

Design Fragments

George Fairbanks

CMU-ISRI-07-108

19 April 2007

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

David Garlan, Co-Chair
William Scherlis, Co-Chair
Jonathan Aldrich
Ralph Johnson

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

Copyright © 2007 George Fairbanks

This research was sponsored by the US Army Research Office (ARO) under grant number DAAD19-01-1-0485 and DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to Carnegie Mellon University's CyLab, the National Science Foundation under grant CCR-0113810 and IIS-0534656, and a research contract from Lockheed Martin ATL.

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 the ARO, the U.S. government, Lockheed Martin ATL or any other entity.

Keywords: design fragments, frameworks, object-oriented frameworks, patterns, interfaces

Abstract

Frameworks are a valuable way to share designs and implementations on a large scale. Client programmers, however, have difficulty using frameworks. They find it difficult to understand non-local client-framework interactions, design solutions when they do not own the architectural skeleton, gain confidence that they have engaged with the framework correctly, represent their successful engagement with the framework in a way that can be shared with others, ensure their design intent is expressed in their source code, and connect with external files.

A design fragment is a specification of how a client program can use framework resources to accomplish a goal. From the framework, it identifies the minimal set of classes, interfaces, and methods that should be employed. For the client program, it specifies the client-framework interactions that must be implemented. The structure of the client program is specified as roles, where the roles can be filled by an actual client program's classes, fields, and methods. A design fragment exists separately from client programs, and can be bound to the client program via annotations in their source code. These annotations express design intent; specifically, that it is the intention of the client programs to interact with the framework as specified by the design fragment.

The thesis of this dissertation is: *We can provide pragmatic help for programmers to use frameworks by providing a form of specification, called a design fragment, to describe how a client program can correctly employ a framework and by providing tools to assure conformance between the client program and the design fragments.*

We built tools into an IDE to demonstrate how design fragments could alleviate the difficulties experienced by client programmers. We performed two case studies on commercial Java frameworks, using demo client programs from the framework authors, and client programs we found on the internet. The first case study, on the Applet framework, yielded a complete catalog of twelve design fragments based on our analysis of fifty-six Applets. The second case study, on the larger Eclipse framework, yielded a partial catalog of fourteen design fragments based on our analysis of more than fifty client programs.

This work provides three primary contributions to software engineering. First, it provides a new technique to help programmers use frameworks. Second, it provides a systematic way to increase code quality. Design fragments provide a means to communicate known-good designs to programmers, and, unlike simple copying of examples, a means of influencing the uses of that design so that revisions can be propagated. Third, it provides an empirically-based understanding of how clients use frameworks, which aids researchers in choosing research directions and aids framework authors in delivery of new frameworks.

Acknowledgments

Throughout this document I have used “we” to refer to the developers of the ideas in this thesis. I have done this because of the enormous debt that I owe to the faculty and students in our department. My advisors and committee, David Garlan, Bill Scherlis, Jonathan Aldrich, and Ralph Johnson, were essential in the transformation of an interesting idea into supported research. I could not have done it without you. I will not forget the formative influence of Mary Shaw on my understanding of software engineering fundamentals.

Bradley Schmerl has helped me in many areas, from aiding the core ideas to alpha testing the tool, and even reading drafts of this thesis. My father and Kenneth Creel, neither one having much of an interest in software engineering, helped greatly by reading drafts of this thesis.

My fellow students, though already fully burdened with their own work, have always been eager to discuss ideas and to help. The tough but fair tone at the weekly department seminars is a testament to their abilities and temperaments.

Any remaining faults in this thesis are mine alone.

I consider myself fortunate to have been in the company of so many brilliant minds. Thank you all for your help and friendship.

Contents

1	Introduction	1
1.1	Example	1
1.2	Difficulties Using Frameworks	4
1.3	Coping Strategies	5
1.4	Design Fragments	6
1.5	Research Questions	9
2	Object-Oriented Frameworks	11
2.1	Object-Oriented Frameworks	11
2.2	Framework Mechanisms	12
2.3	Why Frameworks Are Difficult for Clients	14
2.3.1	Understand Non-Local Client-Framework Interactions	14
2.3.2	Design Solutions Without Owning the Architecture	15
2.3.3	Gain Confidence of Compliance	15
2.3.4	Represent Solutions	15
2.3.5	Encode Design Intent	16
2.3.6	Connect with External Files	16
2.4	Differences Compared to Libraries	17
2.5	Current Best Practice	17
2.5.1	Sources of Information on Frameworks	18
2.5.2	Help in Overcoming Difficulties	21
2.6	Summary	21
3	Thesis and Hypotheses	23
3.1	Thesis	23
3.2	Hypothesis: Real Conditions	24
3.3	Hypothesis: Design Fragment Variety	24
3.4	Hypothesis: Assurance	25
3.5	Integration	26
4	Design Fragments	27
4.1	What a Design Fragment Is	27
4.2	Contents of a Design Fragment	28
4.2.1	Object-Oriented Language	29

