1x	60 - 1x	112	109 - 3x	50.9 - 2x	104 - 2x	24	250	33		35	164
	put()			39.8 - 1x										
	containsKey()								23		29		7	
	containsValue()								15				23	
BidiMap	Any put()	26 - 1x		26.6 - 1x		14 - 1x		4						
		14.9 - 1x												
AbstractTreeBidiMap	Any containsKey()						32							
AbstractDualBidiMap	Any put()	26.2			19.3 - 2x	18 - 2x			145 - 4x		35		1	
	containsKey()						1		21		28			
OrderedBidiMap						4								
DualTreebidiMap	Any containsValue()	71 - 3x	17.7 - 2x	5.7 - 2x	63.1 - 2x	41 - 3x	8.5 - 3x	109 - 4x		117 - 3x	30		36.4 - 2x	
													15	
ListOrderedMap	Any put()					49	54 - 2x	53.6 - 3x		97			90 - 5x	102 - 2x
													79 - 3x	
AbstractMapDecorator	Any containsValue()							4.4					4	
String class	Any			2.4 - 1x										
MapUtils									17					
TIME FOUND DOM		370	271	772	219	370	612	End	561	X	340	?	690	570
TIME FOUND BIDI		153	129	880	389	X	X	End	690	X	142	?	X	613

Table 6.15: Timing data for the control condition of the second collections task

While subjects could reasonably suspect that the error occurred during the call to `put`, quite a few of them (S7, S10, S12, S14, S18, S26) never specifically investigated the documentation of any overriding versions. Two more subjects (S2, S3) only explored the `put` method of one of the methods. Instead, a lot of time was spent looking at classes in general and their documentation, and at other methods (such as `containsKey`). Many found the solution from the very verbose class-level documentation, which described how the underlying data structure worked and thus its limitations.

It is also interesting that most subjects visited supertypes and superinterfaces of the concrete types instantiated in the test program. For example, the `BidiMap` interface was explored by 4 subjects, and 4 more explored its implementing class `AbstractDualBidiMap`. One of these explored `AbstractTreeBidiMap`, and another explored `OrderedBidiMap`. My impression from watching the recordings is that subjects sometimes became disoriented, especially when a link led them to an unexpected location in an unfamiliar hierarchy.

Subjects also spent significant time on types of maps that did not convey the answer to the problem. In particular, Almost everyone explored `MultiHashMap` for at least 30 seconds, and many explored its supertypes including the standard `HashMap` that is familiar to most programmers for longer periods. The `ListOrderedMap` class was explored for around a minute by many of the subjects as well.

6.9.4 Discussion

This task was fundamentally different from the previous one in that this time subjects were aware that a problem resulted from an issue in an overriding version of a method, and only had to find the specific version and the problem. The results from the control group suggest that while most subjects were eventually able to do so within the allotted time, the process was tedious and risky.

The most perplexing facet of the subjects' behavior was the avoidance of the `put` method. A logical approach to solving this problem would have been to systematically examine each of the instantiated map types, investigate its `put` method, and see if it yields the answer; If necessary, supertypes could be investigated as well. In fact, this is the strategy taken by most subjects in the experimental condition: they hovered over `put`, and systematically examined the directive for every overriding version. Instead, subjects divided their time on these classes between scrolling through method lists and occasionally investigating specific ones, and reading the class header documentation. For many of these classes, this header text spanned more than a page, raising the risk that the important details would not be noticed.

A more general finding, however, is that almost all subjects chose to leave the IDE and use the web-based documentation, something they had not done much in the previous tasks. This suggests that there are shortcomings to current IDE support for the exploration of overriding versions.

The closest mechanism in *Eclipse* which could have been of utility to our subjects is the context-sensitive type hierarchy. When selecting a method, a key combination or menu selection brings a lightweight window with a tree-view of subtypes that override that method. Subjects could have done this for `put`, and then searched for the specific subtypes to open their source code, which would include the *JavaDocs*. It is not clear why none of the subjects attempted to use this mechanism, even though some were quite versed in using *Eclipse*. One possibility is that they wanted to avoid the source code. After all, all but one avoided directly opening the sources of the classes instantiated in the main code. I believe that it may be useful for the type window to provide means to examine the *JavaDocs* of the target without require an opening of the source code.

My impression from watching the subjects' work is that they also had trouble grasping the complex hierarchy of the map types and sometimes became disoriented. They would often click on a link and end up in a different class, without an understanding of its relation and place in the hierarchy. While JAVA supports only single inheritance, the use of interfaces results in complex multiple-supertype hierarchies. The HTML based presentation in a *JavaDoc* file and the tree-based viewers in the IDE are both designed around single inheritance and thus a single supertype. I argue that this limits their utility in understanding classes within complex hierarchies.

Furthermore, it also appears that the structure of web-based *JavaDocs*, which separates the set of inherited methods from the set of newly declared or overridden methods leads to confusion. Developers who wish to understand a concrete class as a single cohesive API element need to deal with a seemingly artificial division of the methods based on an implementation detail - whether they are inherited and overridden. This separation makes it difficult to find methods or understand what is actually supported. In addition, the inherited methods are presented in a visually condensed form that makes a search even more difficult.

6.10 Additional general findings

This section presents and discusses general results and behaviors that we observed in our study across multiple tasks. We note that further studies are necessary to determine whether these occur in everyday development.

6.10.1 Repeated readings

A common phenomenon that we observed in the behavior of most subjects across multiple tasks was the repeated hovering over elements whose *JavaDocs* have been presented and presumably read one or more times.

While it is conceivable that this represents a memory failure or a limit on the developer's memory, we suspect that this is not the case. First, this behavior often involved very short *JavaDocs* whose contents can be easily memorized, or that do not provide much information beyond that conveyed in the signature. Second, these repeated readings often took place not long after a previous read of that method, making memory loss less likely. Third, this phenomenon also applied to code elements that were not function calls, such as fields or variables (an issue that we discuss later). Therefore, we suspect that this behavior is often not related to a limit on knowledge absorption and memorization, but rather to another issue.

A possible interpretation is that the hover over a call or an element so that its *JavaDocs* are displayed serves as visual means to support orientation. When the developer wants to make the mental shift in focus from one element to another, he may hover over it until its documentation is presented as a reinforcement of the explicit delay involved in the shifting. Once the documentation is open, it may be kept open while the developer is reflecting but without necessarily being read.

We also suspect that hover actions, and in particular repeated ones, often serve a similar role to the use of the mouse and selections to provide reinforcement and orientation, as we later discuss. In other words, it is possible that the visual presence of the mouse pointer over the call serves to orient the developer to the current point of focus. The appearance of the hover then frees the mouse to travel to another location, with the hover staying open for a few seconds as reinforcement.

In the case of longer *JavaDocs*, developers did seem to often read materials they have previously read, as evidenced by mouse motions, selections and vocalization within the text. We suspect that such repeat readings may not simply be a result of a subject forgetting previously-read text, but rather represent a result of a shift in focus. Our impression was that subjects often followed a theory-driven exploration [75]: they read a particular *JavaDoc* or a sentence within it with a particular theory in mind. Even though they may have absorbed the entire text, they may have been thinking about it in terms of their theory at the time, leading them to pay less attention and not notice other clauses. This behavior may have caused them to miss certain directives in early readings. However, as their theories and goals changed, they might have conducted a repeat visit in which everything was reinterpreted. This reinterpretation, or just random chance or a lack of specific goal would have lead to an eventual discovery of the directive.

6.10.2 Hovering over non-call elements

A related behavior which we found surprising and which consumed significant amount of time for most subjects in most tasks was the use of the hover mechanism for elements that do not have associated *JavaDocs*. Specifically, subjects often examined variables and fields, and often did so repeatedly. Outside the debug mode, which was only permitted in the SWING task, such actions carry very little reward. They only present the type of the object, which can often be determined from the signature, rather than any actual value or useful information.

One possibility is that this behavior serves purposes of orientation as described above. After all, if the subject is aware from previous visits that no information at all will be displayed, then repeated visits would be indicative of a completely different goal that does not involve information consumption, such as orientation.

Nevertheless, since such unproductive behavior is so frequent, and seemed to increase when subjects were becoming "desperate", we suspect that it may also be indicative of a serious need for certain information on these data members and variables. In situations where a failing execution was demonstrated, subjects may have sought the values of these variables at run time, to understand the data flow in the program. This could have been made more severe by the lack of access to the debugger, although such behavior was also present in the SWING task in which debugger use was prevalent.

A particularly interesting possibility is that rather than seeking the value of the variable or the field, what the subjects were actually trying to do is to find information about the role and purpose of the variable. Such details are rarely documented in detail, and especially not with the commenting convention that is reflected as *JavaDocs*. This may indicate a greater need for developers to provide more information about the variables and data flow as comments in their code.

6.10.3 Use of the autocomplete mechanism

Another common behavior that we have encountered was the use of the autocomplete mechanism to identify all the methods supported by a given object. To accomplish this, developers would create a separate line with the variable, add a dot, and obtain the list, or they would eliminate the dot in an existing call and add it again, so that the text is presented.

Clearly, subjects often faced the need to explore all the methods supported by a class, and in particular alternatives to the existing calls. Unfortunately, the autocomplete list is a poor mechanism as it requires a lot of scrolling, is always sorted alphabetically, and requires actual keystrokes before the documentation of the alternative targets are presented.

We note that the use of the web-based documentation offers more screen real estate and a summary documentation for each method, but it has its faults. First, it requires a costly transition to the web browser, with costs of reorientation on return to the IDE. Many subjects appeared reluctant to make this transition frequently. Second, the web-based documentation is organized by class, making it difficult to see the signatures of inherited and implemented methods. The autocomplete list, on the other hand, displays all methods supported by the class regardless of their origins.

While the exploration of alternatives was unproductive in most tasks and often lead to distraction, it was of course the core of the SWING task. It appears that some subjects thought that they needed to add or change calls and so began exploring alternatives. They particularly focused on overloaded versions of the original method or on ones with similar names, such as `receiveNoWait` instead of `receive`.

I believe that the documentation of a method that has strongly-related overloads or alternatives should list these alternatives in its own documentation, so that developers can recognize and understand the differences and make an informed decision. More generally, I suspect that there may be a benefit to organizing the methods of a class in a semantically meaningful way such as the involved parts of the state [27]

In general tasks and when learning APIs, however, the ability to understand the “recipes” to accomplish specific goals is important, and various approaches exist to support this [31, 35]. Recent research [81] suggests that developers may need “placeholders” to aim them at specific ways to accomplish certain goals.

6.10.4 Use of mouse for highlighting

An almost universal behavior in our study was the use of the mouse pointer, text selection, or subvocalization to indicate the current location in code or documentation. These activities seemed to help subjects reinforce their short term memory. When they had to temporarily switch to another artifact, it took them some time to become reoriented.

Present IDE support [47] uses activity history to identify recently visited files and methods. However, if the above behaviors are universal, it may also be beneficial to track activity and present cues within the editor and *JavaDoc* hover. For example, selected or explored code or *JavaDoc* text would be shaded differently from the rest of the text, and would gradually return to its original shade as time passed.

Developers may also benefit from the ability to create short term annotations to indicate, for example, that a certain artifact has been explored or that a clause has been eliminated. Some sort of “instant replay” mechanism may also help facilitate reorientation.

6.10.5 Ordering and learning effects

One of the goals in designing the study was to minimize the potential for a “learning effect”, where realizing the importance of directives in an early task would lead to a more careful scrutiny in the subsequent tasks. To achieve this, we had all subjects carry out the tasks in the same order. We also placed the three debugging tasks at the beginning of the study, to minimize the effect that could build up, as the later tasks exposed subjects to completely different problems. In addition, the second task require substantially different methods than the first - a careful reading of a long text over making the right reading choices.

Nevertheless, one place where learning effects could have potentially taken place was between the first and third tasks. Both tasks required identifying a directive in the second sentence of a short *JavaDoc*, and were also always carried out in the same mode - with or without *eMoose*. The low success rates among controls in the third task suggests that being aware of the importance of careful reading is not necessarily sufficient for success. A large number of potential exploration targets still appears to wear down developers and lead them to miss directives.

6.11 Exit Survey

After completing all tasks, subjects were asked to fill an exit questionnaire covering topics such: their impressions of *eMoose*, opinions about possible extensions, and their programming practices. The anonymous questionnaire involved ranking 27 statements on an integer Likert scale between -3 (strongly disagree) and $+3$ (strongly agree), though no textual labels were provided for the interim values.

The goal of this survey was to obtain some measurement of how the subjects perceived the problem of directive awareness, whether they considered the approach of *eMoose* to be effective, and whether it could fit into their everyday practices.

The survey has several inherent limitations: First, subjects only had limited opportunities to work with *eMoose* and form an informed opinion on its strengths and weaknesses. Second, their impression of *eMoose* may be affected by their performance in the study and by the fact that the tasks were specifically designed to evaluate the strengths and weaknesses of the tool. Third, many subjects were students and may attempt to leave a positive impression. Nevertheless, differences between answers to specific questions may be telling.

The presentation that follows maintains the division of the statements into four groups, and keeps the original phrasing. For each question, we present the distribution of scores in tabular and graph form, and briefly discuss these results.

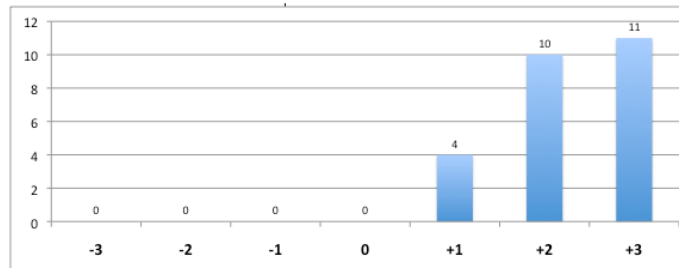
6.11.1 Statements on “Marked calls”

The first set of statements is concerned with the primary feature of *eMoose*- call decoration. Note that throughout the study, we used the term “marking calls” rather than “decorating calls”, so the same terminology appears in these statements.

Statement 1A: eMoose sometimes offered significant help in identifying interesting calls

The goal of this question and the one which follows was to understand whether developers perceived benefits from the contextual features of *eMoose*. The use of the term “interesting” was deliberate, as only one call yielded the solution, but subjects may have perceived a value from investigating the others. The term *sometimes* was used to check if there were any cases where it was helpful: each subject only had a few chances to use the tool. If subjects asked about the semantics of this term, they were told to interpret it as “there were cases”.

Choice	#Subjects
-3	0
-2	0
-1	0
0	0
+1	4
+2	10
+3	11
Count	25
Average	2.28
Median	2

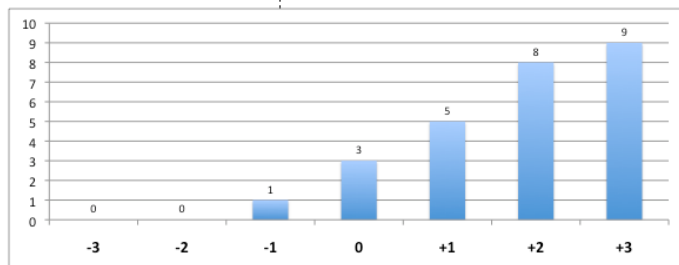


All subjects provided a positive answer which leaned towards the strong agreement, suggesting that value was perceived in at least some cases.

Statement 1B: eMoose usually offered significant help in identifying interesting calls

The main goal of this statement was to qualify answers to the previous statement, as the differences may be telling. Indeed, subjects tended to agree less with this statement, including several who were neutral or negated it. Thus, the perceived benefit is only in some cases, which is realistic and is in line with the goals of our approach.

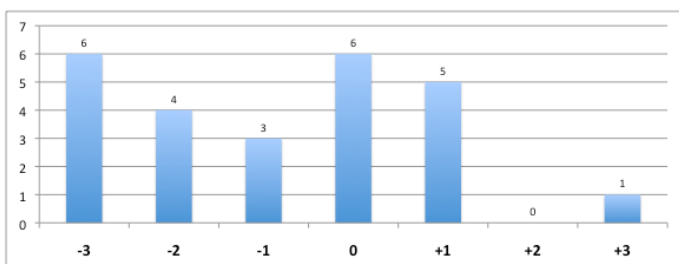
Choice	#Subjects
-3	0
-2	0
-1	1
0	3
+1	5
+2	8
+3	9
Count	26
Average	1.8
Median	2



Statement 1C: eMoose sometimes distracted me by making me look at the wrong calls

The benefits of *eMoose* come at a risk of distraction, and our tasks offered several opportunities for decorations to attract readers to irrelevant calls. Nevertheless, only a small portion of subjects felt that they were distracted.

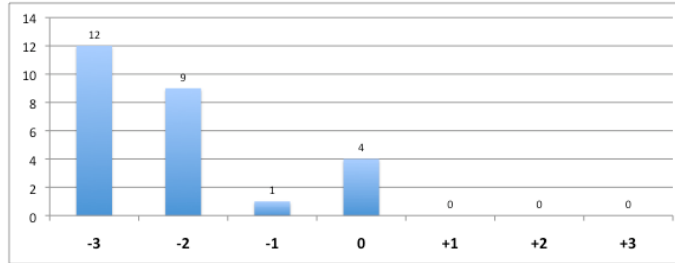
Choice	#Subjects
-3	6
-2	4
-1	3
0	6
+1	5
+2	0
+3	1
Count	25
Average	-0.8
Median	-1



Statement 1D: eMoose usually distracted me by making me look at the wrong calls

This statement is similar to the one that precedes it, but inquires whether distraction was frequent. None of the subjects appeared to think so.

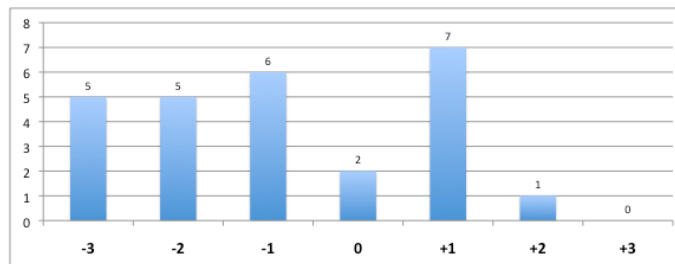
Choice	#Subjects
-3	12
-2	9
-1	1
0	4
+1	0
+2	0
+3	0
Count	26
Average	-2.1
Median	-2



Statement 1E: eMoose marked too many calls

Even if subjects did not actually explore calls which ended up not being relevant to their needs, the sheer number of decorated methods could be distracting. Subjects were split as to whether this was the case, although the majority did not agree.

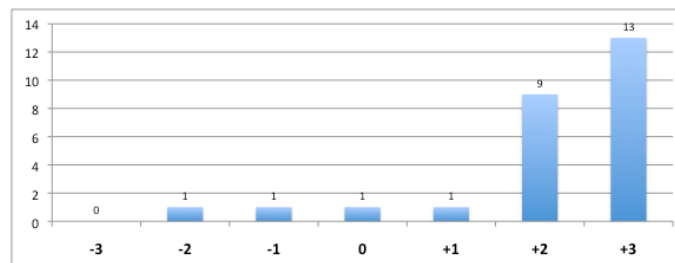
Choice	#Subjects
-3	5
-2	5
-1	6
0	2
+1	7
+2	1
+3	0
Count	26
Average	-0.8
Median	-1



Statement 1F: I tended to look at marked calls first

One of the mechanisms by which *eMoose* can help but also distract callers is by making them pay more attention to the decorated calls. This may interfere with whatever process a developer uses to understand a given code fragment. The strong agreement of most subjects indicates a potential risk of using *eMoose* which should be further investigated.