4.2.2	Configuration File Language	30
4.2.3	Specification Language	31
4.3	Design Fragments and Client Programs	36
4.4	Expressiveness	38
4.5	Using Design Fragments	39
4.6	Design Fragments in the Software Lifecycle	40
4.7	Benefits	41
4.8	Summary	41
5	Tools	43
5.1	Understand Non-Local Client-Framework Interactions	45
5.2	Design Solutions without Owning the Architecture	48
5.3	Gain Confidence of Compliance	49
5.4	Represent Solutions	50
5.5	Encode Design Intent	51
5.6	Connect with External File	51
5.7	Use Cases	51
5.8	Summary	52
6	Case Study: Applet Framework	55
6.1	Applet Framework	55
6.2	Hypotheses	56
6.3	Process	57
6.3.1	Demo Applets	57
6.3.2	Collecting Internet Applets	57
6.3.3	Apply Catalog to Internet Applets	58
6.3.4	Collect Demographic Information	58
6.4	Results	58
6.4.1	Threaded Design Fragments	59
6.4.2	Event Handling Design Fragments	60
6.4.3	Parameterized Applet Design Fragment	61
6.4.4	Manual Applet Design Fragment	62
6.4.5	Demographics	62
6.5	Discussion	63
6.5.1	Threading Bug	65
6.5.2	Non-Conformance	67
6.5.3	Selection Bias	69
6.5.4	Authoring Consistency	69
6.5.5	Hypothesis: Small Catalog	70
6.5.6	Hypothesis: Examples Strategy	72
6.6	Summary	73

7 Case Study: Eclipse Framework	75
7.1 Eclipse Framework	75
7.2 Hypotheses	76
7.3 Process	76
7.3.1 Choose Client Program	76
7.3.2 Identify Client-Framework Interaction Types	77
7.3.3 Create Design Fragment Catalog	77
7.3.4 Identify Required and Optional Interactions	77
7.3.5 Examine Internet Use	78
7.4 Results	79
7.4.1 Client-Framework Interaction Types	79
7.4.2 Design Fragment Catalog	81
7.4.3 Coverage by Design Fragments	81
7.4.4 Internet Use of Design Fragments	82
7.5 Discussion	85
7.5.1 Selection Bias	85
7.5.2 Coverage	86
7.5.3 Hypothesis: Limited Design Fragment Variety	87
7.5.4 Hypothesis: Examples Strategy	88
7.6 Summary	89
8 Related Work	91
8.1 Organization of the Research	91
8.2 Specific Techniques	92
8.2.1 Role Modeling	93
8.2.2 Precise Design Patterns, Code Ties	93
8.2.3 Cookbooks and Recipes	93
8.2.4 Patterns	94
8.2.5 Languages	95
8.2.6 Aspects	95
8.2.7 General Programming Assistance	95
8.2.8 Pattern Mining	96
8.3 Comparison to Design Fragments	96
9 Conclusion	99
9.1 Validation	99
9.1.1 Hypothesis: Real Conditions	99
9.1.2 Hypothesis: Design Fragment Variety	100
9.1.3 Hypothesis: Assurance	102
9.1.4 Thesis	102
9.2 Pragmatism Concerns	104
9.2.1 Programmer Differences	104
9.2.2 Domain-specific Languages	105
9.2.3 Specifications Opportunities	106

9.2.4	High Expression Cost	107
9.2.5	Composability	110
9.2.6	Expressiveness Limitations	110
9.2.7	Long Tail of Design Fragment Catalog	111
9.2.8	Java Language	112
9.3	Discussion	112
9.3.1	Framework Encapsulation	113
9.3.2	Examples as Seed Crystals	113
9.3.3	Advice to Framework Authors	114
9.3.4	Framework Refactoring Opportunities	114
9.4	Contributions	115
9.5	Future work	116
9.6	Summary	118
A	Adding a New Specification	121
A.1	New Specification	121
A.2	Tool Extension	124
A.3	Summary	124
B	Design Fragment Language	127
B.1	Abstract Syntax	127
B.2	XML Schema Definition	128
Bibliography		133

List of Figures

1.1	Responding to selection changes in a tree view in the Eclipse framework	2
1.2	Client-framework interactions	3
1.3	Design fragment structure of Tree View Selection in Eclipse	8
1.4	Binding design fragment roles	9
2.1	Applet callbacks	13
2.2	JavaDoc for <code>ISelectionChangedListener</code>	18
2.3	JavaDoc for <code>IResourceChangeListener</code>	20
4.1	UML structure of Background Continuous V1 design fragment	29
4.2	Abstract syntax of the core design fragment language	30
4.3	Abstract syntax of the XML design fragment language	31
4.4	Design fragment with XML specifications	32
4.5	Abstract syntax of design fragment specifications	34
4.6	Instance declarations reveal architectural understanding	38
5.1	All tool views	44
5.2	Catalog View	45
5.3	Instances View	46
5.4	Context View	47
5.5	Slice View	48
5.6	Problems View	49
5.7	Errors appearing in editor	50
6.1	Applet callbacks	56
6.2	Background Continuous V1 and V2 structure	60
6.3	Mouse event handling structure	61
6.4	Parameter handling structure	62
6.5	Manual Applet structure	63
6.6	NervousText JDK 1.0 Applet	65
6.7	Applet callbacks - correct (left) and incorrect (right)	67
6.8	Simple AWT Applet	69
6.9	Conformance criteria for design fragments	70
6.10	Applet catalog growth	71
7.1	Screenshot of our client program	77

7.2	Design fragment catalog conformance	85
7.3	Instance declarations reveal architectural understanding	87
A.1	Current Parameterized Applet design fragment	122
A.2	Improved Parameterized Applet design fragment	122
A.3	Design fragment XSD for parameter checking	123
A.4	Declaration of presenter and checker in <code>plugin.xml</code>	124
A.5	Interfaces for Analyzer and Presenter	125

List of Tables

2.1	Helpfulness of information sources to overcome difficulties	21
4.1	Specifications, checked and unchecked	35
4.2	Expressiveness of design fragment language	39
6.1	Design fragment frequency	59
6.2	Internet Applet demographics	64
6.3	Non-conformance to design fragment	68
6.4	Frequency of thread field name	72
6.5	All Applets with origin and counts of design fragments	74
7.1	Internet client program search strings	79
7.2	Framework interaction counts	81
7.3	Excerpt of code coverage	83
7.4	Design fragment coverage of framework interactions	84
7.5	Internet client program search strings with lengths	86
8.1	Organization of the research	92
9.1	Helpfulness of information to personas	105
9.2	Cost and benefit of annotations	109

Chapter 1

Introduction

Engineers in most domains encode know-how in the form of often-reused solutions. Software engineers are no exception. For example, software libraries collect together reusable functions or objects that can be used as-is. With libraries, the client programmer’s code requests a service provided by the library, and then resumes once the library call has finished. The client code forms the architectural skeleton of the completed system and the library code is used to implement the smaller details, like math routines or socket communication. This master-servant relationship between the client program and library is easy to understand but prevents the reuse of the architectural skeleton. Unlike libraries, object-oriented frameworks enable reuse of the architectural skeleton.

Object-oriented frameworks, or in this document just “frameworks,” provide partially complete skeleton applications for a particular domain, such as windowing systems or application servers. Examples of frameworks include Enterprise Java Beans (EJB) [47], Microsoft .NET [15], and Java applets [63]. While both libraries and frameworks are ways to reuse software, the defining difference between them is that a program interacting with a library requires services from that library, but a program interacting with a framework both requires services from, and provides services to, that framework. The client program must respond to framework requests and must satisfy any constraints that the framework imposes; for example the EJB framework requires that client programs not start their own threads, and windowing frameworks like .NET require that client programs complete some requests within milliseconds. The price that client programmers pay for reusing the architectural skeleton provided by a framework is that they must play by the framework’s rules.

1.1 Example

To illustrate the nature of the client-framework interactions, let us consider an example from the Eclipse framework. In this example, a programmer has written a client program that places a tree view on the screen, much like the File Explorer view in Microsoft Windows. The excerpt of Java source code in Figure 1.1 shows the parts of the client program that are needed to listen to and respond to changes in the user’s selection; the rest of the client program has been elided. The bolded sections of the source code are references to framework classes, interfaces, and methods.

```

...
public class SampleView extends ViewPart
    implements ISelectionChangedListener {

    private TreeViewer viewer;

    public void createPartControl(Composite parent) {
        viewer.addSelectionChangedListener(this);
        ...
    }

    public void dispose() {
        viewer.removeSelectionChangedListener(this);
        ...
    }

    public void selectionChanged(SelectionChangedEvent event) {
        ISelection s = event.getSelection();
        IStructuredSelection ss = (IStructuredSelection) s;
        ...
    }
}
...
}

```

Figure 1.1: Responding to selection changes in a tree view in the Eclipse framework

The programmer's intent with this code is to be notified when the user clicks on the tree view and changes the selection. This intent is realized via ten *client-framework interactions* between this client program and the framework, as shown in Figure 1.2. The framework provides a superclass, called **ViewPart**, to help with presenting a **TreeViewer**. The **ViewPart** class defines callback methods that correspond to different times in its lifecycle; two of these lifecycle callback methods are the **createPartControl** and **dispose** methods. In **createPartControl**, our client program requests that it receive events corresponding to user selection changes. It does this by invoking the **addSelectionChangedListener** method on its **TreeViewer** (the code where this **TreeViewer** was created is not shown). Symmetrically, in the **dispose** method it invokes **removeSelectionChangedListener** on the **TreeViewer**. In order for event registration and deregistration to work, this class must implement the **ISelectionChangedListener** interface and the one method on that interface, **getSelection**. The framework will invoke **getSelection** on the client when the user's selection changes, passing in an object from the framework, a **SelectionChangedEvent**. At that point the client can find out what was selected by invoking **getSelection**, but this yields an **ISelection**, not the subclass **IStructuredSelection**. However, since **TreeViewers** always yield **IStructuredSelections**, it is safe to downcast the **ISelection** to an **IStructuredSelection**.

Programmers who are familiar with design patterns [23] will see that the *Template* pattern is used here. The **ViewPart** superclass defines an abstract algorithm and lets its subclass fill in the details – in this case the **createPartControl** and **destroy** methods are intended to

1. Subclasses from (framework class) `ViewPart`
2. Implements (framework interface) `ISelectionChangedListener`
3. Holds field of type (framework class) `TreeViewer`
4. Overrides callback method `createPartControl` from `ViewPart`
5. Invokes method `addSelectionChangedListener` on the `TreeViewer` within the `createPartControl` method
6. Overrides callback method `dispose` from `ViewPart`
7. Invokes method `removeSelectionChangedListener` on the `TreeViewer` within the `dispose` method
8. Implements method `selectionChanged` (from framework interface `ISelectionChangedListener`)
9. Invokes method `getSelection` on (framework class) parameter of type `SelectionChangedEvent` within the `selectionChanged` method
10. Downcasts `ISelection` to `IStructuredSelection` (both framework interfaces) within the `selectionChanged` method

Figure 1.2: Client-framework interactions

contain those details. Additionally, the register/deregister method calls, plus the interface, is a use of the *Observer* pattern.

Two parts of this solution would not be obvious simply by knowing the standard design patterns. First, how the two patterns can be combined: that these particular lifecycle callback methods are the appropriate ones to do registration/deregistration. Since the framework changes state, a method call that will succeed during one callback may fail during another. In our example, these two lifecycle callback methods are appropriate places to coordinate with the event registration/deregistration, but the class constructor is not. The second non-obvious part is that the `ISelection` can be downcast to an `IStructuredSelection`. This is not generally true; it is only true because we are listening to a `TreeViewer` and not some other kind of viewer.