Choice	#Subjects
-3	0
-2	1
-1	1
0	1
+1	1
+2	9
+3	13
Count	26
Average	2.1
Median	2.5

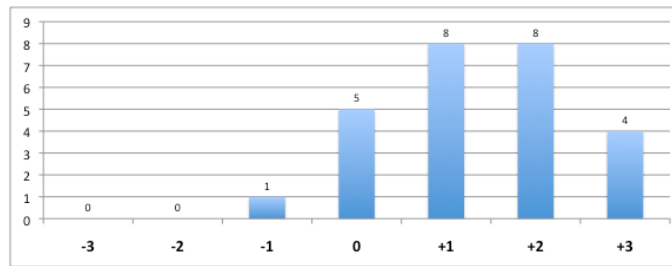


Statement 1G: I ignored some calls because they were not marked

Another risk of using *eMoose* is that a method which is not decorated would be perceived as not important, even though the presence or lack of decoration is not directly related to the relevancy of the method to the developer’s current needs. If developers ignore calls, they may miss important details that are not

directives. In addition, with the present implementation, they may mistake methods in APIs that have not been annotated with directives as less important.

Choice	#Subjects
-3	0
-2	0
-1	1
0	5
+1	8
+2	8
+3	4
Count	26
Average	1.3
Median	1



Again, the results indicate that this is indeed a risk, further motivating the need to better educate users about the meaning of a presence or lack of decoration.

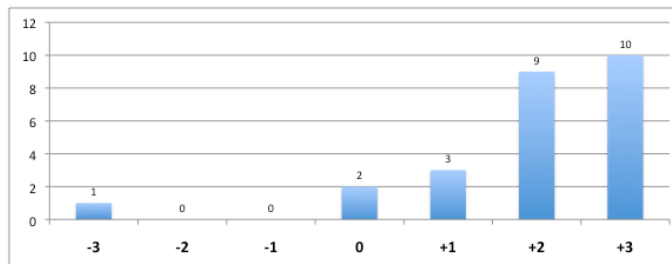
6.11.2 The *JavaDoc* Hover

We now turn to statements focused on the second feature of *eMoose*, the augmented *JavaDoc* hover. These questions are preceded by the following note: “These questions are concerned with the window that appear when you hover over a call. With *eMoose*, the top contained the *JavaDoc* while the bottom contained directives.”

Statement 2A: *eMoose* sometimes offered significant help with the lower pane

Again, we start by asking whether this feature was useful in some cases. The results are positive in almost all cases, although not as strong as for method decoration.

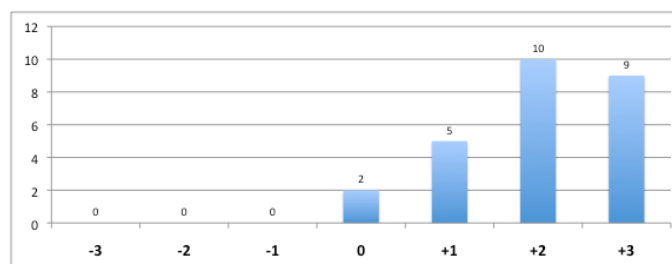
Choice	#Subjects
-3	1
-2	0
-1	0
0	2
+1	3
+2	9
+3	10
Count	25
Average	1.9
Median	2



Statement 2B: *eMoose* usually offered significant help with the lower pane

We also asked the same question with “usually”. Surprisingly, the results are nearly identical.

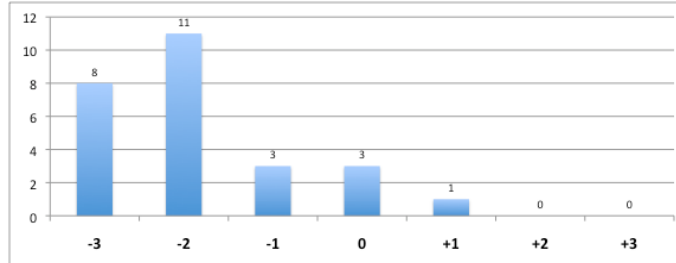
Choice	#Subjects
-3	0
-2	0
-1	0
0	2
+1	5
+2	10
+3	9
Count	26
Average	2.0
Median	2



Statement 2C: The redundancy of information in the lower pane was distracting

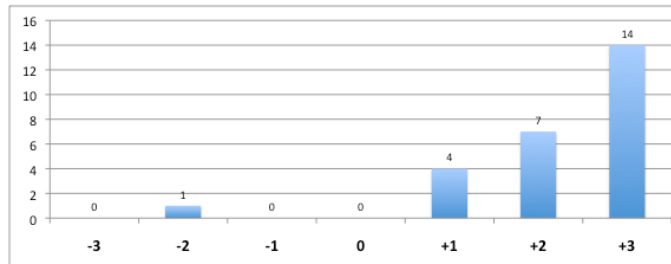
eMoose creates a redundancy between the text in the upper pane and the directives in the lower pane. However, most subjects were not distracted by this redundancy.

Choice	#Subjects
-3	8
-2	11
-1	3
0	3
+1	1
+2	0
+3	0
Count	26
Average	-1.8
Median	-2



Statement 2D: I tended to read the information in the lower pane before reading upper pane

Choice	#Subjects
-3	0
-2	1
-1	0
0	0
+1	4
+2	7
+3	14
Count	26
Average	2.2
Median	3

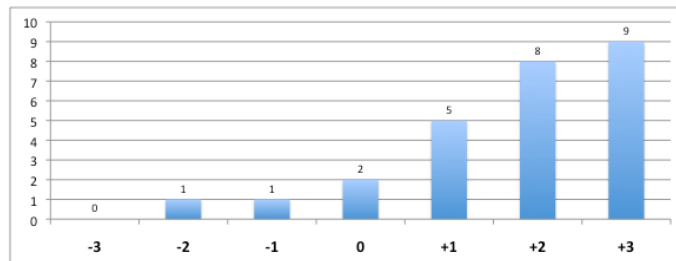


As intended, most subjects read the directives first.

Statement 2E: I decided whether to read the upper pane based on the lower pane

One of the goals of listing directives explicitly is to offer a lower-cost way to explore the method and decide whether to invest more attention, without having to read all its documentation. This, however, raises the risk that import text would be ignored if it was not tagged as a directive.

Choice	#Subjects
-3	0
-2	1
-1	1
0	2
+1	5
+2	8
+3	9
Count	26
Average	1.7
Median	2

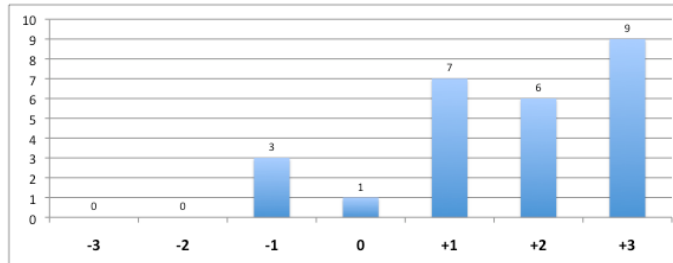


Indeed, subjects reported that they based their decision on the lower pane. Examining mouse motions and scrolling actions in the recordings seems to support these reports, as attention was often initially focused on the lower pane. It would be interesting to study whether directives that are not tagged would indeed be missed by designing such a task in a future study.

Statement 2F: I avoided reading the upper pane when there was information in the lower pane

This statement complements the previous two statements and uses slightly stronger terms. The results were consistent with these statements.

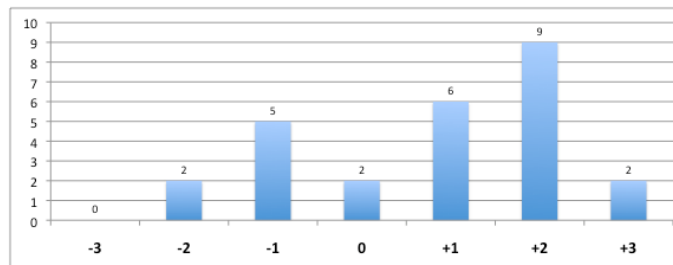
Choice	#Subjects
-3	0
-2	0
-1	3
0	1
+1	7
+2	6
+3	9
Count	26
Average	1.7
Median	2



Statement 2G: It was straightforward to correlate observations to documentation sentences

If developers find the directives relevant and read the entire method text to learn more, they do not receive any help from *eMoose* in connecting the directives to the text. The disagreement among some subjects with the statement that correlation was straightforward suggests that mapping assistance would be a useful feature.

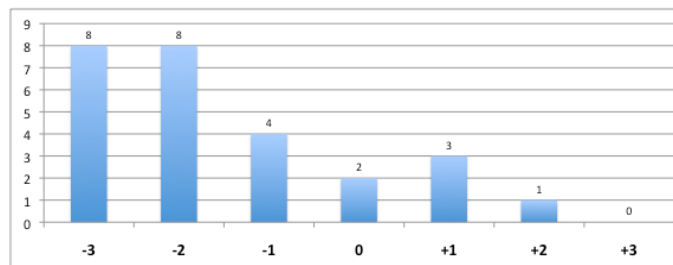
Choice	#Subjects
-3	0
-2	2
-1	5
0	2
+1	6
+2	9
+3	2
Count	26
Average	0.8
Median	1



Statement 2H: It would be better to mark sentences in the JavaDoc than to have the separate pane

A possible alternative design to the redundant listing of directives in the lower pane is to highlight text within the *JavaDoc*. The downside is that this would require scrolling, and would not allow text to be summarized. However, most subjects disliked this idea.

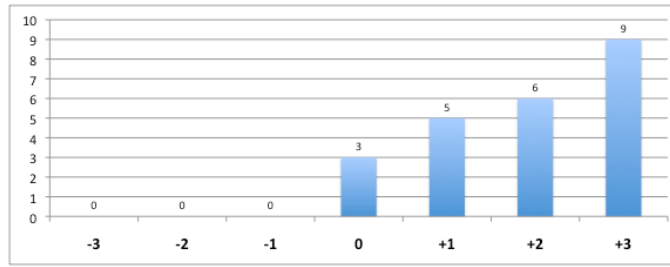
Choice	#Subjects
-3	8
-2	8
-1	4
0	2
+1	3
+2	1
+3	0
Count	26
Average	-1.5
Median	-2



Statement 2I: I occasionally missed something important when reading the documentation

The main reason for explicitly listing directives in their own pane is the fear that they would be missed in the documentation narrative, as was the case in our second JMS task. Indeed, all subjects indicated that they occasionally missed something in the text.

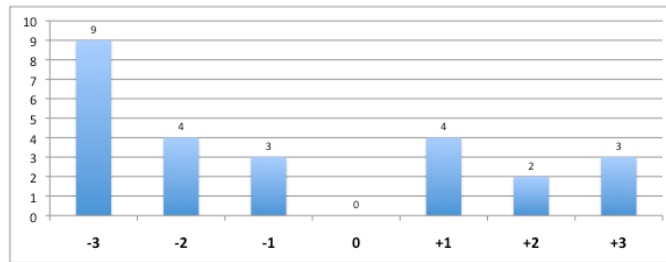
Choice	#Subjects
-3	0
-2	0
-1	0
0	3
+1	5
+2	6
+3	9
Count	23
Average	1.9
Median	2



Statement 2J: I would have liked to see all directives overlaid on the source code instead of having to explicitly hover over calls surrounded by boxes.

In the current implementation of *eMoose*, an extra step is required if a developer wishes to investigate a decorated method and understand what directives are associated. A hidden feature of *eMoose* allows all the directives to “float” in semitransparent bubbles near the calls. However, this raises the risk of clutter, and was rejected by most subjects. Nevertheless, agreement among some subjects suggests that there may be a benefit to this if it can be made less intrusive.

Choice	#Subjects
-3	9
-2	4
-1	3
0	0
+1	4
+2	2
+3	3
Count	25
Average	-0.8
Median	-2



6.11.3 Polymorphism

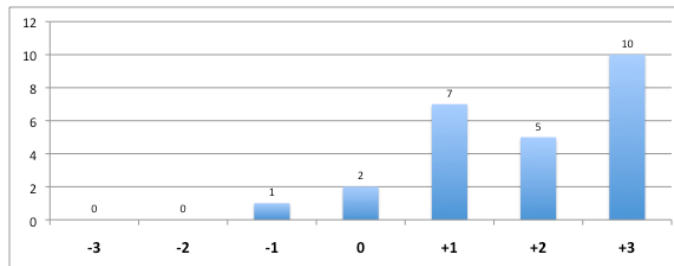
The third set of statements deals with polymorphism and the support of *eMoose*. Before the statements in this set, the clarification text reads:

“Dynamic type is the type of an object during runtime which can be a subtype of the declared (static) type. For example: `Set mySet = new TreeSet()` and `Set mySet = new HashSet();`”

Statement 3A: eMoose helped in finding information in dynamic types

eMoose supports polymorphic code in two ways: first, it highlights calls to targets where a directive is present in a possible overriding version, making readers aware of the polymorphism. Second, it presents the overriding versions in the *JavaDoc* hover, making it easier to identify the associated directives once the polymorphism is already known. While we did not distinguish between the two mechanisms in this statement, most subjects agreed that the support was useful.

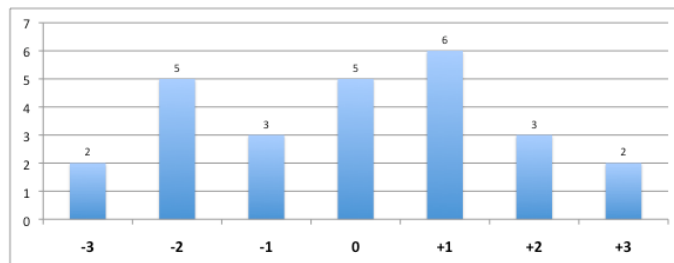
Choice	#Subjects
-3	0
-2	0
-1	1
0	2
+1	7
+2	5
+3	10
Count	25
Average	1.8
Median	2



Statement 3B: eMoose distracted me by showing too many possible dynamic types

The main downside of *eMoose*'s polymorphic support is its conservative approach to subtype analysis - Namely, every present subtype in the project is considered to be a potential dynamic type, even if static analysis would reveal that it cannot. Subjects were equally distributed on whether this was a problem, though some mentioned vocally during the study that they would have liked to see this change.

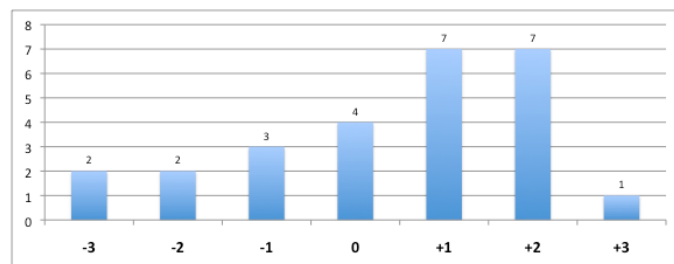
Choice	#Subjects
-3	2
-2	5
-1	3
0	5
+1	6
+2	3
+3	2
Count	26
Average	0.0
Median	0



Statement 3C: I have encountered similar situations with differences in dynamic types before

This statement tried to explore how applicable such support can be to the subject's everyday development work. About half the subjects indicated that they have indeed encountered such situations in the past. However, it is important to remember that subjects varied significantly in their development experience.

Choice	#Subjects
-3	2
-2	2
-1	3
0	4
+1	7
+2	7
+3	1
Count	26
Average	0.4
Median	1



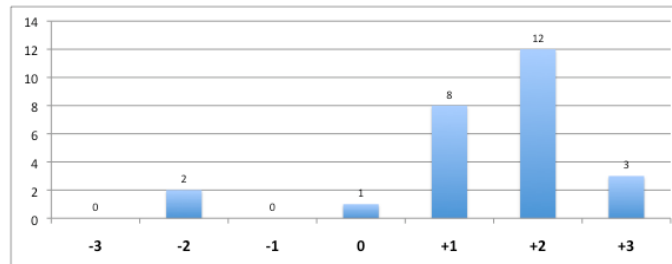
6.11.4 General questions

Finally, the last set presents a variety of statements about the study and the subjects' everyday experience.

Statement 4A: The tasks were challenging

Subjects were asked for a subjective estimate of the challenge in the study tasks. Most felt that the tasks were challenging, although not extremely so.

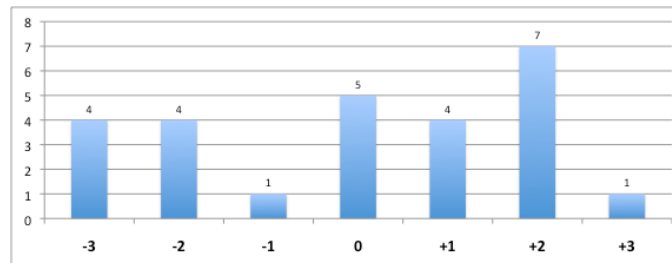
Choice	#Subjects
-3	0
-2	2
-1	0
0	1
+1	8
+2	12
+3	3
Count	26
Average	1.4
Median	2



Statement 4B: The activities were similar to what I often do in my everyday use.

Again, we attempted to glean information about the potential applicability of *eMoose* to everyday use. Subjects were split as to whether these activities were similar to programming behavior.

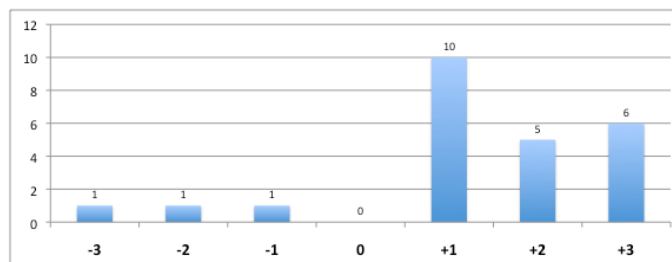
Choice	#Subjects
-3	4
-2	4
-1	1
0	5
+1	4
+2	7
+3	1
Count	26
Average	0.0
Median	0



Statement 4C: To accomplish tasks I had to read documentation significantly more carefully than I am used to in everyday use

Because of the artificiality of the lab setting, the allotted time for small code sections, and the knowledge that this is a limited study, it is possible that subjects paid closer attention to the documentation. The vast majority of subjects agreed, suggesting that the potential for problems in real-world situations may be greater.

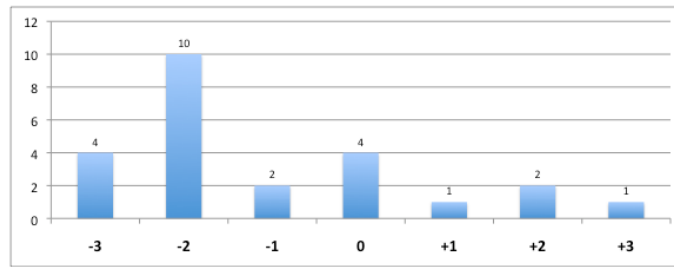
Choice	#Subjects
-3	1
-2	1
-1	1
0	0
+1	10
+2	5
+3	6
Count	24
Average	1.3
Median	1



Statement 4D: I prefer to read JavaDocs in HTML form

Any support within *Eclipse* would be less useful if subjects preferred the traditional way of reading *JavaDocs* over the convenience of using IDE features. However, only a few subjects preferred the HTML.