The solution shown in this example does not have tight locality, since the client-framework interactions are scattered across the program. Not only does the client interact with three framework interfaces and four framework classes, its interactions are spread across the class definition and multiple methods in the class.

Although this code describes a set of client-framework interactions, which can be viewed as interactions between two components, it is implemented with standard object-oriented mechanisms like subclassing and method overriding. Historically, this was a benefit because it allowed frameworks to be developed from standard object-oriented languages, but today it is a hindrance in that it makes these architecturally significant interactions blend in with other run-of-the-mill method calls.

Most languages today have a first class representation for the implementation-hiding interface to libraries, such as a Java *interface* or a C++ *header file*, but no similar representation exists

to declare the currently non-local client-framework interaction or encapsulate the framework implementation.

1.2 Difficulties Using Frameworks

A glance at the above example reveals that client-framework interactions can be complex. As a result, programmers find it difficult to understand non-local client-framework interactions, design solutions when they do not own the architectural skeleton, gain confidence that they have engaged with the framework correctly, represent their successful engagement with the framework in a way that can be shared with others, ensure their design intent is expressed in their source code, and connect with external files. Here are more details on those difficulties, with a fuller discussion appearing in Section 2.3:

Understand non-local client-framework interactions. Client programmers have a conceptually simple goal, like in the example above, but mechanically that may involve subclassing, implementing interfaces, overriding callback methods, and registering/deregistering for events – and these interactions are not co-located in the source code. This complexity yields difficulty both when designing solutions and when comprehending existing code. Programmers have difficulty discovering, learning, and implementing such complex interactions. Programmers evolving an already written program have difficulty piecing together the non-local parts that contribute to a single goal. A single statement in a client program, such as a call to the framework, may be present to enable more than one goal of the client program, which further complicates program evolution.

Design solutions without owning the architecture. A programmer designing a solution must do so within the solution space defined by the intersection of his problem domain and implementation constraints. A programmer will have little difficulty finding an acceptable solution when presented with a requirements document and a Java compiler. However, when the programmer chooses to use a framework then the solution space may be quite constrained, making it difficult to find one of the handful of acceptable solutions. The programmer may derive value from the way the framework constrains his actions, but he has difficulty inventing solutions that conform to those constraints.

Gain confidence of compliance. Programmers can test their code as always. However, since their code acts as a client of the framework there is an additional worry about poorly-understood contracts. For example, will the framework ever pass null as a parameter, what is the sequence of callbacks, or can this service method safely be requested during this callback?

Represent solutions. Despite the required complex client-framework interactions, natural language tutorials and example code are still the most common means of expressing known-good solutions. Consequently, it is difficult and inefficient to share understanding between programmers. Learning does take place but it is massively repeated as each programmer has to learn or infer what is already known by many others.

Encode design intent. Once a programmer codifies his design in a client program it is tedious for another programmer to discern the original intent, and often impossible for tools. The fact that the mechanics are non-local often prohibits the use of “intention revealing” method names [5] that could otherwise informally capture the intent of some opaque source code.

Connect with external files. Client programs for the early frameworks used only the object-oriented mechanics available in the programming language, but modern frameworks may also require the client to write an external file. This file is declarative and is often XML. In the Struts [32] framework, for example, processing nodes are written in Java but the connections between them are declared in an external XML file.

1.3 Coping Strategies

Despite these difficulties, client programmers are rational in their choice to use frameworks. Modern frameworks can be millions of lines of code [11], and the cost of recreating them is sufficiently high that client programmers endure the difficulties. Two essential strategies, what Polya would call heuristics [50], for overcoming the difficulties have emerged: devising a solution from scratch via a full understanding of the framework, and recreating an example solution. As we will discuss in Section 9.2.1, client programmers use both strategies in varying amounts based on the programmer’s temperament. We refer to these strategies using the shorthand names *first principles* and *examples*.

By *first principles* we mean a strategy where the client programmer studies the workings of the framework, creates a mental model of how the framework, then devises a client program that accomplishes his goal. The creation of a mental model necessarily involves surveying the available framework classes, interfaces, and methods; coming to an understanding of the semantics of framework methods, either through provided documentation or other means; understanding the sequence of framework callbacks; understanding which framework methods can be called by the client during which callbacks, and understanding the design patterns used by the framework. The framework may have additional constraints on client programs, including restrictions on concurrency, timing, or transactions. With small frameworks, it is possible to read much of the framework source code to learn this information, but with large frameworks, the programmer must rely primarily upon API documentation.

Predicates can express framework constraints and required post-conditions. Consequently, an advantage of this strategy is that tools can check specifications, providing client programmers with conformance assurance. There is potential for the set of constraints to be completely specified, but in practice they are usually not fully understood or documented. For example, visual controls in Microsoft’s ASP.NET have eleven lifecycle callbacks [44], yet their documentation is scattered and incomplete [52]. Consequently, client programmers may master all the published knowledge, yet still have difficulty designing solutions. The client programmer’s initial investment to learn the framework principles can be amortized if he uses the framework often, and mastery of general design patterns will help him to learn the next framework.

An inherent limitation of this strategy is that it does not help a client programmer to find a solution that satisfies both his requirements and the framework constraints. Knowing that an electrical device must plug into a framework that supplies 115 volts and 10 amperes is important, but does not guide an engineer to design a television or an air conditioner. Another limitation is that this strategy does not differentiate between typical and atypical use of the framework, such as which lifecycle callbacks are typically used to register for events.

The *examples* strategy seeks to avoid the upfront costs and undocumented constraints inher-

ent in the *first principles* strategy. The client programmer identifies an existing solution that matches his current goal. Existing solutions are represented in a variety of forms, including how-to articles and recipes, but most common are other client programs. Most framework authors deliver a small number of example client programs to show potential clients how to use parts of the framework. These example client programs can be considered an informal map of the framework, where the client-framework interactions they use are known territory, and the possible interactions they do not exercise are unknown territory. Programmers explore this unknown territory at their own risk because the known territory is better debugged. Client programmers who use the *examples* strategy must identify an appropriate example, identify the set of relevant client-framework interactions needed, and reproduce these interactions in their own code. This is the strategy that is recommended by leading framework authors [22]. Examples minimize the amount that client programmers must learn about the framework before successfully making progress.

There are downsides with this strategy too. Client programmers may have difficulty identifying examples that match their goals. Once they have identified an example, extraction of the example from its context may fail because of under-inclusion of necessary code, or may become bloated because of over-inclusion of irrelevant code. Because it is not based on a principled understanding of framework constraints, composing two working examples may yield a new program that violates a framework constraint.

Examples are usually embedded in entire programs, or are written up in natural language articles, making it difficult or impossible for tools to check conformance. Also, since an example shows just one way of accomplishing a goal, the client programmer may have difficulty building a general understanding of the framework. Perhaps most importantly in the long term, any strategy involving code copying creates a maintenance problem, since improvements or bug fixes in the original cannot easily be propagated into the copies.

The choice of representing knowledge in a first principles strategy or an examples strategy appears in many contexts. In the documentation of requirements, it is seen in functional requirements versus use cases, which are examples of first principles and examples, respectively. In the field of artificial intelligence, knowledge-based systems can choose between model-based representations [67] and case-based representations [56].

An ideal solution would incorporate the advantages of both the first principles strategy and the examples strategy while avoiding most of their disadvantages. Such a strategy would help programmers quickly find solutions to problems, as in the examples strategy, but enable tool assistance and aid learning about the framework, as in the first principles strategy.

1.4 Design Fragments

In this thesis we propose to help client programmers by providing *design fragments*, which express known-good ways to use a framework. A design fragment is a specification of how a client program can use framework resources to accomplish a goal. It identifies the minimal set of framework classes, interfaces, and methods that should be used, and specifies the client-framework interactions that must be implemented by the client program. Design fragments exist separately from client programs, and are bound to the source code via annotations. A client pro-

gram bound to a design fragment expresses the programmer's design intent. That is, bindings in the source code express the intent of the client program to interact with the framework as specified by the design fragment.

Since a design fragment describes a single way to use a framework, it has the advantages of the examples strategy. Additionally, because design intent is expressed, analysis tools can provide client programmers with conformance assurance, as in the first principles strategy. A design fragment helps client programmers to learn the existing framework API by providing references to the parts needed to accomplish a goal. These references allow the client programmer to begin learning the generality of what is offered by the framework. As such, design fragments blend the benefits of both the examples strategy and the first principles strategy.

A design fragment is composed of the following four parts:

1. The name of the design fragment.
2. The goal of a client programmer that is accomplished by this design fragment.
3. A description of what the programmer must build to accomplish the goal of this design fragment, including the classes, methods, and fields that must be present. This description also includes the behavior of these methods.
4. A description of the relevant parts of the framework that interact with the programmer's code, including the callback methods that will be invoked, the service methods that are provided, and other framework classes that are used.

A design fragment can express the client-framework interactions from the example shown earlier, the Eclipse selection listener. Figure 1.3 shows a UML static structure diagram of the design fragment, with the classes, interfaces, and methods from the framework appearing above the bar. Below the bar is a class (`RoleView`) with a field (`roleTree`) and three methods (`createPartControl`, `destroy`, and `selectionChanged`). While the names of the framework resources are predetermined, the class named `RoleView` will be bound to the client program's class that engages in this design fragment.

In addition to specifying the client program's structure, a design fragment also describes its required behavior. The design fragment specifies the required method calls and downcasting of objects described in client-framework interactions 5, 7, 9, and 10 listed earlier in Figure 1.2. Analysis tools can check that the source code implements these interactions correctly, subject to the sophistication of the analysis. Our simple analysis tools can check all but the last interaction, the downcast from `ISelection` to `IStructuredSelection`. Specifications of the design fragment structure and behavior are tool-readable, except for any specifications written in natural language.

Design fragments are collected together into a catalog, and client programmers can search the catalog to find a design fragment that matches their needs. When there is more than one way to accomplish a goal, the catalog will contain design fragments with variants.

A client programmer must express his intent to comply with a design fragment by adding Java 5 annotations to the source code. First, he declares that the source code implements a particular design fragment, and then binds any roles in the design fragment to the corresponding parts of the source code. In our selection change listener example, he would add an annotation like the following to declare that this code implements the `SelectionChangedListener` design