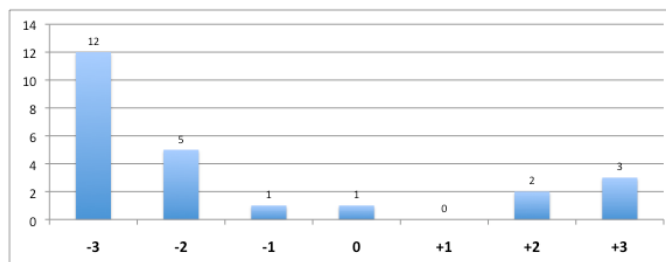
Choice	#Subjects
-3	4
-2	10
-1	2
0	4
+1	1
+2	2
+3	1
Count	24
Average	-1.1
Median	-2



Statement 4E: I prefer command line tools and editors like emacs to IDEs like Eclipse

There is a well known divide between developers who prefer integrated IDEs and may be more open to additional tools and additional presented information, and “purists” who prefer simple text-based tools with minimal additional information. *eMoose* is designed with the first group in mind, so it made sense to check the preferences of our subjects. The vast majority of subjects preferred IDEs. However, this likely results from characteristics of the subject population: JAVA programmers in their early 20s who are comfortable with *Windows*. One subject noted during the study that as a C++ programmer on *Linux*, he dislikes IDEs.

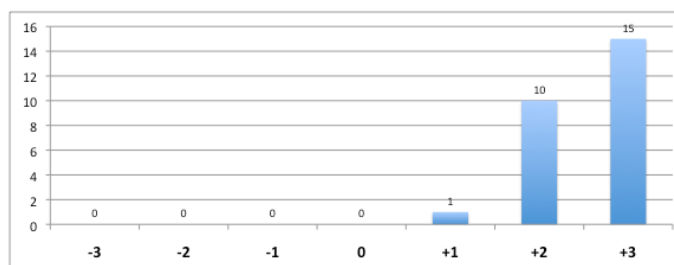
Choice	#Subjects
-3	12
-2	5
-1	1
0	1
+1	0
+2	2
+3	3
Count	24
Average	-1.4
Median	-2.5



Statement 4F: eMoose may be useful in my everyday use

Finally, we asked whether the subjects could potentially see a use for a tool like *eMoose* in their everyday development. While answers were uniformly positive, the reliability of this question is inherently limited.

Choice	#Subjects
-3	0
-2	0
-1	0
0	0
+1	1
+2	10
+3	15
Count	26
Average	2.5
Median	3



6.12 Limitations

This study demonstrated the reality of the directive awareness problem in code, and of the significant impact that *eMoose* may have on this problem. The detailed analysis of the transcripts also allowed us to develop theories to explain the mechanisms behind these problems and the impact of *eMoose*. Nevertheless, one must be careful when attempting to generalize these results to real-world development, as the study has several limitations, which we now discuss.

6.12.1 Artificial nature of tasks

The biggest limitation of this study is that most of its tasks³ were specifically designed to test for certain pitfalls that are related to directive awareness, and to evaluate the effectiveness of *eMoose* when its potential returns are highest. This was done under a premise that if controls had no problems or *eMoose* had no impact in such extreme and custom-designed cases, it is unlikely that these phenomena would be manifested in real-world scenarios. However, getting the expected results does not necessarily imply that the effects are as significant in real world conditions. Subsequent research and deployment by users is needed to validate this question.

In thinking about the tasks of the study and their potential for generalization, it may be beneficial to distinguish between the control condition and the experimental condition. Even if the results for the experimental condition could be dismissed solely as an artifact of the specific task design which is favorable to *eMoose*, the difficulties of subjects in the control condition are still concerning. In my industrial development experience, scenarios like the ones in the study tasks are quite common and have resulted in actual bugs. There are often important details in an unexpected or incorrect location, deep inside a detailed specification, or in a conflicting overriding version. As a result, I believe that the same awareness problems likely generalize to larger scenarios although they are perhaps less frequent. Nevertheless, even if a developer encounters a problem of this sort once every few hours, the expense in subsequent debugging due to errors that might have been prevented is still significant.

The impact of *eMoose* in real-world scenarios is likely to be smaller in magnitude than in the study tasks for several reasons: First, the incidence of problems for controls may be less frequent. Second, the developers may be more familiar with the API, as in this study I have specifically chosen unfamiliar ones, so there may be less of a need for *eMoose*. Third, the directive may not be as clear and straightforward to tie to problems as the ones chosen for the task. Fourth, there may be more distractions.

Nevertheless, I believe that *eMoose* could still be quite effective in real settings, especially when developers use my corpuses of directives for popular APIs. The hundreds of directives I have identified in commonly used packages and classes (and thousands in the entire library) makes it likely that at least some directives would not be immediately familiar to the users. In addition, we have seen from the third task that distraction by directives may not be a concern. Finally, the effectiveness of *eMoose* in the polymorphism tasks suggests that it may be of particularly high value in the cases of unexpected conformance violations which are infrequent but still common.

6.12.2 Debugging nature of tasks

Another major limitation of this study is that it did not focus on the prevention of errors, which is the main goal of *eMoose*, but rather on debugging. The use of debugging tasks ensured that subjects needed to identify the directive and that they were exposed to the same codebase. My attempt to evaluate the prevention of errors, in the second SWING task, proved impractical and the task had to be cancelled.

The errors in the study tasks were designed to be plausible in nature. Since controls did not become aware of the directives in a focused debugging effort, it is unlikely that they would have become aware of them had they copied code, reversed statement order, or picked the wrong call. An early awareness of the directives may have led to the prevention of errors.

A related problem is that task scopes and fragment sizes were smaller to ensure that code and documentation examination was the primary activity. In everyday use, where there are other competing activities such as design and testing, the tool will only be applicable during a smaller portion of the time.

³The first SWING task also attempted to distract *eMoose* users

At those times, however, it may potentially be more effective than in the study, because developers would likely be reading documentation less carefully.

6.12.3 Subject background

A major limitation of this study was that most subjects were relative novices rather than experienced developers. While further study is necessary with experienced developers, I suspect that these subjects actually represent a major portion of industrial developers. They were graduate students in an IT oriented program in a major school, all had prior internship experience, and most go to the industry upon graduation.

I believe that by choosing such APIs or parts of them that none of the subjects were familiar with, I slightly “evened” the playing field by reducing the risk that some subject’s prior familiarity with a specific call would obfuscate the results in either condition.

Though more experienced developers are more familiar with more APIs, they are unlikely to be familiar with all of its intricacies. For example, experienced subjects were familiar with SWING and the collections framework, but none were familiar with `JLayeredPane` or `Bag` and their directives.

Because subjects were relatively similar in their academic background and industrial experience, it is difficult to objectively quantify their true experience level in order to correlate it to performance. Informally, I subjectively assessed experience based on the background sheet and a conversation with the subject prior to beginning the session. There were subjects who primarily developed software in an academic capacity, and some that developed as a hobby since their teens. My impression is that the least experienced subjects struggled more, made more dubious reading choices, formed incorrect theories, etc. The most experienced subjects, on the other hand, sometimes were extremely focused and able to quickly identify the relevant call, focusing on it and immediately spotting the directive. The majority of developers likely fall between these two extremes and are therefore likely to have awareness difficulties and to benefit from *eMoose*. Note that even the experienced and most educated subjects occasionally failed tasks without *eMoose*.

6.12.4 Set of directives

A possible threat to the validity of this study is that calls were decorated or left undecorated based on a corpus of directives that I have created prior to creating the tasks for this study. A major concern is whether other developers would come up with a similar set. In Chap. 7, I present results from a study showing consistency between my decisions for these methods and those of several other independent developers.

Chapter 7

Consistency in identifying directives

7.1 Introduction

In Chap. 4 I presented my definition of directives, and suggested a taxonomy of directive types. Then, in Chap. 6, I presented empirical evidence that directives can convey information that may affect developer performance on certain maintenance tasks. I created the set of directives used in that study by applying my judgement to the above definition. An important question, however, is whether my own judgement could have biased this set, or whether other developers would have come up with a similar set.

A more general question, with important consequences for this dissertation, is whether developers are generally in agreement about whether a particular documentation clause constitutes a directive. This question is important because the premise behind identifying and increasing awareness of directives is that they are likely to provide value to the majority of developers. If a clause designated as a directive by the set creator is only useful to a minority of developers, then it could prove distracting to the others.

Furthermore, since the creation of directive sets for a particular API is a lengthy process, its prospects of being carried out depend on the ability to “split the work” and to collaboratively refine this set. If developers are not consistent in tagging the directives, then one section of the annotated library would omit directives that other developers may consider useful, while another section would highlight information that others do not benefit from. If the sets can be edited by everyone, then this could result in a permanent state of flux.

The question of consistency can be considered at five levels of granularity, described below:

1. Method-level - Are developers in agreement on *which methods* convey *at least one directive* and which do not? I argue that this is the most important question for *eMoose* because it directly affects its main feature - the decoration of methods. Inconsistency would severely reduce the utility of the decorations for two reasons: First, if many decorations lead users to read the *JavaDocs* only to find irrelevant information, they might start ignoring them. Second, if many important directives are omitted, users may have to investigate undecorated methods to be on the safe side. Agreement between developers, on the other hand, would further motivate the support for directive ratings.

2. Clause level - Are developers in agreement on *which clauses* within the documentation of a method constitute directives? This question is also important because the *JavaDoc* hover explicitly lists the directives in the lower pane, so a lack of agreement would reduce the utility of this pane. Users would have to read the full *JavaDoc* to see if it contains information that is useful and not tagged. One difficulty in establishing consistency here is that there may be disagreement as to the division of the documentation text into clauses, which can result in overlaps.

3. Typing - Are developers in agreement on the types assigned to each directive? Answers to this question determine presentation and filtering. However, certain directives could belong to multiple types or to no types.

4. Rating - Are developers consistent in the ratings assigned to each directive? The choices by the creator of the directives set have a significant impact on presentation and filtering. In the client-server implementation, the ratings provided by multiple individuals are averaged out.

5. Phrasing - *eMoose* allows the phrasing of a directive in the lower pane of the *JavaDoc* hover to be different from its text in the original documentation (and the upper pane). This allows set creators to shorten or clean up the description. It is important that developers are consistent in their rephrasing of directives to avoid misleading future users who would only read the lower pane, but measurements of natural text similarity may be more difficult to obtain.

Thoroughly investigating all these questions is outside the scope of this dissertation. Nevertheless, I conducted a small study focused on the first level of granularity, attempting to answer the question: “is there consistency between developers in identifying methods with directives?” In particular, I was interested in whether there is consistency on the methods that appeared in the *eMoose* study. To this end, a small set of subjects were presented with printouts of the *JavaDocs* of several classes from the JMS and SWING APIs, and asked to tag all clauses that they considered a directive. Measures of reliability were then calculated based on which methods had at least one directive tagged.

7.2 Study design and materials

I recruited 6 subjects who *did not* participate in the lab study of Chap. 6. The first five subjects were all experienced developers currently studying towards a doctoral degree in various specialities of computer science. The sixth was an experienced developer currently working in industry, who holds an undergraduate degree.

All subjects received a study booklet with instructions that will be summarized here; the full text is reproduced in Appendix A. The instructions are intentionally short, in an attempt to see how developers would naturally comprehend the concept of directives, without training to bring them towards consistency. In particular, subjects are intentionally not exposed to an existing corpus of directives with ratings. To avoid biasing results, no additional explanations were given and no questions were allowed about the concept of directives during the study; subjects were only allowed to ask clarification questions about the API itself.

The booklet begins by explaining the purpose of the study and providing a general description of directives:

Directives (as we later explain) convey information that is particularly important for the function’s users to become aware of, as they are imperative for the proper use of the function. This is different from specifications, which constitute the majority of *JavaDoc* text, and which provide sufficient details for those interested in understanding everything about the function or in ensuring that the contract is used correctly. Both definitions, however, are somewhat amorphous, which is why we are trying to determine how different individuals approach them.

This is followed by two examples: First, `Math.random()` and the fact that multithreaded code can be made more efficient by giving each thread its own `Random` instance. Second, the `String.replaceAll()` example that we have seen before, where newer versions of the documentation warn against using specific characters in the replacement string. The text then continues:

There are many cases where directives are a lot more obvious and carry more significant implications. For example, a method's documentation may instruct the user to call another method first, to not invoke it from a certain thread, or to be responsible for releasing a handle that it obtains from the platform. Note that if something appears trivial or common it may not be a directive or may be a directive of marginal importance. For example, all JavaDocs list all the parameters and often require that parameters not be null. Since this is fairly standard, and users are expected to check for this anyway, this is not a directive. However, if there is a restriction on the concepts of a parameter, for example, then this is a directive.

Now, the booklet turns to procedures.

In this study your role will be to identify directives in the printouts of the JavaDocs for several classes. You will receive a booklet consisting of some background material about the API, followed by the actual printout. Please go systematically over the text of each method. When you find something that could possibly be a directive (even if you are not sure), use a highlighter to mark the entire text fragment that would correspond to a directive. This text could potentially be rewritten as a more concise and direct instruction, but you do not have to do so. If at doubt as to whether something is a directive or not, mark it anyway.

Note that the same method documentation may contain multiple directives, so please mark all of them. Markings should only be applied to the detailed JavaDocs of public methods and constructors, and not to the documentation of the class, its fields, or nonprivate methods. In addition, you should only be concerned with directives aimed at users of an instance of a class (or subclasses) rather than directives aimed at users who will subclass the current class or override its methods.

Note that subjects are instructed to mark anything that they initially suspect to be a directive, even if they eventually decide that it is not. The goal behind this instruction was to gain an insight at the kinds of clauses that are being considered.

Next, subjects are asked to rate each highlighted clause on a scale from 0 to 7, with a "standard" confidence of 4. A value of zero is reserved for clauses that are determined to not be a directive. Note that this range is wider than that used in *eMoose*. However, our goal is not to compute the consistency of these ratings but just to have an idea of the confidence of the raters.

Subjects are then asked to also attempt to classify all directives with one or two types from a given list of mnemonics with short descriptions, but are also allowed to skip this if that would allow them to finish tagging a larger portion of the class. Again, the goal is not to actually calculate consistency but rather to have some idea of whether there is potential for it.

Finally, the instructions also state: "We realize that the task as a whole is mundane and tiring, but please try to remain attentive. Feel free to take breaks at any point." Indeed, my impression was that subject performance degraded over time, and they tended to skim the *JavaDocs* more and miss more details.

7.2.1 APIs used in the study

Since the lab study described in Chap. 6 was based on sets of directives that I created for specific APIs, these APIs became a natural candidate for an investigation of tagging consistency. The first part of the reliability study involves several interfaces and classes from JMS which were featured in the *eMoose* study. Since there is no natural ordering for these types, I arbitrarily chose one that allowed some subtypes to immediately follow their supertypes, and that placed the hierarchy of "topic" types after that of the "queue" types. Since the two hierarchies are similar, we can check for consistency after time has passed. The selected order is: `QueueConnectionFactory`, `Connection`, `QueueConnection`, `Session`,

QueueSession, Queue, MessageConsumer, QueueReceiver, TopicConnectionFactory, TopicConnection, TopicSession, Topic, TopicSubscriber

The second part of the study, involving SWING, takes the inheritance chain starting at `JLayeredPane` and heading upwards through: `JComponent` and `Component` to `Container`. However, due to time constraints subjects were only expected to work their way up to the end of `JLayeredPane`. In practice, within the allotted time, all of them made it at least half-way through the `JComponent` class, with some reaching its end.

Note that the printouts were created using default settings, so some methods were split across pages. For SWING, we used the documentation version from JAVA 6 and for JMS from version 5 of the *J2EE*.

7.3 Results for methods used in the *eMoose* study

Before we investigate consistency for the entire set of methods, let us first examine only the results for methods that appeared in the *eMoose* study. Methods will be presented in the order in which they appeared in the printouts, though many unrelated methods appeared between them.

createQueueConnection (): The first class in the booklet is `QueueConnectionFactory`, whose only method is `createQueueConnection`. This method was decorated in our first JMS task and conveyed the solution to the bug - it mentioned that `start` should be invoked before messages can be received. Since this is the first method that subjects encountered in the booklet, it is not surprising that all six identified this clause as a protocol directive. All subjects except S6 rated it as at least 4.

setClientId (): The next class is `Connection`. One of its methods, `setClientId`, is inherited by `TopicConnection` and is the decorated method at the core of our second JMS task. The relevant directive in that task was the prohibition on invoking any other methods on the object prior to calling that method. All subjects rated at least one clause in its documentation, though ratings varied from 1 to 7.

start (): The next relevant method in `Connection` is `start`, which was the method that subjects needed to add in the first JMS task and which was presented in undecorated form in the second task. The documentation of the method mentions that repeated calls are ignored, which has led subject S2 to consider it as a protocol directive with a rating of 1 (very low). None of the others highlighted this method.

createQueueSession (): After `Connection` comes the `QueueConnection` subclass, with its `createQueueSession` method which was undecorated in the study. The documentation mentions an issue with the `AcknowledgeMode` parameter of this method, leading subjects S3 and S2 to identify a parameter directive (rated 4 and 2 respectively) while the three others did not.

close (): Next comes the major `Session` class. Its `close` method appeared decorated in the cleanup code of both JMS tasks and has been explored by some subjects. The documentation of this method is quite lengthy. It discusses different protocol and threading issues; and provides several performance recommendations. All subjects highlighted at least one clause in this method with a rating of at least 3, though the exact clauses differed and in some cases overlapped.

createQueue (): The next class is `QueueSession`, and it declares the `createQueue` method which appeared decorated in the core area of the first task. The documentation is lengthy and mentions various issues such as portability limitations, purpose, etc. All subjects identified a directive with a rating of at least 2.

receive (): Later on, we get to `MessageConsumer` and its `receive` method. This method, which was decorated in the first JMS task, is where execution had blocked. The documentation mentions two reasons for why the call would become blocked, though not the issue of calling `start`. Surprisingly,

only subjects S3–S6 highlighted this threading clause. This is interesting because S1 and S2 did tag the `createQueueConnection` method which mentions the need to call `start`. It is not clear if the difference was due to fatigue, or whether the need to call `start` seemed more relevant and surprising while the fact that message reception blocks until messages are produced seemed trivial.

`createTopicConnection()`: The next set of classes are the `Topic` analogues of the `Queue` classes seen earlier. Once again, we get `TopicConnectionFactory` with its `createTopicConnection` which is highlighted by everyone. The ratings given by all subjects except S5 are consistent with the ones previously given to the `Queue` version

`createTopicSession()`: Also in `Topic`, the parameter clause in `createTopicSession` is tagged by subjects S2 and S3 with the exact same rating as for the `queue` analogue `createQueueSession`, though it is not clear if this similarity results from consistency or from a memory effect.

`createTopic()`: Similarly, the decorated `createTopic` in `TopicSession` is decorated by everyone with a range of ratings and a pattern that is similar to the earlier `createQueue`. Again, it is not clear if this is memory or consistency.

We now turn to SWING and the `JLayeredPane` class which was investigated by all subjects. We are going to only examine the methods which were invoked in the `actionPerformed` method of our first SWING task.

`putLayer()`: The decorated method `putLayer` and the critical directive instruction to use `setLayer` instead was highlighted by everyone as an alternative-directive with a rating of between 3 and 7.

`setLayer()`: The replacement method `setLayer` which instructs callers to invoke it before adding the element to the parent container also received a consistent protocol tag from everyone.

`moveToFront()` and `moveToBack()`: Next, the undecorated `moveToFront` and `moveToBack` methods were not marked by anyone.

`setPosition()`: Finally, `setPosition` which was decorated was tagged by all subjects but S5, who considered it but eventually gave it a rating of zero. This method indicates that position numbering (a parameter) is defined by the container rather than the layer. However, only subject S3 listed this as a parameter directive, while the others did not pick a type.

Overall, when limited to the set of methods that appeared in the *eMoose* study, we see fairly high agreement between subjects with regards to whether a method contains directives or not. The main exception is `createQueueSession` and `createTopicSession`, where two subjects believed that the parameter clause constitutes a directive. Therefore, the decorations on the source code would generally be the same had these subjects collaboratively created the directive set.

Within the methods, however, the story is very different, as subjects differed in the clauses that they decorated, the types that they chose, and the rating. It is possible that more consistency can be achieved with additional training, and perhaps with encouragement to reexamine methods where at least one clause has been tagged. In particular, the ratings are very inconsistent among subjects. It is very likely that some of this inconsistency is due to the lack of training, since subjects did not have a common corpus of rated directives to which they could relate.

7.4 Consistency over the entire set

In the scope of this dissertation, I am only focused on method-level agreement: whether developers are consistent in identifying methods that convey directives and will therefore be decorated.

To this end, I took the raw tables listing the clauses tagged by each subject, and distilled them into

a matrix of binary-valued cells whose rows represent methods and whose columns represent subjects. A particular cell is “checked” if and only if the subject identified at least one directive with a rating of at least 1 in the documentation of that method. The first part of the table, covering the JMS methods, is presented in Table 7.4, The second part, which appears in Table 7.5, covers up to the point in the middle of `JComponent` which was reached by all subjects. The third part, in Table 7.6, includes methods of `JComponent` which were only reached by some of the subjects. In the columns for each subject, cells contain the highest rating given by that subject to any clause in the directive, and are empty otherwise. The inclusion of the numeric rating here is for informational purposes only, as our calculation will not take them into account.

7.4.1 Number of subjects identifying tagging a directive in each method

To help us better understand the decorations in Tables 7.4–7.6, Fig. 7.1 presents the distribution of the 105 “votes” for the methods up to the end of `JLayeredPane`. Each column represents the number of methods that were tagged with directives by the corresponding number of subjects. Columns towards the edges represent consistency while those in the middle represent inconsistencies. As we can see, 44 of the methods are unanimously considered by all subjects to not convey any directives, while 20 are unanimously considered to convey a directive. If we decided the status of a method based on a simple majority, we would have 35 methods with directives and 64 without. If we allow a single “dissenting vote” and join the two edge columns on each side, we would get 25 methods that convey directives, 55 that do not, and 25 where there is disagreement.

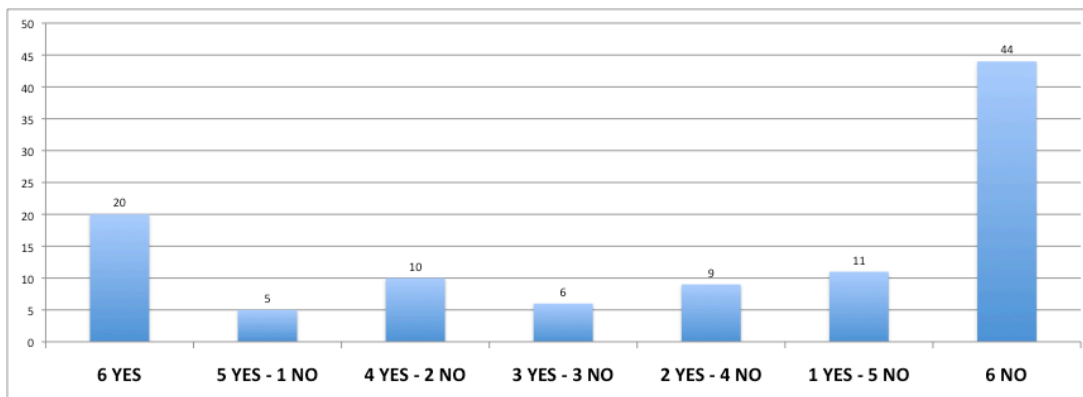


Figure 7.1: Number of methods tagged as having directives by number of subjects (up to end of `JLayeredPane`)

We note that while the number of methods with directives may appear quite high, the impact on invoking source code can be curbed by rating the directives and filtering decorations by rating. Since the given ratings highly varied between subjects, the level of agreement can instead be used to produce the rating.

If we add a portion of the methods of `JComponent` up to the point where subjects S1 and S3 dropped off due to the time limit, the picture changes somewhat. As can be seen in Fig. 7.2, subjects encountered many methods with no directives, and consistency between subjects dropped. While this may be due to a substantial difference in the API, it may also represent the effects of fatigue or time pressure on subjects. For instance, subject S1 barely marked 7 methods in all of `JComponent` before running out of time.

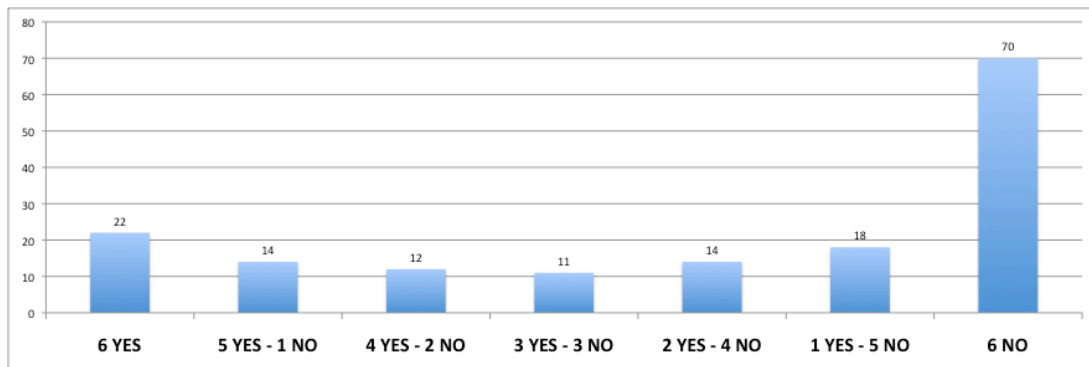


Figure 7.2: Number of methods tagged as having directives by number of subjects (up to middle of `JComponent`)

7.4.2 Number of methods with directives identified by each subject

One way to understand the inconsistency between subjects in the charts above is to examine how many methods were tagged as having directives by each subject. It turns out that in the methods of `JMS`, subjects S1 and S6 tagged just 19 and 27 methods respectively, while subjects S2–S5 respectively tagged 39, 40, 36 and 40. If we further examine the methods of `SWING` up to the point reached by everyone in `JComponent`, the total number of tagged methods for S1 and S6 is 30 and 44 respectively, while for the rest it is between 60 and 70. Such a difference in proportions clearly contributes to the inconsistency.

When asked after the study, subject S1 indicated that when considering whether a clause was a directive, he tried to take into account how frequently the method or relevant situation would be valid, and eliminated those that seemed less important. This subject was familiar with the premise of the *eMoose* tool, and felt that the set needs to be trimmed to reduce the potential number of decorations.

7.4.3 Reliability between subjects

In order to better understand the consistency between subjects or lack thereof, we calculated *Cohen's Kappa* [17] for each pair of subjects. Cohen's Kappa is a quantitative measurement of the similarity in how *two* observers assign a set of items into categories. It aims to measure the degree to which the similarity between the assignments is greater than what would result if each observer made the same number of assignments in each category, but distributed them randomly across items.

While there is no universal way to interpret Kappa values, the most common division of its values is as follows [52]: A value of 0 represents a complete lack of agreement, equivalent to a random assignment. Values up to 0.20 represent slight agreement. Values from 0.21 to 0.40 represent “fair agreement”. The range of 0.41 to 0.60 is “moderate agreement”. Kappas between 0.61 and 0.80 represent “substantial agreement”, with values up to 1.00 representing an “almost perfect agreement”. Values of 1 occur only with perfect agreement.

To calculate Kappas for our subjects, I took every combination of two columns from the tables presented above. For each method, the identification of a directive (nonempty cell) was considered to be the first category, while the lack of a directive (an empty cell) was considered to be the other category. The Kappas were then calculated in the standard way.

For the methods up to the end of `JLayeredPane`, the last class that all subjects finished, the matrix of Kappa values is presented in Table 7.1. As we can see, all kappas are at least within the “moderate

	S1	S2	S3	S4	S5	S6
S1		0.54	0.45	0.55	0.50	0.68
S2			0.50	0.57	0.63	0.56
S3				0.73	0.79	0.56
S4					0.74	0.62
S5						0.61
S6						

Table 7.1: Cohen’s Kappa values for methods up to end of `JLayeredPane`

	S1	S2	S3	S4	S5	S6
S1		0.47	0.35	0.50	0.47	0.62
S2			0.48	0.58	0.68	0.56
S3				0.64	0.69	0.51
S4					0.68	0.61
S5						0.64
S6						

Table 7.2: Cohen’s Kappa values for methods up to middle of `JComponent`

agreement” range. There is very substantial agreement between subjects `S3`, `S4`, `S5`, with the highest agreement between `S3` and `S5` bordering on the “almost perfect” category. There is also substantial agreement between subject `S1` and `S6`; both tagged the smallest portion of methods, but there appears to be significant similarity in this portion. The average of all Kappas is 0.60, just in the “substantial” range.

Table 7.2 presents the Kappa values for the methods up to the point of separation in `JComponentPane`. We can see that values are significantly lower, though resembling the ones from the previous table in their magnitudes. Even as subjects were getting tired and less careful, in all but one case there is at least moderate agreement, and in some cases substantial. The average of all Kappas is 0.565, in the upper end of the “moderate” range.

Finally, we also calculated Kappas between the four subjects who made it to the end of `JComponentPane`. As can be seen in Table 7.3, there is still substantial agreement between `S4`, `S5` and `S6`, and moderate to substantial agreement between this group and `S2`.

7.5 Discussion and threats to validity

The study presented in this chapter represents an initial exploratory attempt to determine the degree to which developers agree in the identification of directives. The ability to identify information useful to the majority of prospective developers underlies our approach of making directives more salient. Our results offer initial evidence to support this premise, at least when it comes to identifying methods with directives as these would become decorated. Subjects were remarkably consistent with regard to the methods which appeared decorated and undecorated in the earlier lab study. This helps dismiss a serious

	S1	S2	S3	S4	S5	S6
S1						
S2				0.48	0.62	0.50
S3						
S4					0.64	0.62
S5						0.62
S6						

Table 7.3: Cohen’s Kappa values for methods up to end of `JComponent`

threat to the validity of that study. Subjects were also substantially consistent at identifying methods with directives across the whole API. Significant inconsistencies, however, were found in the exact clauses marked by each subject, and in the given ratings.

As previously indicated, I suspect that many of these inconsistencies are due to the lack of training. Since subjects were not presented with any examples of a rated directives, they had no frame of reference by which to calibrate their perceptions of ratings. Furthermore, since subjects generally performed only a single pass over the documentation, they had little chance to go back and correct earlier ratings based on later decisions. Further study is needed to determine if training can indeed help achieve consistency in rating.

Another factor that likely played a significant effect was fatigue. Developers are not used to reading large amounts of *JavaDoc* this carefully, and having to do so under time constraints for two hours was likely unsettling. It was very clear that subjects did not enjoy the task, and many were visibly fatigued by the end of the first hour. This limited the care paid to each method, as was exemplified by subject S1 who started skipping large ranges of methods with only an occasional glimpse into their text. Having annotated the JDK and other libraries, I can certainly attest to a similar experience, though I carried this annotation over a long time. While these issues likely affected the study, I believe that their impact on the feasibility of large library annotation is not significant. I do not intend that entire libraries would be tagged by a handful of individuals in one stretch. Rather, as typical in Wiki based efforts, the tagging of directives should take place over time by many individuals as they notice something important and leave a mark to assist future readers. Nevertheless, a shorter follow-up study with more subjects and adequate training may be useful.

Note that fatigue may have also contributed to the inconsistencies in identifying the exact intervals constituting directives. As they made their way through the documentation booklet, subjects became careless about marking specific sentences, often marking an entire paragraph or a single word. Since the focus of this study was at the method level rather than the clause-level, I was not stringent when subjects exhibited this behavior. In a shorter study, however, it may be possible to obtain more accurate measurements about this issue.

Finally, note that there was not enough data to formally analyze consistency in typing. My impression was that for many methods there was agreement between subjects that explicitly indicated a type, but that in these cases the wording was typically very clear. When there were inconsistencies, they were typically between two or three types. Again, this would require a more formal follow-up study.

Class	Method	S1	S2	S3	S4	S5	S6	#YES	#NO	
QueueConnectionFactory	createQueueConnection()	6	6	4	5	4	2	6	0	
	createQueueConnection(String, String)	7	6	6	5	4	2	6	0	
Connection	createSession(...)			4	2	1	0	4	2	
	getClientId()							0	6	
	setClientId(...)	1	5	6	7	3	7	6	0	
	getMetadata()							0	6	
	getExceptionListener()		3					1	5	
	setExceptionListener(...)		3		1	1	1	4	2	
	start()		1					1	5	
	stop()			5	2	2	4	4	2	
	close()	2	3	5	3	4	1	6	0	
	createConnectionConsumer(...)		3	4	5			3	3	
	createDurableConnectionConsumer(...)	4	3		5	1		4	2	
QueueConnection	createQueueSession(...)			4	2			2	4	
	createConnectionConsumer		3	4	5	1		4	2	
Session	createBytesMessage()							0	6	
	createMapMessage()		3					1	5	
	createMessage()			1		1		2	4	
	createObjectMessage()							0	6	
	createObjectMessage(...)							0	6	
	createStreamMessage()							0	6	
	createTextMessage()							0	6	
	createTextMessage(...)							0	6	
	createTransacted()							0	6	
	getAcknowledgeMode()		4			1		2	4	
	commit()		3				4	1	5	
	rollback()		3					4	1	
	close()	4	3	5	3	3	4	6	0	
	recover()		2	4	4	3		4	2	
	getMessageListener()				2			1	5	
	setMessageListener()	4	4	7	5	3	2	6	0	
	run()	4	2	4	7	1		5	1	
	createProducer(...)							0	6	
	createConsumer(Destination)							0	6	
	createConsumer(Destination, String)							0	6	
	createConsumer(Destination, String, bool)		2	3		1		3	3	
	createQueue(...)	3	5	7	6	3	1	6	0	
	createTopic(...)	3	5	7	6	3	1	6	0	
	createDurableSubscriber(Topic, String)		3	5	5	4		4	2	
	createDurableSubscriber(Topic, String ...)		3	7	5	4		4	2	
	createBrowser(Queue)							0	6	
	createBrowser(Queue, String)							0	6	
	createTemporaryQueue()			6				1	5	
	createTemporaryTopic()			6		0		2	4	
	unsubscribe()		4	7	6	2	1	5	1	
	QueueSession	createQueue	3	3	7	6	3	2	6	0
		createReceiver(Queue)							0	6
		createReceiver(Queue, String)							0	6
createSender(Queue)								0	6	
createBrowser(Queue)								0	6	
createBrowser(Queue, String)								0	6	
Queue	createTemporaryQueue();							0	6	
	getQueueName()			7	6	3		3	3	
MessageConsumer	toString()							0	6	
	getMessageSelector()							0	6	
	getMessageListener()							0	6	
	setMessageListener(...)		5	6	2	3	5	5	1	
	receive()			5	1	5	2	4	2	
	receive(int)			5	4	4	5	2	5	
	receiveNoWait()							0	6	
	close()	4	3	6		5	2	5	1	
QueueReceiver	getQueue							0	6	
TopicConnectionFactory	createTopicConnection	6	6	4	5	1	2	6	0	
	createTopicConnection(String, String)	6	6	4	5	1	2	6	0	
TopicConnection	createTopicSession()			4	2			2	4	
	createConnectionConsumer(...)	3	2	6	6	1	4	6	0	
	createDurableConnectionConsumer()	3	2	6	6	1	4	6	0	
TopicSession	createTopic(String)	3	3	7	6	1	2	6	0	
	createSubscriber(Topic)		4			1		2	4	
	createSubscriber(Topic, String, boolean)		2	4		2		3	3	
	createDurableSubscriber(Topic, String)	1	2	7	5	2	2	6	0	
	createDurableSubscriber(Topic, String,...)	1	4	7	5	2	2	6	0	
	createPublisher		3					1	5	
	createTemporaryTopic()			4		1		2	4	
unsubscribe(String)		3	6	6	5	4	4	2		
Topic	getTopicName()			7	3	1		3	3	
	toString()							0	6	
TopicSubscriber	getTopic()							0	6	
	getNoLocal()							0	6	