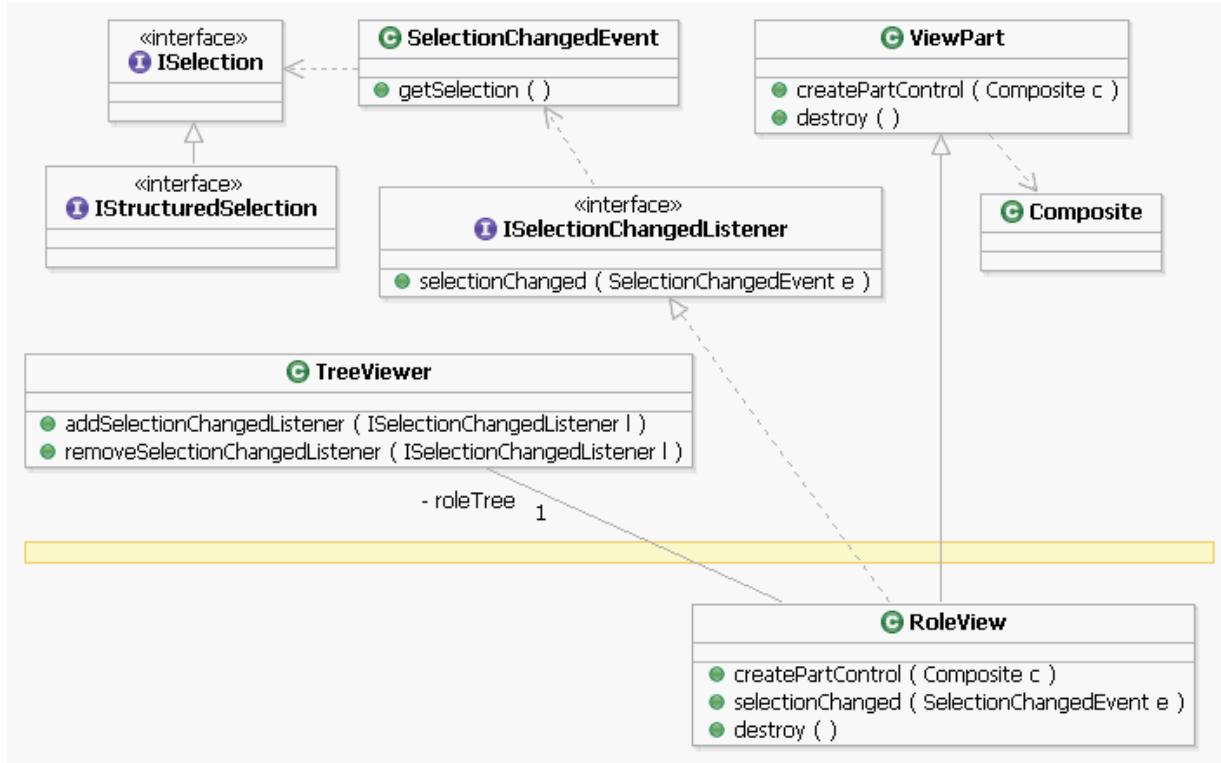


Figure 1.3: Design fragment structure of Tree View Selection in Eclipse

fragment, giving this instance of the design fragment the name `scl1`:

```
@df.instance(df="SelectionChangedListener", inst="scl1")
```

This design fragment has five roles to be bound: the role for the class `RoleView`, the role for the field `roleTree`, and the roles for the three methods. The `RoleView` class binding is shown in Figure 1.4, with the binding annotation in bold. With the current tooling, these binding annotations must be written by the programmer, but we see no impediments to wizard-style tools that guide the programmer and write the annotations automatically. These bindings are enduring design intent that can be used by other programmers, and by tools, to understand the source code. Using our design fragment tools, a client programmer can browse a catalog, list the design fragments, and navigate directly to the source code where they are used.

Once a programmer has bound a design fragment to the source code, analysis tools can ensure conformance with the structural and behavior constraints, both now and as the source code evolves. For example, our analysis tools will warn the programmer if he fails to deregister for selection changed events in the `destroy` method. The expressed design intent in these bindings unlocks future benefits. First, the design fragment specification will be elaborated with additional checkable behavior constraints as they are discovered or documented, or as the design fragment is debugged. And second, the sophistication of analysis tools will increase, allowing previously expressed, but uncheckable, specifications to be checked, such as the downcasting in our Eclipse selection listener example. This future, where declaration of design intent enables code quality increases, is in marked contrast to the maintenance problems of current cut-and-

```

...
@df.bindings({
    @df.binding(inst="sc11", role="RoleView")
})
public class SampleView extends ViewPart
    implements ISelectionChangedListener {
    ...
}
...
}

```

Figure 1.4: Binding design fragment roles

paste practices.

Design fragments and the associated tools directly address the difficulties raised earlier. Design fragments concisely explain the non-local client-framework mechanics. Client programmers can use design fragments as known-good solutions rather than inventing new solutions. Client programmers are provided with assurances that their code conforms to the design fragment and therefore engages with the framework correctly. Design fragments are specifications that express known-good solutions that can be shared with other programmers. And through the source code annotations, client programmers encode their design intent, enabling other programmers to better comprehend the program.

1.5 Research Questions

The *first principles* strategy has been investigated before. Role modeling has been applied to client-framework interactions [55] and seeks to explain the patterns employed by the framework authors. Earlier work such as Contracts [31] describes the constraints the framework places on all client programs and was later extended in the Framework Constraint Language [33].

The *examples* strategy has also been investigated before. Design fragment specifications are similar to JavaFrames [28], which are tool-readable patterns of client-framework interaction. JavaFrames in turn are descended from textual recipes [21, 41], which describe a known-good way to use a framework. We have extended JavaFrames’ expressiveness by adding references to the framework elements programmers must interact with, and bridging object-oriented languages and declarative configuration files.

Despite the research into these strategies, many open questions remain. How do client programmers interact with frameworks – are they inventing new patterns of interaction or are they following pre-existing patterns? Is it practical to collect examples of client-framework interactions into a catalog, or are there too many varieties to count? Do design fragments explain most of the observed client-framework interactions, or are most of the client-framework interactions in a client program unprecedented? What kinds of tools should be provided to overcome the problems with using frameworks? What are the structural and behavioral constraints that design fragments should express, and are there opportunities for analysis tools to assure their conformance?

In short, what has not been demonstrated is the pragmatism of the *examples* strategy and the ability of assurance tools to help. This dissertation investigates the following thesis:

We can provide pragmatic help for programmers to use frameworks by providing a form of specification, called a design fragment, to describe how a client program can correctly employ a framework and by providing tools to assure conformance between the client program and the design fragments.

Our evidence includes case studies performed on real frameworks, real languages, and real client programs. We demonstrate that the *example* strategy is pragmatic because there is limited variety in how these real programs interact with real frameworks. That is, two client programs that use the same framework are likely to interact with that framework in the same way. Collecting examples of uses of a large framework is not onerous and the resulting catalog of design fragments is small. We also show that it is possible for analysis tools to assure conformance between the source code and the constraints in the design fragment, further helping the client programmer.

This work provides three primary contributions to software engineering. First, it provides a new technique to help programmers use frameworks. Design fragments and the tools directly address the problems we have identified with frameworks. Design fragments improve on previous abstractions by describing relevant resources in the framework, connecting object-oriented code with external declarative configuration files, and by enabling new constraints and tools to be added to our predefined set.

Second, it provides a systematic way to increase code quality. Design fragments provide a means to communicate known-good designs to programmers, and, unlike simple copying of examples, a means of influencing the uses of that design so that revisions can be propagated. Bindings between design fragments and client programs express the programmer's design intent, and this enduring expression of intent enables conformance assurance from both existing, and future, analysis tools.

Third, it provides an empirically-based understanding of how clients use frameworks, which aids researchers in choosing research directions and aids framework authors in delivery of new frameworks. Specifically, the observed variety of client-framework interactions is very low, despite the opportunity for high variety.

This dissertation opens by providing background on object-oriented frameworks, then proceeds to articulate the specific research hypotheses we have investigated. Details on the design fragments specification and tooling are presented next. Two case studies that apply design fragments to the Applet and Eclipse frameworks are then presented, followed by a survey of related work and our conclusions.

Chapter 2

Object-Oriented Frameworks

This chapter provides an overview of object-oriented frameworks and differentiates them from traditional libraries. The essential characteristic of frameworks is the inversion of traditional control – the framework decides when and where to call the client code instead of the reverse – but this characteristic is also the source of increased complexity. Client programmers must discover how to accomplish their goals within the constraints of the framework, ensure that their programs use the framework correctly, and understand existing programs that they are evolving. Current best practices, including copy-and-paste programming and reading API documentation, are incomplete solutions to these challenges.

2.1 Object-Oriented Frameworks

There are two often-used definitions [38] for *object-oriented frameworks*. The first definition describes the framework’s purpose: “A framework is the skeleton of an application that can be customized by an application developer.” The second definition describes the structure of a framework: “A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact.”

An object-oriented framework is characterized by an inversion in the normal control relationship between the programmer and the library [55]. In the traditional usage of a library, the programmer decides when to invoke library functions and which ones to invoke. In a framework, the programmer is told by the framework when and where his code will be called.

Programmers are willing to sacrifice this control in order to receive other benefits. Examples of benefits include automatic management of transactions and concurrency (Enterprise Java Beans (EJB) [47]), operation within a windowing operating system (Microsoft .NET [15]), and interoperability with other tools in a software engineering environment (Eclipse framework [12]). Typically, the size of the framework code will dwarf the size of applications written within it. The Eclipse framework, for example, is almost two million lines of Java code [11] while an application written to work inside of it may be as small as a few dozen lines.

Using a framework also entails a reduction in risk for the client programmer. The framework author has extensive knowledge of the domain that is represented in the framework’s code. The framework’s code has also been used many times and bugs have been discovered and eliminated.

The framework itself may evolve over time to incorporate improved understanding of the domain and to eliminate more bugs. Had he not chosen to use a framework, the client programmer would have to do all these himself, with a corresponding increase in risk that he does them incorrectly.

The next section describes object-oriented and framework mechanisms. Terminology that we will use in this dissertation has been italicized. We refer back to the tree view client program from the introduction to provide concrete examples of the concepts we describe.

2.2 Framework Mechanisms

Object-oriented frameworks are built on top of the standard mechanisms in object-oriented languages. Java is an object-oriented language and defines *classes* of *objects* that can contain *fields* (data) and *methods* (behavior). Classes can be related in a hierarchy; in Java, a *subclass* is said to *extend* a *superclass*. *Interfaces* are collections of methods without implementations. A class can *implement an interface* by providing method implementations for all of the methods defined in that interface. Classes, fields, methods, and interfaces can be declared to be *public*, *private*, or *protected*, which manipulates which objects can refer to them.

Subclasses *inherit* all of the fields and methods from their superclass. A subclass at the bottom of a hierarchy of classes will inherit all of the fields and methods up to the top of the hierarchy. A subclass can *override* any method defined in its superclass by providing a new definition for that method. It is not unusual for a class to have the majority of its fields and methods provided by its superclasses.

Classes may declare methods that are *abstract*, and not provide an implementation for them. Such classes are themselves abstract, which means that it is impossible to create an instance of them directly. A subclass can override the abstract method and it would be possible to create an instance of that subclass.

The *template pattern* uses these mechanics to define an abstract algorithm in the superclass that is refined in a subclass. Here is a simple example: the superclass defines an algorithm that calls methods A, B, and C. These three methods are defined in the superclass, but method B is defined as abstract and referred to as a *template method*. A subclass would complete the pattern by extending the superclass and defining a non-abstract method B. This pattern is valuable since it allows a generic algorithm to be specified once, in the superclass, and refined many times through the creation of subclasses.