Table 7.4: Presence of directives in JMS queue methods

Class	Method	S1	S2	S3	S4	S5	S6	#YES	#NO	
JLayeredPane	Constructor							0	6	
	addImpl()		2					1	5	
	remove()		6	2		1		3	3	
	removeAll()							0	6	
	isOptimizedDrawingEnabled()							0	6	
	putLayer()	4	6	4	7	3	4	6	0	
	getLayer()	4	6	4	7	3	4	6	0	
	getLayerPaneAbove()							0	6	
	setLayer(Component)	4	5	4	6	4	2	6	0	
	setLayer(Component, int, int)			4				1	5	
	getLayer(Component)							0	6	
	getIndexOf()		6					1	5	
	moveToFront()							0	6	
	moveToBack()							0	6	
	setPosition()	2	7	4	4	0	2	6	0	
	getPosition()							0	6	
	highestLayer()							0	6	
	lowestLayer()							0	6	
	getComponentCountInLayer()							0	6	
	getComponentsInLayer()							0	6	
	paint()							0	6	
	getComponentsToLayer()							0	6	
	getObjectForLayer()							0	6	
	insertIndexForLayer()							0	6	
	paramString()	2	4					2	4	
	getAccessibleContext()			1		2	1	2	4	
	JComponent	JComponent			3				1	5
		setInheritsPopupMenu		5			1		2	4
		getInheritsPopupMenu							0	6
		setComponentPopupMenu(JPopupMenu)		5	7		3		3	3
getComponentPopupMenu			5	4	3			3	3	
updateUI()								0	6	
setUI()								0	6	
getUIClassId()		4	2	7	7			4	2	
getComponentGraphics(Graphics)			2				4	2	4	
paintComponent(Graphics g)			5					1	5	
paintChildren(Graphics g)			4					1	5	
paintBuffer(Graphics g)								0	6	
update(Graphics g)			3	6	4	1	2	5	1	
paint(Graphics g)		6	7	7	7	7	6	6	0	
printAll()								0	6	
print(Graphics g)		3	2		2	3		4	2	
printComponent(Graphics g)								0	6	
printChildren(Graphics g)								0	6	
printBorder(Graphics)								0	6	
isPaintingTile()			2		4			2	4	
isPaintingForPrint()			3		6	1		3	3	
isManagingFocus()								0	6	
setNextFocusableComponent()								0	6	
getNextFocusableComponent()								0	6	
setRequestFocusEnabled()			3	6	1	5	3	5	1	
isRequestFocusEnabled()								0	6	
requestFocus()			6	6	6	3	5	5	1	
requestFocus(boolean)		5	6	6	6	3	5	6	0	
requestFocusInWindow()								0	6	
grabFocus()			7	6	7	5	2	5	1	
setVerifyInputWhenFocusTarget()				5				1	5	
getVerifyInputWhenFocusTarget()								0	6	
getFontMetrics()								0	6	
setPreferredSize(Dimension)								0	6	
getPreferredSize()				1				1	5	
setMaximumSize(Dimensions)			4	2		7		3	3	
getMaximumSize()								0	6	
setMinimumSize(Dimensions)			4	2		7		3	3	
getMinimumSize()								0	6	
contains(int, int)					3			1	5	
setBorder(Border)		6	5	6	2	3		5	1	
getBorder()								0	6	
getInsets()								0	6	
getInsets(Insets)			4	3	3	7	1	5	1	
getAlignmentY()								0	6	
setAlignmentY()								0	6	
getAlignmentX()								0	6	
setAlignmentX()								0	6	
setInputVerifier(InputVerifier)								0	6	
getInputVerifier()								0	6	
getGraphics()				4	2			2	4	
setDebugGraphicsOptions()				3	5			2	4	
getDebugGraphicsOptions()					3			1	5	
registerKeyboardAction()		5	1		7	1	6	5	1	
registerKeyboardAction()		5	1		7	1	6	5	1	
unregisterKeyboardAction()		5	1		7	1	6	5	1	

Table 7.5: Presence of directives in SWING methods (annotated by everyone)

Class	Method	S1	S2	S3	S4	S5	S6	#YES	#NO
	getRegisteredKeyStrokes()							0	6
	getConditionForKeyStroke()				1	1		2	4
	getActionForKeyStroke(KeyStroke)							0	6
	resetKeyboardActions()				3			1	5
	setInputMap(int, InputMap)		5		5	1		3	3
	getInputMap(int)				5			1	5
	getInputMap()		2		1		2	3	3
	setActionMap()		6		6	1		3	3
	getActionMap()							0	6
	getBaseline(int, int)		2		4	1	2	4	2
	getBaselineResizeBehavior()		2		6	1		3	3
	requestDefaultFocus()							0	6
	setVisible()							0	6
	setEnabled(boolean)		6		6	3	4	4	2
	setForeground(Color)				3	1	1	3	3
	setBackground(Color)		4		4	1	1	4	2
	setFont(Font)							0	6
	getDefaultLocale()		2		3		2	3	3
	setDefaultLocale()		2		3		2	3	3
	processComponentKeyEvent()		5					1	5
	processKeyBinding()							0	6
	setToolTipText(String)							0	6
	getToolTipText()							0	6
	getToolTipText(MouseEvent)							0	6
	getToolTipLocation(MouseEvent)							0	6
	getPopupLocation(MouseEvent)		4					1	5
	createToolTip()							0	6
	scrollRectToVisible()							0	6
	setAutoscrolls(boolean autoscrolls)		3					1	5
	getAutoscrolls()							0	6
	setTransferHandler()				3	2	2	3	3
	getTransferHandler()							0	6
	processMouseEvent()							0	6
	processMouseMotionEvent()							0	6
	enable()							0	6
	disable()							0	6
	getAccessibleContext()							0	6
	getClientProperty()							0	6
Jcomponent	putClientProperty()				4	2	1	3	3
	setFocusTraversalKeys()				3			1	5
	isLightweightComponent()							0	6
	reshape()							0	6
	getBounds()				3		3	2	4
	getSize(Dimension)				3		3	2	4
	getLocation(Point)				3		3	2	4
	getX()				2	1	1	3	3
	getY()				2	1	1	3	3
	getWidth()				2	1	1	3	3
	getHeight()				2	1	1	3	3
	isOpaque()							0	6
	setOpaque()							0	6
	computeVisibleRect()							0	6
	getVisibleRect()							0	6
	firePropertyChange()							0	6
	firePropertyChange()							0	6
	firePropertyChange()							0	6
	fireVetoableChange()							0	6
	addVetoableChangeListener()							0	6
	removeVetoableChangeListener()							0	6
	getTopLevelAncestor()							0	6
	addAncestorListener()							0	6
	removeAncestorListener()							0	6
	getAncestorListener()							0	6
	getListeners()						1	1	5
	addNotify()		1					1	5
	removeNotify()		1					1	5
	repaint(long, int, int, int, int)		3			1	1	3	3
	repaint(Rectangle r)		3			1	1	3	3
	revalidate()		3			1	1	3	3
	isValidateRoot()							0	6
	isOptimizedDrawingEnabled()							0	6
	paintImmediately(int, int, int...)		7			4	6	3	3
	paintImmediately(Rectangle)							0	6
	setDoubleBuffered()							0	6
	isDoubleBuffered()							0	6
	getRootPane()							0	6
	paramString()		2			4		2	4

Table 7.6: Presence of directives in SWING methods (annotated by some subjects)

Chapter 8

Conclusions and Future Work

In this concluding chapter, Sec.8.1 recaps the evolution of this work and the connection between its chapters. Sec. 8.2 summarizes the contributions and implications. Sec. 8.3 presents open questions and directions for further research.

8.1 Retrospection

Because the focus of my research has evolved over time, this dissertation was divided into two distinct parts, one focused on the design and the other on APIs and code. This section briefly recaps the previous chapters and how each lays a foundation for the following one.

The initial goal of my research was to identify new ways to support software design in distributed settings. Since successful examples are rare, I chose to study collocated design sessions, and identify behaviors that would be difficult to translate to a distributed medium without additional tool support. To this end I conducted an initial observational study of design exercises (Chap. 2), focused on the physical design environment and the use of artifacts. At that point, direct observation and frequently taking snapshots of the environment seemed adequate.

In the settings in which this study took place, designers had access to a large variety of drawing surfaces (“canvases”). I found that designers given this freedom tended to scatter their work across many canvases all around the environment, and used spatial and visual cues to locate them. Many canvases were also arranged into complex and ever-shifting containment hierarchies. I also observed the continuous formation and disbanding of ad-hoc teams which used gestures and gaze to establish common ground. Interestingly, much design information was never captured in permanent drawing activities, and diagrams were frequently recreated with different contents.

Many of these findings presented potential problems for the transition to distributed settings, where constraints of the electronic medium and the difficulty of establishing common ground would restrict this freedom. However, I began suspecting that these findings have broader implications for the support of design in general, even in collocated settings. The constant shifting of canvases, the dependency on contextual information, and the loss of much design knowledge due to the use of ephemeral gestures, all present a threat to the eventual use and interpretation of the design products. Indeed, looking at the photographs it was often difficult to piece together the design and its evolution. As a result, I shifted my attention to the problem of preserving context and coping with a large number of canvases and their contents.

To investigate these problems, I conducted a second and more elaborate study using video record-

ings (Chap. 3) that focused on the actual drawing activities and the representations used. Multiple teams working on similar problems were used as a baseline for comparison. The study offered many valuable insights into the representational choices that further emphasized the need to preserve traceability of the design activities. For instance, developers deliberately diverged from standard notations, combining representations and creating ad-hoc notations that evolved over time but presented interpretation challenges. They also concurrently used multiple representations to explore certain ideas, forming implicit contextual dependencies that are not evident when artifacts are examined separately.

A related finding from this study, however, was the risk of delocalization due to the use of multiple representations. Namely, as teams captured design information about an entity or concept in one artifact, they later failed to recall this information when dealing with another instance of that entity in another artifact. For instance, previously captured assumptions or decisions that were collected together were often not recalled. I suspected that this problem, which I termed “neighbor knowledge awareness”, could be generalized to other domains.

After initially exploring means to improve traceability, I shifted my focus to the knowledge awareness problem. Since collaborative design using software tools (where interventions could easily be explored) is not common, I focused my attention and the second part of this dissertation on the domain of software implementation. The artifacts were now source code fragments which either defined and documented API methods, or made use of such methods and depended on knowledge conveyed by specific clauses of the documentation. In a detailed examination of a major API, I identified many important and potentially surprising knowledge elements which I termed “directives”, and proposed a taxonomy of their types (Chap. 4).

I developed the *eMoose* tool to “push” awareness of the presence of directives in invoked methods (Chap. 5). I then conducted a detailed lab study (Chap. 6) to investigate whether developers become aware of directives, and whether they benefit or suffer from the novel technique proposed by this dissertation - of providing visual indications on the presence of directives. I found that directive awareness presents serious problems, and that the approach has potential - if a reliable set of directives can be identified. I took one step in this direction with an exploratory study which suggested that such a set could consistently be identified by multiple individuals (Chap. 7).

I have been using the tool successfully in my work in industry and have encountered several bugs that likely resulted from a developer’s lack of awareness of an important detail in the documentation of an invoked method. Unfortunately, during my doctoral studies I did not have the opportunity to polish the tool sufficiently for distribution that would have enabled field evaluation. I hope to do so in the future.

8.2 Contributions

8.2.1 Improving the understanding of representation choices in software design

The findings from the studies of software design in the first part of this dissertation make significant contributions to our understanding of the representations used in design. One important finding was that given the freedom, designers may deliberately choose to diverge from standard notations and violate their restrictions. They may also deliberately choose to concurrently use multiple delocalized diagrams and representations to explore the same part of the design, rather than try and come up with a single representation. Designers appear to do so for the immediate benefits to creativity from working with their preferred levels of structure and abstraction. In doing so, they appear willing to sacrifice both the communicative value of a shared notation and the long-term readability and simplicity of implementation of their products.

What appears to compensate for these behaviors in the short term is that designers share context, allowing them to understand the notations and implicit connections between artifacts during the session. As a consequence of these findings, I argued that it is critical to preserve contextual and historical traceability information on the work of design teams in order to facilitate long term interpretation. In other words, rather than merely support design teams by helping them draw more effectively, as proposed by prior research, I argue that we can help them focus on immediate goals by offloading the responsibility of preserving traceability knowledge.

My findings in these studies also led me to identify the neighbor knowledge awareness problem, which I then applied to code, leading to the other contributions of this work, described below.

8.2.2 Identifying and demonstrating the neighbor knowledge awareness problem in code

The importance of APIs and their documentation rises significantly as software development becomes accessible to wider audiences. Since documentation is often the only medium of communication between the authors of an API and their users, there is a near-consensus that it should be complete and thorough. There appears to be an underlying assumption that if an issue is documented, callers are likely to become aware of it.

The first novel contribution of this work in this domain is in demonstrating that such awareness is not guaranteed in typical development scenarios, and in highlighting that this can be a potential cause for major software errors. Awareness is particularly problematic in polymorphic situations, where overriding versions of a method present additional awareness concerns.

It is well known that developers do not read the documentation of every method invoked by the code they write or maintain. However, little was previously known on whether this leads to errors and on how reading decisions are made. To the best of my knowledge, this work is the first attempt to investigate the documentation reading choices made in typical development scenarios and the first to show a lack of awareness of a wide variety of important clauses.

These findings are critical because they highlight a very serious disconnect between API authors and users. If authors design their API under the premise that their users are familiar with its documentation, they may choose to avoid the seemingly redundant checks for correct use. If users then violate the contract, the resulting failures may be very difficult to trace. Function authors need to understand that users may incorrectly assume that they perfectly understand the function just based on its signature, and develop defensively against these scenarios. Meanwhile, users may be surprised to learn that they have incorrectly used methods that they had assumed they were perfectly familiar with.

My lab study also suggests that developers apply discretion in deciding which call targets to explore. The first three tasks suggested that the theory of information foraging [56] may explain the developers' behavior, and this model also fits the polymorphic situations in later tasks. Based on their experience and various cues ("scents"), developers form an estimate of the relevancy of the method, the prospects for learning new information, and its potential impact. In other words, the author of the API currently has very little impact on whether the documentation would be read. In fact, by providing an intuitive name and signature to the method, the author may make callers believe that they understand everything about the method, and reduce the prospects that its documentation be read. A key to overcoming these problems may be to provide additional "scents" via an external tool, which is the approach advocated by this dissertation and the *eMoose* tool.

8.2.3 Providing a technique to increase awareness of directives

The main and novel contribution of this dissertation is in proposing the decoration of calls in the source code as cues that their targets should be investigated, and in demonstrating the effectiveness and limited distraction from using this technique. In my lab study, decorations appeared to tilt the balance towards the exploration of methods containing important information, which were not explored by controls. The tool has also been effective at making developers aware of the presence of directives in subtypes. The technique was implemented within the *eMoose* tool which is freely available, although not tested in the field.

A major concern with any technique that adds visual cues to the already busy code editor is the risk of overload and distraction. While in some cases developers in our study did follow certain decorated calls and read documentation that ultimately did not provide value, this behavior was not common. The presence of a decoration did not lead developers to immediately explore the call's target, but rather merely added another "scent" into the decision whether to explore. Overall, subjects in our study did not spend much time on unrelated decorated methods and did not report feeling distracted. Furthermore, the costs of unfruitful exploration of certain decorated methods may not be higher than the cost of unfruitful explorations in the absence of any decorations - controls in our study spent much time on a variety of methods.

My findings also demonstrate that there is a benefit to presenting particular clauses within a method's documentation in a visually distinct way, such as highlighting or listing them separately. While recent works have proposed adding materials to *JavaDocs* [81], my findings show a need to change the formatting of the existing text. This conclusion receives support from my directive identification study, which suggests that developers can be consistent at determining which methods contain particularly important clauses and in many cases what these clauses are.

8.3 Open questions and future research directions

This dissertation demonstrated that not all documentation clauses are equal in their importance. Under certain conditions, awareness of certain clauses, which we named "directives", is more critical for the correct use of the function than awareness of other clauses. Furthermore, it also highlighted the potential of decorating calls as a way to increase awareness of directives and avoid serious errors. It is possible that more generally, decorating references can be effective for increasing awareness of delocalized information.

This section presents several important research problems that must be addressed in order to allow the proposed approach to provide significant everyday value to developers. First, there is the problem of building the directive set. Second, there is a need to filter directives based on the users' needs. Third, we need to address other types of knowledge, elements, and activities.

8.3.1 Building directive collections

Since the primary function of *eMoose* is to make callers aware of the presence of directives in invoked methods, its utility is directly related to the completeness and the quality of the set of directives provided to users.

Manual annotation

In the course of my work, I spent a significant amount of time thoroughly going over the documentation of the JAVA standard library and of *Eclipse*, and identifying directives in order to distribute *eMoose* with built-in sets of directives for these APIs. The effort was protracted and error prone; there are likely things that I did not notice, and things that I tagged that are not as important as I initially perceived. I believe that having a single individual annotate an entire API for directives in one concentrated effort is not practical, unless that person annotates his own API in the course of developing it.

For this reason, I consider it very significant that even without training and under significant time pressure, multiple developers in our tagging study were able to significantly agree on which methods contained directives. It shows that the tagging effort can be split across multiple individuals, who may be able to produce a relatively consistent set of directives. Such consistency is important for several reasons: First, it suggests that clauses that are likely perceived as valuable are indeed tagged. Second, it suggests that developers would not be generating much “noise” for one another. Third, it suggests that a collaborative tagging effort may eventually reach a “fixpoint” and that the annotating developers would not revert each other’s work. Nevertheless, I found that my subjects could not maintain sufficient concentration during tagging sessions lasting longer than an hour.

For these reasons, my vision for the tagging of directives in APIs is one of a slow evolution at multiple focal points rather than of a concentrated linear effort. This is similar to social tagging and content creation on the web. When a developer identifies a clause in the documentation that he deems important, perhaps because he had been negatively affected by a lack of awareness of it, he would tag the clause as a directive. This tagged directive would join a slowly growing community-generated set. Errors and vandalism would be mitigated by the use of the rating mechanism, as described earlier, and as is often done in the *Wiki* world. The ability of communities to create these sets must still be evaluated in the field.

Automatic detection

Suppose that project teams could collaboratively annotate directives in project artifact as they worked, and that there was a growing set of annotated third-party APIs as I envisioned. Even under these favorable conditions, at least initially, many of the methods encountered by developers would still refer to APIs or project classes that have not yet been annotated. In order to make the tool more appealing and practical for users, we need a way to automatically or heuristically recognize documentation clauses that are likely to be directives, or at least to identify methods that are likely to contain directives and deserve attention from human annotators. Such automatically generated tag sets could be used to initially present some tentative decorations. As users explore these decorated methods and rate their directives, the tags would be fine-tuned and join the manually generated sets.

I do not believe that in the foreseeable future, automated detection could be used for *all* directives. As seen in our tagging study, there are subjective differences and calls based on experience and opinion. Moreover, in my survey of APIs, some of the clauses which I determined to contain important information worthy of tagging as directives were often stated in long and convoluted sentences, or fragmented within a complex paragraph. For now, only human readers will be able to detect these.

However, it was my impression that a significant portion of directives were stated as fairly straightforward imperative sentences. For certain types, such as restrictions, many instances fit into a pattern or used a recurring phrase, such as “for debugging purposes only”. Such a subset may be more amenable to automatic recognition. Recent works [84, 83] that involved the automated analysis of specific types of comments in specific domains, such as locking in operating systems, show the potential merits of an

approach that would handle each of the types described in our taxonomy separately. These works also showed that distribution of comment types varies across APIs and the domains they are concerned with. It is possible that this fact can be further leveraged to fine-tune and automated recognition engine, for example by increasing confidence if a certain directive type is more likely in that API.

If type-based recognition could indeed be followed for many of the directives, the remaining problem would be to handle comments that do not fit any or single type. It is likely that much human intervention would be needed. However, certain patterns and phrases can help detect candidate clauses and draw these human annotators. For example, phrases like “note that” or “the caller must” are often cues that some directive will be presented.

As indicated earlier, the use of automated technique will likely result in a set of tentative directives, for which human confirmation would be necessary. Whereas an organized review process could be applied for entire APIs prior to packaging and distributing a directive set, on-the-fly recognition in project and proprietary artifacts presents more challenges. The *eMoose* tool would have to be modified to present easily recognizable decorations when these tentative directives are referenced, and mechanisms for accepting or rejecting some of these proposals. User decisions will have to globally affect other decorations of the same methods, as we now discuss.

8.3.2 Filtering directives and decorations

In the study presented in this dissertation, subjects worked for relatively short times with relatively short and unfamiliar code fragments. With few exceptions, each method was invoked exactly once and was previously unfamiliar. Therefore, with each method exploration, subjects could potentially learn something new. In real day-to-day development scenarios, developers are exposed to much more code and visit the same locations multiple times. They are likely to encounter many invocations of the same method in different contexts. One of the greatest challenges to making our knowledge pushing technique applicable for real world use is to find ways to minimize or fine tune the set of decorated methods. While in my experience of using *eMoose* only a minority of invoked methods are decorated, and there is no obligation to explore them, the numbers are still too high and can lead to too much wasted exploration efforts.

Taking past reading actions into account

One potential source of distraction in the current implementation of *eMoose* is that methods are decorated regardless of reading activity or changes in their informational value over time. Suppose that a method is decorated and leads to an important and unfamiliar directive. A first exploration would yield significant value. However, the call remains decorated. As a result, the reader would have to remember its location and ignore it. In addition, as the number of explored calls increases, spotting the few that has not been explored becomes harder, reducing the informational value of the decorations.

Eventually, however, the caller will return to the same location or encounter a call to the same method in another location. If the user did not forget the information, the call could be distracting. If the user remembers the information but not the call it was associated with, a wasteful exploration might take place.

A seemingly straightforward solution to this problem, suggested by some users who examined the tool, is to stop decorating methods once they have been explored, so that they won't distract the user in the future. One variation of this approach is to do so only locally (for a specific call or viewport) rather than for all instances of the method. Another variation is to explicitly require users to take another action to make a decoration disappear.

While this approach will clearly reduce the number of decorations, it is based on three assumptions that may not always be valid: First, that the information is appropriately consumed as soon as it is first read. Second, that once consumed, the information is not forgotten. Third, that the method and information would be recalled in other contexts.

In our lab study, subjects often explored the same method multiple times, and often did not realize the importance of the information the first time they had encountered it. It is even possible that they would have initially marked the method as read, and that this would have prevented the subsequent reading where they had realized its importance. Even if information is remembered in the short term, it is not clear that it can be recalled effectively in the longer run, as we have seen with assumptions in the design session. The developers' recollection may also be based on contextual or visual cues, so it is not clear that recollection would take place when a call is seen in another context.

A delicate balance must be maintained between wastefully presenting information that is still fresh in memory and reinforcing information that is likely fading. This work merely explored whether developers become aware of information in the first place. An important research question is for how long newly learned knowledge about methods can be effectively recalled, and how this memory can be reinforced. While the findings may help fine-tune *eMoose*, especially given the ability to adjust the contrast of method decorations, they are important to provide a better understanding of software errors and ways to avoid them.

By examining the records of which decorated methods were eventually explored by a developer, it might be possible to fine tune the saliency of other decorations. For instance, if a developer rarely examines decorations on constructors, it might be preferable to reduce the contrast of decorators on other constructions that he may encounter. This more generally relates to the behavior of the developer and the context in which exploration decisions are made, as described below.

Using code context to filter presented directives

One of the questions raised in the above discussion is whether information learned in one context would be recalled in another. A related and more general question is whether we can identify what information would be more relevant in specific contexts, and appropriately adjust our presentation.

When I tagged directives in my survey and when others did so in my tagging study, we did so in a context-free manner. That is, we used our experience as developers to evaluate whether there are likely to be situations where a particular clause could be important. The current implementation of *eMoose* is also "context-free". The algorithm merely checks that the target of a method call has an associated directive, rather than considering the semantics of the call or the context of invocation.

Calling context clearly has an impact on relevance, as at the extreme certain directives are completely irrelevant in specific contexts where the code has certain characteristics. For example, in code that is single-threaded, locking or threading directives may not be relevant. The writer of a unit test or of some internal mechanism may not care about directives warning users that a method is only for internal use. The directive itself may convey certain conditions which may not match the calling context. In our study, many of the directives in JMS referred to a form of administrative configuration that was not relevant, causing distraction. In some cases, relevancy may depend on the goals and attitudes of the developer. For instance, performance related directives may not be relevant when writing tests or prototype code, but may be extremely critical in time-sensitive applications.

In most cases, context likely affects the relative importance of a directive rather than whether it should be decorated at all. Our ability to change the contrast of the decorations based on calculated ratings can be used to emphasize specific calls in specific contexts. For this purpose, however, it is not sufficient to rely on collaborative filtering and user-provided ratings, as users would have rated the directives in

the context that they had deemed important. Rather, we need some way of automatically calculating relevance in a specific context.

Accomplishing this feat promises great rewards but presents very serious challenges. First, there is a need to recognize and model the semantics of the directives and their qualifications. This goes beyond simple natural text recognition to detect the directive in the first place. There is a need to understand the purpose of the directive and identify terms related to the context, such as “administrative configuration” in the JMS API. Second, there is a need to model the current context, both in terms of local static code properties such as threading characteristics and more so in terms of the program and likely invocation scenarios. Moreover, there is need to identify and model the characteristics of the software and current goals of the developer. For instance, is this mission critical code or testing code? Is performance or security currently a priority? Third, there is a need to associate the information from the model of the directives with the model of the user and the code context.

While these challenges are significant, recent advances show some promise. Several researchers have been modeling the activities of developers, identifying specific forms of activities and mindsets. Meanwhile, development processes and tools such as *Mylar* have enabled a workflow where tasks and intents are stated *before* work is begun rather than after it is completed at the point of committing the check in. I believe that task activation could provide one important cue that could be used for identifying the current goals and perhaps some context properties.

Finally, one noteworthy property of a code context is whether the instruction conveyed by a directive is followed or violated in the context of a specific call. For instance, if the documentation of `bar()` requires that `foo()` be invoked first, there is little value to decorating the call to `bar()` if it is indeed preceded by a call to `foo()`. The ability to model protocols and other invocation rules and then statically check them against the program is a very active research field. It is therefore worth investigating whether some directives can automatically be converted into existing formalisms for which a checking tools exist. Nevertheless, it is important to remember that directives are inherently natural text elements and are therefore easier for users to create than some formal specification.

8.3.3 Pushing other types of knowledge and working in other domains

This work has demonstrated that developers may fail to become aware of important information in the documentation of invoked methods, and that decorating references can be effective. However, I believe that these findings can be generalized, and there is other information which may be worth “pushing”, both in the current domain of JAVA source code and in other domains.

Other knowledge in methods

Methods are the core building blocks of JAVA programs. The focus of this dissertation had been on “public” information - the header documentation of API methods and the directives they conveys. However, methods can be associated with other important knowledge fragments of which callers should be aware, including private internal ones and public and delocalized ones.

One of the characteristics of mature and widely available APIs is quality documentation that conveys complete specifications in which directives may be present. While in projects that are actively being developed many classes may serve as APIs for other parts of the program, the quantity and quality of header documentation is generally lower. These methods may fail to convey all the relevant invocation knowledge, thus presenting less opportunities for our technique to be useful.

However, these methods are also more likely to be incomplete or not sufficiently robust compared to

methods of published APIs. For instance, they may be inefficient or unsafe, not handle certain cases, or impose certain restrictions or assumptions that may be eliminated in the final version. Since these limitations are not part of their intended behavior, this information will likely not appear in the header. It may, however, be documented internally as comments near the offending code. In my experience I have frequently encountered internal documentation that describes limitations of specific constructs. In many cases, this serves as a reminder to that developer or to other maintainers that more work is needed,. Since it may be indicative of a surprisingly incomplete implementation, this information may be critical for clients who are invoking the method and would not otherwise think of reading its source code.

Similarly, studies [79] show that many methods contain *task comments* such as to-do reminders that persist for long periods of time. Some developers prefer these to the more significant and visible external bug reports. Unless a systematic survey of the open task comments is carried out, however, they may not be noticed or addressed until the code is inspected or until a failure occurs. Again, callers may benefit from knowing that a method contains an open action item, and this may even prompt them to address it earlier.

The studies also show that some task comments serve as a form of communication between developers. A comment may direct a specific question at specific developers, but may not be noticed until the code is inspected. Developers may benefit from knowing that a method they invoke requires their attention.

The above forms of information can be considered internal or private as they are not part of the published interface of the method. A natural direction for research is to evaluate the extent and severity of awareness problems related to such knowledge. If this problem is significant, we can then explore whether knowledge pushing can be effective. Recall that directives are just one form of knowledge item which *eMoose* is design to push. The existing implementation already recognizes to-do comments and decorates referencing calls. Similarly, tagged comments inside the source code can be used to form KIs, and invoking methods would be decorated to attract attention.

While this dissertation has focused on information in the source code, information on program elements may also be located in other mediums. First, many project support tools, such as bug databases, may convey important information about a method, such as a known fault or remaining action item. In open-source projects and APIs, much information may also be present on the web, such as a community-maintained list of caveats or best practices. This information is, of course, delocalized when the method or a reference to it examined. With tools such as *HipiKat*, one could actively search for this information. An important question is whether we can draw in relevant information and filter it effectively, in order to provide callers with awareness of the most important information.

Other program elements, languages, and paradigms

In the scope of this dissertation I have only addressed JAVA methods. These are the building blocks of all JAVA programs, and calls to these methods constitute clear references and contexts in which information may be needed. Based on my examination of APIs and my development experience, I believe that other program elements in JAVA do not present the same kind of delocalization problem. Fields are rarely accessed directly in JAVA, as they are typically encapsulated by accessor methods. Constants are sometimes documented, but in general I have not seen many cases where detailed and surprising information was associated.

Classes, however, present a unique challenge. While the documentation of most classes is merely descriptive, I have encountered classes that convey critical and surprising information. In many cases, however, this information was global in nature, and could apply to many methods. For instance, the class documentation could describe threading requirements or performance caveats of a group of methods. In

other cases, the documentation describes the correct protocol for carrying out certain actions, citing a sequence of multiple methods. The problem with classes is that they are referenced many times: first by name in the declaration, and then indirectly via the object on which methods are invoked. The points at which we could present a decoration, such as the parameter listing of a method, could be distant from the point where the information is relevant. Furthermore, suppose that a class describes the correct way to carry out a certain action: how would we determine if this action is even attempted in a code fragment? In my opinion, most relevant knowledge could be manually attached as a knowledge item to existing methods of the class.

A more interesting avenue of exploration, in my opinion, is how to apply this technique to other paradigms. Many new object oriented languages, such as *Python* and *Ruby*, are dynamically typed. Objects are not declared with a type, so the target of a call and its legality may change over time. This presents two significant challenges. First, when examining source code, the call may have multiple and unrelated targets. Second, it would be a lot more difficult to heuristically determine the possible types of the object. The popularity of the web also brought on new forms of APIs, such as RESTful and XML based APIs. These are not documented or invoked in the traditional way, presenting new challenges to our knowledge pushing technique.

Other domains

This dissertation has begun with the intent of supporting collaborative design, and we have now come full circle. We have seen anecdotal evidence that awareness of delocalized knowledge may be important in design. We have also seen clear evidence that it presents a challenge in implementation. Though outside our scope, the next step to completing this circle would be to evaluate the severity of this problem in the domain of design, and identify ways to support it. Electronic design tools present many ways to visually indicate the presence of delocalized knowledge. There are, however, two challenges. First, we must identify a way to automatically tie instances of the same entity, even if different names or notations are being used. Second, whereas method calls typically refer to a single target, the identity relation can tie many different instances of the same entity in different artifacts. We must identify ways to determine what information should be pushed, and how to combine information from multiple sources to facilitate its consumption.

Appendix A

Reproductions of booklets used in our studies

A.1 Materials from the *eMoose* lab study

A.1.1 Introduction booklet

Documentation Use Study – Preliminaries, Uri Dekel, CMU

Preliminaries

Welcome and thank you for participating in this study!

The goal of this study is learn about the way in which documentation is used in software maintenance, and the impact of certain modifications to the standard JavaDoc mechanism on these practices.

The plan is for the study to take up to 2.5 hours

- Various preliminaries (About 15 minutes)
 - Reading this form
 - A short test of Eclipse familiarity (prerequisite for participation)
 - Reading and filling the consent forms
 - Tutorial on the eMoose tool
- Six maintenance tasks (15 minutes + 5min for reading for each task)
- Debriefing questionnaire (about 15 minutes)

You will be paid 25\$ for your participation in the study.

You will also participate in a prize raffle based on your performance: you will receive a number of raffle tickets equivalent to the number of tasks you solve on time. Please keep the tickets until the end of the study (Sep 08).

Please note: Some of the tasks in this study will be repeated by multiple subjects. **You are asked not to discuss the nature or solution to these tasks with anyone else to avoid contaminating our subject pool.** Since compensation will not depend on performance, there is no benefit to others in knowing the solutions in advance.

Eclipse Usage Test

The study compares the influence of certain additions to the standard Java tooling of the Eclipse IDE and therefore depends on the subject's prior familiarity with these features certain concepts of object-oriented programming. To continue with the study, you will be asked to successfully complete the following test within five minutes.

Failure to pass this test will prevent you from participating in the study. You will receive a compensation of 3\$ for your time.

Below is a sequence of steps that you must carry out. Answer questions verbally. Note that some activities can be carried out with keyboard shortcuts or via menus; both are ok.

- Open the project `edu.cmu.contextstudy.tutorial`
- Go to package `edu.cmu.contextstudy.tutorial`
- Open the source code editor for `Driver.java`
- Reveal all the methods defined in `Driver` without leaving the editor
- Reveal all the methods defined or inherited into `Driver` without leaving the editor
- Move the insertion point to the `run()` method WITHOUT manually scrolling.
- Reveal the type hierarchy for `B` without leaving the code editor
- For the variable `o` in `Driver.run()` and each of the classes `A, B, C, D, E`, are they a supertype, a static type, or a possible dynamic type? [Let the experimenter know if you are not familiar with these terms]
- Reveal the JavaDoc for the call to `foo` in method `run` without leaving the method.
- Open the source code of `foo` in the static type of `o` without leaving the editor.
- Open the source code of `foo` in class `C` without leaving the editor.
- Find all references to class `C` without leaving the editor.
- Close all editor windows and the tutorial project

JavaDoc and eMoose tutorial

Standard JavaDoc mechanisms in Eclipse

The only documentation with which this study is concerned is JavaDoc documentation, with which you should be familiar from Java experience. Throughout the study, you will have access to the web-based versions of the JavaDocs of the APIs you will be using. However, all the information you need is available using the JavaDoc support in Eclipse, which we shall now cover.

- Open again the `edu.cmu.contextstudy.tutorial` project
- Open the source code editor for `Driver.java` and scroll to method `main`.
- Spot the call to `Math.random` and hover over the class and then over the call, revealing the JavaDoc for each.
- Repeat the last action, but press F2 to receive a resizable window with all the details. Note that you can now scroll and reveal missing text.
- Look at the JavaDoc view (unhide it if necessary) and see that it reflects the last item you clicked on or the last movement of the insertion point.
- Click on the calls to `foo` and `bar` in `run()` and note the JavaDoc that appears with the tooltip and with the JavaDoc view: it refers only to the static type.
- Add a second call to `bar()`; note how a tooltip briefly appears as you are making the selection using the completion suggestion.

Tagged JavaDocs

In each of the six tasks you will be performing, you will be assigned to one of three conditions: standard, tags, and eMoose. In the standard condition, you will be encountering standard JavaDoc using the Eclipse mechanisms described above.

In the other two conditions, you will be using “tagged JavaDocs”. These are JavaDocs that for some methods have been manually annotated by the experimenters. The annotations consist of one of more lines at the end of the JavaDoc block, each with the form “@tag TYPE.SUBTYPE: TEXT”. In most cases, the type is “usage” to indicate that these are instructions to the client. The subtype can vary, but examples include:

- Restriction – When (and when not) to call the method
- Protocol – Indicate some sequence of actions in which this call must or must not take place
- Parameter and Return – Restrictions or guidelines about the parameters or returned items.
- Limitation – Some way in which the method will not perform.
- Sideeffect – Some unexpected potential behavior

In most cases, these tags would be based on text in the method’s JavaDoc, often verbatim or with minor changes. The original text is often highlighted in the original with prefixes like “note that”. However, you should not assume that all important information has been tagged. To understand the context of a tag, you may want to read the original JavaDoc text.

It is also important to note that the tags were created for entire APIs prior to the selection of tasks for the study. Therefore, the tags are not necessarily relevant to your current task; they may help you spot something you may miss otherwise, or they may distract you.

Documentation Use Study – Preliminaries, Uri Dekel, CMU

Finally, note that the tags will not appear in the web version of the JavaDocs.

Using eMoose

In certain tasks, some functionality from the eMoose tool will be activated. When that is the case, the tool will continuously scan the source code currently shown in the viewer and identify method calls. If the JavaDoc of a potential target of a call has a tagged comment (as described earlier), a box will surround that call. You can then use the hover mechanism to reveal an eMoose tooltip. The upper half of the tooltip consists of the original JavaDoc (with tags). The lower half consists of a “tree” showing the class name, invoked method, and the associated tagged comments; you can use F2 to zoom.

Note that as with tagged comments, the tagging took place before the tasks were created, so not all boxes and not all the tagged comments within them are going to be relevant to your current needs.

Finally, note that eMoose will create a box even if just a possible dynamic type contains a tagged comment, and the tree will reveal all these comments. For example, after activating eMoose (ask the experimenter to do so), hover over foo() and bar(). Note that eMoose does not try to identify which subtypes are likely to be dynamic types in the situation; it includes all the subtypes. However, the tree hides branches with no tagged comments. This can create a disparity between the upper and the lower half of the tooltips: the upper part only shows the JavaDoc of the static type, while the lower part shows observations from dynamic types as well.

More procedures

During the study, you will perform a series of code maintenance tasks. For each task, you will be shown a codebase, some background information, and requirements of something to accomplish, typically with a starting point and time limit. You will be given time to explore other parts of the code, and when you are ready you will be moved to the starting point. You may be asked to “Think aloud” and describe your activities to the experimenter.

Note that some of the code may use libraries and APIs with which you are not likely to be fully familiar. This is part of the study, so please let the experimenter know if you have used any of them before.

In some tasks you will be given access to the eMoose tool, while in others you will use the only standard facilities of the environment. You are not required to use the tool’s recommendations, and you must be aware that in some cases they may be helpful while in others they may not. You will be given access to eMoose in half of the tasks; do not use it in others.

You will be allowed to use a web browser located on the second monitor, although it is not necessary for the study: all the information you seek can be found within the IDE.

Unless specified otherwise, you will not be allowed to execute the code or run the debugger. The study is focused on the understanding of code, and the instructions will typically point you to the appropriate code section, making the use of the debugger superfluous.

After the study, you will be asked to answer a questionnaire.

Note that to enable analysis of your strategy and tool use, the screen will be recorded along with voice using the microphone on the table.

A.1.2 Tasks booklet

Documentation use study – Task booklet

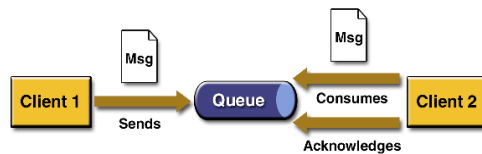
Introduction to the JMS API

What is JMS?

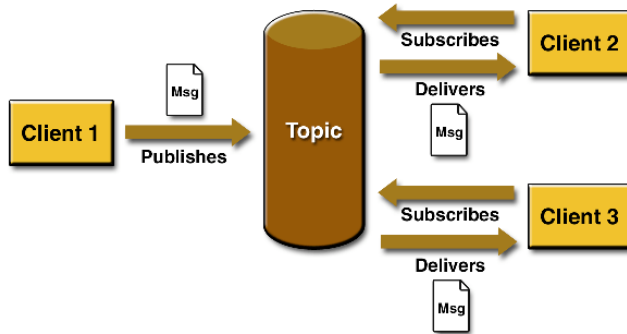
The two tasks which will be described below are concerned with the Java Messaging Service (JMS) API, which is published by Sun as part of J2EE to allow applications to create, send, receive, and read messages across processes and machines, while hiding underlying networking details.

JMS defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with various messaging implementations. JMS is reliable, ensuring that messages are delivered once and only once. Note that Sun only publishes the JMS API, while multiple vendors or (“providers”) publish implementations (e.g., Apache ActiveMQ).

JMS can operate in a point-to-point (P2P) manner through a `Queue`, where each message has only one `Consumer` and there is support for acknowledgements. The `Queue` ensures that there are no timing dependencies: the `Sender` can send before the `Receiver` is running, and the receiver can receive after the sender is done running; material is deposited in a queue.



JMS can also operate in publish/subscribe mode through topics, in which one client posts messages to a `Topic`, which multiple subscribing clients can receive. Unlike P2P, the same `Message` can have multiple consumers. There are, however, timing dependencies, as a client only receives messages posted after its subscription, and stops receiving them when the subscription is terminated. In other words, unlike P2P, subscribing to a topic to which messages have previous been sent **does not** deliver the messages.



Programming model

JMS uses a *broker*, a process located on port 61616 on a certain machine which keeps tracks of queues and topics and manages the messages in them. That process will already be active during the study. Client programs will use abstractions of these entities using names, so a queue or a topic is actually physically maintained in the broker.

The core use of JMS involves several objects described below:

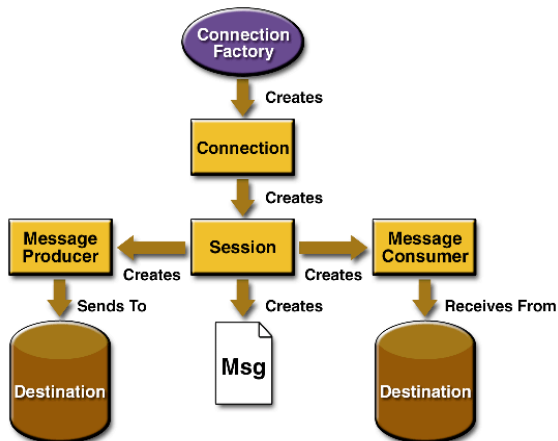
Both `Sender` and `Receiver` use a `ConnectionFactory` to create a `Connection` object that corresponds to the selected JMS implementation.

Connections are used to create a `Session` (different types for queues and for topics) that represent an atomic unit of work; sessions are responsible for creating message producers, consumers, destinations, and the messages themselves.

The `Destination` represents a `Queue` or a `Topic` from the point of view of a `Session`; it is “created” in each session. The actual implementation and storage of these structures is in the broker process.

The producers and the consumers are associated with specific destinations and used to time sending and receiving.

There are many types of messages, which can receive various metadata; they are created by the sender.



Preparation for the JMS tasks

Begin by opening the project `edu.cmu.contextstudy.jms`, and open the `jmssamples` and `testsuite` packages. The former contain a variety of sample programs that use JMS, while the latter contains some of the `JUnit` tests you will be using in this study.

To simplify execution and support `JUnit`, we wrapped some of the example programs (which have `mains`) with a `TestHarness` thread that runs each method `main` in a separate thread, and fails the `JUnit` test if the underlying sample program quits unsuccessfully or if it hangs. If you wish, feel free to examine `TestHarnessRunnable` in the `testsuite` package to see how it is implemented, although this is not necessary to your task.

The three tasks you will carry out simulate the last phase of a debugging scenarios: you have already investigated potential causes for a problem, narrowing it down to a specific code section, which you will have to inspect to determine what the problem is. You are asked to show proof for the fix, by pinpointing why it occurs and demonstrating that making certain changes will solve the problem.

First JMS task

[Make sure eMoose is DEACTIVATED]

Open the `SynchSenderAndReceiverTest` class in the `testsuite` package. As you can see, the test involves executing two threads, a sender and a receiver. Both are started with the same queue name, and the sender is started with a parameter of 20, indicating that it should send 20 messages to the receiver.

Running the test shows you that the sender thread sends 20 text messages (announcing each); it then sends one nontextual message (to indicate that sending is complete) and terminates successfully. The receiver thread, however, is silent and does not terminate. Feel free to examine the sender if you wish. However, the problem is in the receiver side.

[ACTIVATE eMoose if asked]

Now open the file `SynchQueueReceiver` in the `jmssamples` package. It contains only the main method. If you ran the debugger, you would find that the thread blocks at the call to receive in line 110. Find out why and fix this; your time starts when you scroll to that line.

Second JMS task

[Make sure eMoose is DEACTIVATED]

Open the `DurableSubscriberTest`, and see that this time we only have one thread, which invokes `DurableSubscriberExample` with a certain topic name.

Recall that unlike P2P, in standard publish/subscribe a subscribing client only receives messages posted while it is active. If a subscription is stopped and then restarted, all interim messages are lost. This is sometimes a problem, so JMS also supports durable subscriptions; these are named, and the JMS provider stores the messages even if the subscriber is currently closed. The `TopicSession` can then be used to create a durable topic subscriber, which receives any pending messages whenever it is started.

This concept is demonstrated by the `DurableSubscriberExample`, which defines internal classes named `DurableSubscriber` and `MultiplePublisher`. The main program looks like this:

```
public void run_program() {
    DurableSubscriber durableSubscriber = new DurableSubscriber();
    MultiplePublisher multiplePublisher = new MultiplePublisher();
    durableSubscriber.startSubscriber();
    multiplePublisher.publishMessages();
    durableSubscriber.closeSubscriber();
    multiplePublisher.publishMessages();
    durableSubscriber.startSubscriber();
    durableSubscriber.closeSubscriber();
    multiplePublisher.finish();
    durableSubscriber.finish();
}
```

Running the program should produce the following output:

```
Starting subscriber
PUBLISHER: Publishing message: Here is a message 1
PUBLISHER: Publishing message: Here is a message 2
SUBSCRIBER: Reading message: Here is a message 1
SUBSCRIBER: Reading message: Here is a message 2
PUBLISHER: Publishing message: Here is a message 3
SUBSCRIBER: Reading message: Here is a message 3
Closing subscriber
PUBLISHER: Publishing message: Here is a message 4
PUBLISHER: Publishing message: Here is a message 5
PUBLISHER: Publishing message: Here is a message 6
Starting subscriber
SUBSCRIBER: Reading message: Here is a message 4
SUBSCRIBER: Reading message: Here is a message 5
SUBSCRIBER: Reading message: Here is a message 6
Closing subscriber
```

[ACTIVATE eMoose if necessary]

Unfortunately, there is some problem in the constructor of `DurableConstructor` which starts at line 119 that will cause it to crash. Identify what the problem is and fix.

Introduction to the Swing LayeredPane

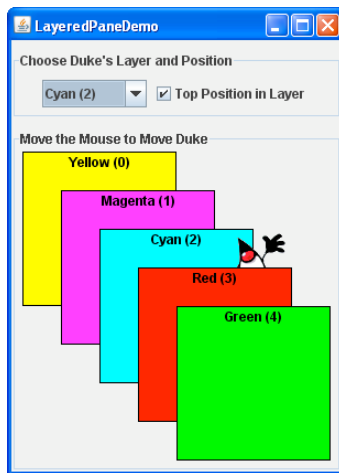
The standard java library provides two different toolkits for creating user interfaces.

The Abstract Window Toolkit (AWT) was the first, and provides a basic set of UI components. These components are often implemented with the native widgets of the platform, so look-and-feel may be different across programs.

The Swing library is a later toolkit built, in part, on top of the AWT. Swing offers a richer set of widgets, and allows components to have the same consistent look-and-feel across platforms.

Dialog and windows in AWT and SWING consist of a hierarchy of components that contain additional components and so on. All containers in AWT are subtypes of `Container`, and containers in Swing are also remote subtypes, often via `JPanel` (a lightweight container) or `JLayeredPane`. In Swing, components can be added or removed to the container, and they are then laid-out automatically by a layout manager.

The two tasks that follow make use of `JLayeredPane`, a swing container that not only accepts internal components, but allows them to be arranged in “layers” so that objects in one layer are occluded by objects in higher layers. For example, in the picture below, each colored box is in a different layer within the layered pane titled “Move the Mouse to Move Duke”. Duke the Java mascot is currently in the second layer. When moved around, he will appear in front of the yellow and magenta layers, but be hidden behind the red and green layers.



Documentation use study – Task booklet

Note that components in a layer have an int position in the range $-1..N-1$ where N is the number of components in that component's layer. -1 always represents the lowest (bottom) position while 0 represents the highest (i.e., $N-1$). In the picture above, duke is in position 0 and is therefore visible above the cyan box. If the checkbox is ticked, his position will change to -1 , and he will appear behind the cyan box, though still in its layer and above the magenta and yellow. Ticking it again will reverse the effect.

First LayeredPane task

[DEACTIVATE eMoose]

Open the `edu.cmu.contextstudy.jdk` project and the corresponding package, then open the `LayeredPaneDemo.java` file.

Run the program; the window that opens should correspond to the picture you saw earlier. Start by moving Duke around and see that the behavior corresponds to expectations. Uncheck the checkbox, and see that Duke is now behind the cyan. Check it again and see that things are back to normal.

Now use the listbox to shift to a higher layer and see the impact. Now switch to a lower layer and move duke around; unfortunately, he did not move back to the lower layer. Things are only fixed if you check and uncheck the on-top checkbox, after which things work as expected. Clearly, something is wrong.

[ACTIVATE eMoose if necessary]

Your task: identify the source of the problem, and fix it. The time starts when you first see the program.

Note that if you do not find the problem within a certain timeframe (not disclosed), the experimenter will narrow your focus to a specific method.

Second LayeredPane task

[DEACTIVATE eMoose]

Open and run `LayeredPaneExampleDual`

To better demonstrate the idea of position and make the example more customizable, the `LayeredPaneDemo` class has been reworked. It now includes two layered panes, one in which Duke will appear on top, and one in which he will appear on the bottom (moving the mouse in one box moves a Duke in the second one as well). Instead of the original checkbox, two new checkboxes now appear, controlling whether each of the two panes appears in the window. An additional checkbox controls whether the control panel appears at the top or at the bottom of the window.

All changes to any of the three checkboxes are now routed to the empty method at the end of the program. Please implement it so that all components are shown and arranged as necessary based on the checkbox states, which you can check with `isSelected` on the objects `controlPanelOnTop`, `useOnTopPane`, and `useOnBottomPane`;

Apache Commons Introduction

Even though the standard Java library is quite extensive, it does not offer many of the reusable abstractions that many larger projects need. To this end, the Apache foundation which operates many java based projects created a project called Apache-commons to provide such components to its own projects and the community. There are multiple subprojects, such as digester (which reads XML straight into Java), beanutils for utilities to beans, collections, etc.

Our focus here will be in the collection frameworks that provides additional types of collections not provided by the standard library.

Important note: As of Java 5, much of the Java collections library has been converted to using generics (templates). Apache commons has not. In this study, you will see collections used without specifying a type. This may cause some warnings but has nothing to do with the study or the problems you will be facing. You should also ignore any warnings about deprecations.

First collections task

[DEACTIVATE eMoose]

Open the project `edu.cmu.contextstudy.apachecommons` and open the package `edu.cmu.contextstudy.apachecommons.testsuite`.

You will be working on the `StandardCollectionsTest` class (don't open it yet) which contains a JUnit test.

The test setup operates as follows:

- It creates an array of Collections containing the following collections:
 - `HashSet` – The standard Set implementation from the JDK
 - `PriorityBuffer` – An apache-collections collection that offers unique element retrieval order
 - `TreeList` – An apache-collections list implementation using a tree
 - `HashBag` – An apache-collections implementation of a Bag interface (counts instances of each value)
 - `Vector` – The standard JDK vector implementation
- It runs a loop 200 times, in which it randomizes a number in the range of 0 to 30, and then adds it to each of the collections with their “add” method.

[ACTIVATE eMoose]

Now open the file and examine the `testMutualRetainment` method that runs after the setup. When the program is run, the first assert statement fails.

Without making changes to the program or running the debugger, identify the cause of the problem, though you are not asked to fix it. You have to show a reliable proof for the cause.

Second collections task

[DEACTIVATE eMoose]

Open the project `edu.cmu.contextstudy.apachecommons` and open the package `edu.cmu.contextstudy.apachecommons.testsuite`.

You will be working on the `StandardMapsTest` class (don't open it yet) which contains a JUnit test.

The test setup operates as follows:

- It creates an array of Map objects containing the following maps:
 - `HashMap` – Standard java map, implemented via `Set`
 - `TreeMap` – Standard java map, operates on comparable keys
 - `DualTreeBidiMap` – An implementation of the `BidiMap` interface defined by apache collections, maintains bidirectional mappings (i.e., one can ask for a map in the other direction)
 - `MultiHashMap` – An implementation of `Map` defined in Apache collections that allows multiple values for a key by having each key map to a list of values rather than the value itself.
 - `DoubleOrderedMap()` – An implementation of `Map` in apache-commons that implements the standard `Map` interface using a red-black tree, which allows keys and values to be stored sorted, facilitating access to both.
 - `ListOrderedMap` – An implementation of `Map` in Apache collections that also retains the order of additions.
- It creates two arrays of size 200, a labels array and a numbers array.
- It runs a loop 200 times, in which it randomizes a number in the range of 0 to 30. The number is placed in the corresponding place in the numbers array, while the corresponding labels array simply contains “label #x” where x is the iteration number. In other words, labels do not repeat themselves, but numbers do.

Now open the file and examine the `testMapPopulation` method that runs after the setup. When the program is run, the loop would fail (assertion or exception) at two iterations.

Without running the debugger, identify which ones and why, and show proof.

A.1.3 Debriefing questionnaire

Documentation Use in Software Maintenance - Study Recap Survey

Documentation Use in Software Maintenance Recap Survey (ver. 2)

Subject number: | _____ |

In order for us to better understand the strengths and the weaknesses of the eMoose tool, please answer the questionnaire, which consists of 27 closed questions and 5 open ones. For the closed questions, please state your agreement or disagreement (-3 for strongly disagree, +3 for strongly agree, 0 for neutral; skip if no opinion).

Note that “sometimes” means “occasionally” or “I can think of a few instances”, whereas “usually” can be interpreted as “occasionally not”, or “I can only think of a few instances where not...”

Marking Calls (7 questions)

(1a) eMoose sometimes offered significant help in identifying interesting calls

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(1b) eMoose usually offered significant help in identifying interesting calls

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(1c) eMoose sometimes distracted me by making me look at the wrong calls

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(1d) eMoose usually distracted me by making me look at the wrong calls

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(1e) eMoose marked too many calls

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(1f) I tended to look at marked calls first

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(1g) I ignored some calls because they weren't marked

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

JavaDoc Hover (10 questions)

(2a) eMoose sometimes offered significant help with the lower pane

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(2b) eMoose usually offered significant help with the lower pane

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(2c) The redundancy of information in the lower pane was distracting

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(2d) I tended to read the information in the lower pane before reading upper pane

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(2e) I decided whether to read the upper pane based on the lower pane

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(2f) I avoided reading the upper pane when there was information in the lower pane

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(2g) It was straightforward to correlate observations to documentation sentences

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(2h) It would be better to mark sentences in the JavaDoc than to have the separate pane

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(2i) I occasionally missed something important when reading the documentation narrative

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

(2j) I would have liked to see all observations overlaid on the source code instead of having to explicitly hover over calls surrounded by boxes.

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

Dynamic types (3 questions)

Dynamic type = The type of an object during runtime which can be a subtype of the declared (static) type. E.g., `Set mySet = new TreeSet()` and `Set mySet = new HashSet()`;

3a. eMoose helped in finding information in dynamic types

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

3b. eMoose distracted me by showing too many possible dynamic types

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

3c. I have encountered similar situations with differences in dynamic types before

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

General (7 questions)

4a. The tasks were challenging

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

4b. The activities were similar to what I often do in my everyday use.

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

4c. To accomplish tasks I had to read documentation significantly more carefully than I am used to in everyday use

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

4d. I prefer to read JavaDocs in HTML form

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

4e. I prefer to read JavaDocs in HTML form

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

4f. I prefer command line tools and editors like emacs to IDEs like Eclipse

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

4g. eMoose may be useful in my everyday use

Strong disagree -3 -2 -1 0 +1 +2 +3 Strong agree

Documentation Use in Software Maintenance - Study Recap Survey

Open questions about eMoose

Please answer the following questions briefly:

1a. How did you decide which calls to look at if several calls were marked?

What was the most useful thing you got from eMoose?

What was the most serious distraction that eMoose has caused?

What did you like about the tool?

What did you dislike about the tool?

May we contact you in the future for additional questions about this study, and how?

Thank you for your participation !

A.2 Materials from the directive tagging study

A.2.1 Introduction booklet

Directive Tagging Study – Booklet, Uri Dekel, CMU

Introduction

Welcome and thank you for participating in this study!

The goal of this study is to determine the level of consistency among individuals in tagging certain clauses, which we call directives, within the documentation of Java-based API methods. This is necessary in order to determine whether a community of developers can effectively tag directives in large libraries.

Directives (as we later explain) convey information that is particularly important for the function's users to become aware is as they are imperative for the proper use of the function. This is different from specifications, which constitute the majority of JavaDoc text, and which provide sufficient details for those interested in understanding everything about the function or in ensuring that the contract is used correctly. Both definitions, however, are somewhat amorphous, which is why we are trying to determine how different individuals approach them.

For example, consider the following documentation of `Math.random()`:

random

```
public static double random()
```

Returns a `double` value with a positive sign, greater than or equal to 0.0 and less than 1.0. Returned values are chosen pseudorandomly with (approximately) uniform distribution from that range.

When this method is first called, it creates a single new pseudorandom-number generator, exactly as if by the expression

```
new java.util.Random
```

This new pseudorandom-number generator is used thereafter for all calls to this method and is used nowhere else.

This method is properly synchronized to allow correct use by more than one thread. However, if many threads need to generate pseudorandom numbers at a great rate, it may reduce contention for each thread to have its own pseudorandom-number generator.

Returns:

a pseudorandom `double` greater than or equal to 0.0 and less than 1.0.

See Also:

[Random.nextDouble\(\)](#)

Most Java programmers are well familiar with this method and with its return value range, and with the fact that values are chosen according to a uniform distribution. However, this documentation also contains two clauses that may surprise and have an impact on clients. One states a side effect of the call: the creation of a random generator on the first call. The second states that to improve performance when many threads need to generate numbers, one may want to have a separate generator in each thread. Both of these clauses are candidates for directives, although their importance is limited, and their informative nature means that ignoring them causes no serious ill effects.

There are cases, however, when such surprises could lead to serious effects. For example, consider the documentation for `String.replaceAll`:

Directive Tagging Study – Booklet, Uri Dekel, CMU

replaceAll

```
public String replaceAll(String regex, String replacement)
```

Replaces each substring of this string that matches the given [regular expression](#) with the given replacement.

An invocation of this method of the form `str.replaceAll(regex, repl)` yields exactly the same result as the expression

```
Pattern.compile(regex).matcher(str).replaceAll(repl)
```

Note that backslashes (\) and dollar signs (\$) in the replacement string may cause the results to be different than if it were being treated as a literal replacement string; see [Matcher.replaceAll](#). Use [Matcher.quoteReplacement\(java.lang.String\)](#) to suppress the special meaning of these characters, if desired.

Parameters:

`regex` - the regular expression to which this string is to be matched
`replacement` - the string to be substituted for each match

Returns:

The resulting `String`

Throws:

[PatternSyntaxException](#) - if the regular expression's syntax is invalid

Since:

1.4

See Also:

[Pattern](#)

According to this text, the replacement string is also treated as a regular expression, and must not contain dollar signs or backslashes. This may come as a surprise to someone who is used to `String.replace`. Some of those users may not even be aware that `replaceAll` works with regular expressions.

There are many cases where directives are a lot more obvious and carry more significant implications. For example, a method's documentation may instruct the user to call another method first, to not invoke it from a certain thread, or to be responsible for releasing a handle that it obtains from the platform.

Note that if something appears trivial or common it may not be a directive or may be a directive of marginal importance. For example, all JavaDocs in Java list all the parameters and often require that parameters not be null. Since this is fairly standard, and users are expected to check for this anyway, this is not a directive. However, if there is a restriction on the concepts of a parameter, for example, then this is a directive.

Procedures

In this study your role will be to identify directives in the printouts of the JavaDocs for several classes. You will receive a booklet consisting of some background material about the API, followed by the actual printout.

Please go systematically over the text for methods. When you find something that could possibly be a directive (even if you are not sure), in your opinion, use a highlighting pen to mark the entire text fragment that would correspond to a directive. This text could potentially be rewritten as a more concise and direct instruction, but you do not have to do so. If at doubt as to whether something is a directive or not, mark it anyway.

Note that the same method documentation may contain multiple directives, so please mark all of them. Markings should only be applied to the detailed Javadocs of `public` methods and constructors, and not to the documentation of the class, its fields, or nonprivate methods. In addition, you should only be concerned with directives aimed at users of an instance of a class (or subclasses) rather than directives aimed at users who will subclass the current class or override its methods.

Directive Tagging Study – Booklet, Uri Dekel, CMU

You are requested to highlight every clause which you consider to be a possible directive, even if you ultimately decide that it is not. Once you added the highlight, you are also asked to pick a rating between 0 and 7 in a regular pen next to the highlight. The rating should represent your opinion on this directive, and should factor in your confidence, how surprising or nontrivial it is, the importance of developers becoming aware of it or the consequences if they do not, and the situations in which it could be relevant. The “standard” confidence should be 4, with lower numbers indicating much lower rating. For example, “call X first” or “don’t invoke from UI thread” would likely be rated 7, while the directives we saw earlier for Random would likely be rated closer to 1. If you decide that something is, ultimately, not worthy of being a directive, mark it as 0.

In addition, you are asked to classify each highlighted directive with a rating of above 0, by annotating it with one (or at most, two) of the mnemonics for directive types that will be listed at the end of this document. Make sure you understand the distinction between types before you start working.

The markings you make will be compared to the markings created by several other subjects, to see how similar they are. The most important factor is whether a clause is marked or not, with the rating and typing supporting some information. Therefore, if you are not sure, mark the text as low rating rather than not mark at all.

In some cases, a very similar clause will repeat almost identically in several methods. Please mark all cases. We realize that the task as a whole is mundane and tiring, but please try to remain attentive. Feel free to take breaks at any point.

You may ask the experimenter questions, if necessary, about API concepts or terminology.

The first class you will look at will serve as a pilot and practice, and you will interact with the experimenter after you have finished tagging it.

Directive Types

Please keep this page open for reference during the study

GEN – General – Please use this if nothing else is appropriate

RES – Restriction - Forbids the use of the method from certain contexts (e.g., "don't invoke from the UI thread") or defines the entities allowed to make the invocation (e.g., "To be called only from debug infrastructure")

LIM - Limitation - Alerts the user to some (unexpected) limitation in how the method works. For example, "does not announce changes to listeners"

PROT – Protocol - Conveys some invocation sequence. For example "don't invoke this before you invoked X" or "remember to notify Y after calling this".

THRD – Threading - Conveys some issues relating to threading, such as requiring the use of a system thread or indicating that execution may block.

LOCK – Locking - Conveys specific locking requirements

PARA – Parameters - Conveys specific instructions about the parameter, that are far from being trivial (and thus covered just by the @param tag). For example, the parameter value should not include certain content.

RET – Return value - Conveys specific instructions about the return value that are far from being trivial (and thus covered just by the @param tag). For example, deallocation responsibilities.

PERF - Performance - Conveys to the client that there is some performance issue with using this method. For example, that it takes a lot of time

SIDE – Side effect - Alerts the user to some sideeffect associated with invoking this method

SEC – Security - Alerts the user to some security implications or requirements associated with invoking this method.

ALT – Alternative - Conveys to the users that they may want to use a different method. For example, "to cause a refresh, call X instead".

REC – Recommendation - conveys to the users that they may want to perform additional operations. For example, "you may want to validate the URL first", but that not doing so is not an error

Bibliography

- [1] How to write doc comments for the Javadoc tool. <http://java.sun.com/j2se/javadoc/writingdoccomments/>.
- [2] A. Aguiar and G. David. Wikiviki weaving heterogeneous software artifacts. In *WikiSym '05: Proceedings of the 2005 international symposium on Wikis*, pages 67–74, New York, NY, USA, 2005. ACM Press.
- [3] S. W. Ambler. *The Object Primer - Agile Model-Driven Development with UML 2.0*. Cambridge University Press, 2003.
- [4] F. Bachmann and P. Merson. Experience using the web-based tool wiki for architecture documentation. Technical Report CMU/SEI-2005-TN-041, CMU Software Engineering Institute, 2005.
- [5] A. D. Baddeley. *Human Memory: Theory and Practice*. Allyn and Bacon, 1997.
- [6] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–6, New York, NY, USA, 1989. ACM Press.
- [7] K. Bierhoff and J. Aldrich. Lightweight object specification with tpestates. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 217–226, New York, NY, USA, 2005. ACM.
- [8] K. Bierhoff and J. Aldrich. Modular tpestate checking of aliased objects. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 301–320, New York, NY, USA, 2007. ACM.
- [9] A. Black. Visible planning on paper and on screen: The impact of working medium on decision-making by novice graphic designer. *Behaviour and Information Technology*, 9(4):283–296, 1990.
- [10] J. Bloch. How to design a good api and why it matters. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 506–507, New York, NY, USA, 2006. ACM.
- [11] N. Boulila. Group support for distributed collaborative concurrent software modeling. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 422–425, Washington, DC, USA, 2004. IEEE Computer Society.
- [12] L. C. Briand, Y. Labiche, M. D. Penta, and H. D. Yan-Bondoc. An experimental investigation of formality in uml-based development. *IEEE Transactions on Software Engineering*, 31(10):833–849, 2005.

- [13] F. P. Brooks. *The Mythical Man-Month*. AW, anniversary edition edition, 1996.
- [14] Q. Chen, J. Grundy, and J. Hosking. An e-whiteboard application to support early design-stage sketching of uml diagrams. In *IEEE Conference on Human-Centric Computing (HCC'03)*, 2003.
- [15] L.-T. Cheng, C. R. de Souza, S. Hupfer, J. Patterson, and S. Ross. Building collaboration into IDEs. *Queue*, 1(9):40–50, 2004.
- [16] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 557–566, New York, NY, USA, 2007. ACM Press.
- [17] J. Cohen. A coefficient of agreement for nominal scales. *Psychological Bulletin*, 20:37–46, 1960.
- [18] O. Creighton, M. Ott, and B. Bruegge. Software cinema-video-based requirements engineering. In *RE '06: Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, pages 106–115, Washington, DC, USA, 2006. IEEE Computer Society.
- [19] *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, Chicago, Illinois, USA, November 2004. ACM Press.
- [20] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.
- [21] C. H. Damm and K. M. Hansen. Distributing knight. using type-based publish/subscribe for building distributed collaboration tools. In *NWPER'2002*, 2002.
- [22] C. H. Damm, K. M. Hansen, and M. Thomsen. Tool support for cooperative object-oriented design: gesture based modelling on an electronic whiteboard. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 518–525, New York, NY, USA, 2000. ACM Press.
- [23] C. H. Damm, K. M. Hansen, M. Thomsen, and M. Tyrsted. Supporting several levels of restriction in the uml. In *UML*, volume 1939 of *Lecture Notes in Computer Science*, pages 396–409. Springer, 2000.
- [24] C. R. B. de Souza, D. Redmiles, L.-T. Cheng, D. Millen, and J. Patterson. How a good software practice thwarts collaboration: the multiple roles of apis in software development. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 221–230, New York, NY, USA, 2004. ACM Press.
- [25] U. Dekel. Supporting distributed software design meetings: what can we learn from co-located meetings? *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.
- [26] U. Dekel. A framework for studying the use of wikis in knowledge work using client-side access data. In *WikiSym '07: Proceedings of the 2007 international symposium on Wikis*, pages 25–30, New York, NY, USA, 2007. ACM.
- [27] U. Dekel and Y. Gil. Revealing class structure with concept lattices. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 353, Washington, DC, USA, 2003. IEEE Computer Society.
- [28] U. Dekel and J. D. Herbsleb. Notation and representation in collaborative object-oriented design: an observational study. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 261–280, New York, NY, USA, 2007. ACM.

- [29] ACM DesignFest homepage. <http://designfest.acm.org>.
- [30] S. Elrod, R. Bruce, R. Gold, D. Goldberg, F. Halasz, W. Janssen, D. Lee, K. McCall, E. Pedersen, K. Pier, J. Tang, and B. Welch. Liveboard: a large interactive display supporting group meetings, presentations, and remote collaboration. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 599–607, New York, NY, USA, 1992. ACM Press.
- [31] G. Fairbanks, D. Garlan, and W. Scherlis. Design fragments make using frameworks easier. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 75–88, New York, NY, USA, 2006. ACM.
- [32] P. M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *J Exp Psychol*, 47(6):381–391, June 1954.
- [33] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 70–79, Washington, DC, USA, 2007. IEEE Computer Society.
- [34] B. Fluri, M. Würsch, E. Giger, and H. C. Gall. Analyzing the co-evolution of comments and source code. *Software Quality Control*, 17(4):367–394, 2009.
- [35] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. Hooking into object-oriented application frameworks. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 491–501, New York, NY, USA, 1997. ACM.
- [36] G. W. Furnas and B. B. Bederson. Space-scale diagrams: understanding multiscale interfaces. In *CHI '95: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 234–241, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.
- [37] J. Gil and S. Kent. Three dimensional software modelling. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 105–114, Washington, DC, USA, 1998. IEEE Computer Society.
- [38] J. Grundy and J. Hosking. Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 282–291, Washington, DC, USA, 2007. IEEE Computer Society.
- [39] J. D. Herbsleb. Metaphorical representation in collaborative software engineering. In *WACC '99: Proceedings of the international joint conference on Work activities coordination and collaboration*, pages 117–126, New York, NY, USA, 1999. ACM Press.
- [40] J. D. Herbsleb, H. A. Klein, G. Olson, H. Brunner, J. Olson, and J. Harding. Object-oriented analysis and design in software project teams. *Human-Computer Interaction*, 10(2):249–292, 1995.
- [41] E. Hill. Developing natural language-based program analyses and tools to expedite software maintenance. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 1015–1018, New York, NY, USA, 2008. ACM.
- [42] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of n-queries for software maintenance and reuse. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 232–242, Washington, DC, USA, 2009. IEEE Computer Society.
- [43] S. Hupfer, L.-T. Cheng, S. Ross, and J. Patterson. Introducing collaboration into an application development environment. In *CSCW'04* [19], pages 21–24.

- [44] C.-L. Ignat and M. C. Norrie. Grouping in collaborative graphical editors. In *CSCW'04* [19], pages 447–456.
- [45] W. Ju, A. Ionescu, L. Neeley, and T. Winograd. Where the wild things work: capturing shared physical design workspaces. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 533–541, New York, NY, USA, 2004. ACM Press.
- [46] M. Kersten. *Focusing Knowledge Work with Task Context*. PhD thesis, University of British Columbia, 2007.
- [47] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM Press.
- [48] S. R. Klemmer, M. W. Newman, R. Farrell, M. Bilezikjian, and J. A. Landay. The designers' outpost: a tangible interface for collaborative web site. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 1–10, New York, NY, USA, 2001. ACM Press.
- [49] D. Knuth. *Literate Programming*. Lecture Notes. Center for the Study of Language and Information, 1992.
- [50] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 344–353, Washington, DC, USA, 2007. IEEE Computer Society.
- [51] F. S. R. . S. J. Kraut, R. E. Visual information as a conversational resource in collaborative physical tasks. *Human-Computer Interaction*, (18):13–49, 2003.
- [52] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, March 1977.
- [53] C. F. J. Lange and M. R. V. Chaudron. Effects of defects in uml models: an experimental investigation. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 401–411, New York, NY, USA, 2006. ACM Press.
- [54] T. D. LaToza, D. Garlan, J. D. Herbsleb, and B. A. Myers. Program comprehension as fact finding. In *ESEC-FSE '07: Proceedings of the 6th joint meeting of the european software engineering conference and the 14th ACM SIGSOFT symposium on Foundations of software engineering*, pages 361–370, New York, NY, USA, 2007. ACM Press.
- [55] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM Press.
- [56] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector. Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1323–1332, New York, NY, USA, 2008. ACM.
- [57] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

- [58] J. Lin, M. W. Newman, J. I. Hong, and J. A. Landay. Denim: finding a tighter fit between tools and practice for web site design. In *CHI '00: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 510–517, New York, NY, USA, 2000. ACM Press.
- [59] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, New York, NY, USA, 1987. ACM.
- [60] A. MacLean, R. M. Young, V. M. E. Bellotti, and T. P. Moran. Questions, options, and criteria: elements of design space analysis. *Human Computer Interaction*, 6(3):201–250, 1991.
- [61] N. Mangano, A. Baker, and A. van der Hoek. Calico: a prototype sketching tool for modeling in early design. In *MiSE '08: Proceedings of the 2008 international workshop on Models in software engineering*, pages 63–68, New York, NY, USA, 2008. ACM.
- [62] D. P. Marin. What motivates programmers to comment? Master's thesis, EECS Department, University of California, Berkeley, Nov 2005.
- [63] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.
- [64] A. Mehra, J. Grundy, and J. Hosking. Supporting collaborative software design with a plug-in, web services-based architecture. In *Workshop on Directions in Software Engineering Environments (WoDiSEE) at ICSE '04*. IEEE Computer Society, 2004.
- [65] B. Meyer. *Eiffel : The Language*. Prentice Hall, 1991.
- [66] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2000.
- [67] A. Nugroho and M. R. Chaudron. A survey into the rigor of uml use and its perceived impact on quality and productivity. In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 90–99, New York, NY, USA, 2008. ACM.
- [68] Object Management Group. Uml 2.0 specification.
- [69] Y. Padioleau, L. Tan, and Y. Zhou. Listening to programmers taxonomies and characteristics of comments in operating system code. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 331–341, Washington, DC, USA, 2009. IEEE Computer Society.
- [70] P. Pirolli and S. K. Card. Information foraging. *Psychological Review*, 106:643–675, 1999.
- [71] B. Plimmer and M. Apperley. Interacting with sketched interface designs: an evaluation study. In *CHI '04: CHI '04 extended abstracts on Human factors in computing systems*, pages 1337–1340, New York, NY, USA, 2004. ACM Press.
- [72] B. Plimmer and J. Grundy. Beautifying sketching-based design tool content: issues and experiences. In *CRPIT '40: Proceedings of the Sixth Australasian conference on User interface*, pages 31–38, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.
- [73] J. Ramey, T. Boren, E. Cuddihy, J. Dumas, Z. Guan, M. J. van den Haak, and M. D. T. De Jong. Does think aloud work?: how do we know? In *CHI '06: CHI '06 extended abstracts on Human factors in computing systems*, pages 45–48, New York, NY, USA, 2006. ACM.

- [74] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantir: raising awareness among configuration management workspaces. In *Proceedings of the 25th international conference on Software engineering*, pages 444–454. IEEE Computer Society, 2003.
- [75] E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Commun. ACM*, 31(11):1259–1267, 1988.
- [76] J. C. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy pascal programs. *Human-Computer Interaction*, 2:163–207, 1985.
- [77] R. Stein and S. E. Brennan. Another person’s eye gaze as a cue in solving programming problems. In *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*, pages 9–15. ACM Press, 2004.
- [78] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby. Shared waypoints and social tagging to support collaboration in software development. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 195–198, New York, NY, USA, 2006. ACM.
- [79] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer. Todo or to bug: exploring how task annotations play a role in the work practices of software developers. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 251–260, New York, NY, USA, 2008. ACM.
- [80] J. Stylos and B. A. Myers. The implications of method placement on api learnability. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 105–112, New York, NY, USA, 2008. ACM.
- [81] J. Stylos, B. A. Myers, and Z. Yang. Jadeite: improving api documentation using usage information. In *CHI EA '09: Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, pages 4429–4434, New York, NY, USA, 2009. ACM.
- [82] Sun. Requirements for writing java api specifications.
- [83] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. */*comment: bugs or bad comments?*/*. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, New York, NY, USA, 2007. ACM.
- [84] L. Tan, D. Yuan, and Y. Zhou. Hotcomments: how to make program comments more useful? In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.
- [85] S. Tilley and S. Huang. A qualitative assessment of the efficacy of uml diagrams as a form of graphical documentation in aiding program understanding. In *SIGDOC '03: Proceedings of the 21st annual international conference on Documentation*, pages 184–191, New York, NY, USA, 2003. ACM Press.
- [86] W. Visser. Designing as construction of representations: A dynamic viewpoint in cognitive design research. *Human-Computer Interaction*, 21(1):103–152, 2006.
- [87] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The effect of modularization and comments on program comprehension. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 215–223, Piscataway, NJ, USA, 1981. IEEE Press.

- [88] J. Wu, T. Graham, and P. Smith. A study of collaboration in software design. In *2003 International Symposium on Empirical Software Engineering (ISESE '03)*. IEEE Computer Society, 2003.
- [89] J. Wu and T. C. N. Graham. The software design board: A tool supporting workstyle transitions in collaborative software design. In *Proceedings of Engineering for Human-Computer Interaction and Design, Specification and Verification of Interactive Systems, LNCS 2844*, pages 92–106. Springer Verlag, 2004.
- [90] A. T. T. Ying, J. L. Wright, and S. Abrams. Source code that talks: an exploration of eclipse task comments and their implication to repository mining. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.
- [91] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language api documentation. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, November 2009.