Frameworks build on these basic object-oriented mechanisms. The *framework author* provides a set of classes and interfaces that cannot be changed by the *client programmer*. Framework classes provide *service methods* that may be called by the *client program*; other methods are private implementation details. Referring to the introductory example in Figure 1.1, examples of service methods are `addSelectionChangedListener`, `removeSelectionChangedListener`, and `getSelection`.

Altogether, the framework classes define the architectural skeleton to which the client program must conform [39]. Client programs of the framework define their own classes that may extend classes from the framework and may implement interfaces from the framework. Generally, when the client program is started, the framework code starts running first and at selected times it will run the methods from the client program.

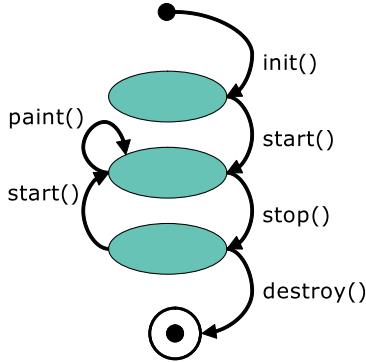


Figure 2.1: Applet callbacks

These “selected times” are referred to as *callbacks* from the framework. A callback is a transfer of control from the framework to the client program and is accomplished by the framework invoking a method defined in the client program, usually overridden template methods. From the client program’s perspective, callbacks can happen in response to spontaneous events in the system, such as when the user clicks a button or a web server receives a request. *Lifecycle callbacks* communicate the state of a framework class as it progresses through its lifecycle. For example, the framework may callback on the client program to let it know that an object is being created, activated, deactivated, and destroyed. Figure 2.1 shows the sequence of callbacks for the Applet framework. Referring to the introductory example from Figure 1.1, examples of lifecycle callbacks are `createPartControl` and `dispose`. The `selectionChanged` method is a callback, but since it is generated spontaneously, it is not a lifecycle callback.

The source code written by the client programmer does not run except during callbacks, so callbacks are the only opportunity for the client program to influence the behavior of the framework. The nature and frequency of the callbacks constrains the kind of influence a client program can have, because the client programmer cannot change the framework classes and consequently cannot change the callbacks. During callbacks, client programs can implement their own logic as well as calling the service methods on the framework. However, not all framework service methods are can legally be called from all callbacks. Callback methods often pass objects from the framework as parameters to the client program. The framework may expect the client program to hold on to these objects or it may expect them to be released by the end of the callback method.

A client program may interact with a framework in ways that are allowed by object-oriented programming languages. These *client-framework interactions* include: subclassing, implementing interfaces, overriding superclass methods, calling framework service methods, creating instances of framework classes, and holding on to framework objects. In order to accomplish a single *goal*, client programmers will employ multiple client-framework interactions in their program since there is usually not a one-to-one mapping between them. The client-framework interactions from the introductory example were listed in Figure 1.2.

Partly as a result of the low-level nature of these client-framework interactions, the full set of services provided by a framework tends to be large and detailed. This is in contrast to Service Oriented Architectures [13] whose set of services tend to be smaller and more aggregated. The

designer of a framework, however, has control over the level of detail and will choose it deliberately. A more detailed and rich set of services will be harder for clients to learn but will enable the creation of a greater range of client programs.

2.3 Why Frameworks Are Difficult for Clients

Despite being built from the same materials as other object-oriented programs, frameworks provide client programmers with new difficulties that arise from the nature of the client-framework interactions.

2.3.1 Understand Non-Local Client-Framework Interactions

When examining a client program, client programmers are faced with the difficulty of piecing together non-local client-framework interactions to understand the program. To accomplish a single, coherent task, the program must usually contain multiple client-framework interactions, and these interactions cannot be co-located. For example, a class declares its superclass and the interfaces it implements at the beginning of the class definition, defines methods for multiple callbacks throughout the class body, and calls framework service methods from within the callbacks. These interactions may be defined across multiple classes in the client program.

The difficulty is multiplied when a client program accomplishes multiple goals, as most do. The programmer reading the source code must decide if a particular interaction, say a call to a service method, exists to accomplish one goal or many. Our case studies reveal that 15% of client-framework interactions exist to accomplish multiple goals (Section 7.4).

Another compounding factor is that client programmers have difficulty identifying the parts of the framework that can be changed [38]. Since frameworks are built using the same raw materials as any other object-oriented program, it is difficult to identify which parts of the framework are intended to be hidden machinery and which parts are callback methods or service methods.

The essential nature of a client framework program makes it more difficult to comprehend than an equivalent program that uses libraries. Since frameworks typically provide the skeletal structure and control flow of an application, browsing the client program's source code provides only disconnected clues to the application's overall structure and behavior. This is in contrast to an application similarly written with libraries, where the programmer's code reveals the structure and control flow. Comprehension of the whole is easier when viewing the top of the tree rather than the leaves. When using a library, the programmer writes the most abstract parts of the program, and the library handles the details. Understanding the big picture is easy because these most abstract parts are usually localized within the client program. Conversely, when using a framework, the programmer provides the details to complete the framework's skeleton. Because of the framework mechanics, these details will be scattered across many classes and methods in his client program. When reading a client program, understanding the big picture is much harder because of this non-locality.

2.3.2 Design Solutions Without Owning the Architecture

Since the framework owns the architectural skeleton of the program, callbacks are often the only opportunity for client programmers to influence how the client program runs. When they design a framework, framework authors consider a set of ways that they expect client programs to use the framework. Some frameworks have many fine-grained callbacks, others are coarser, depending on what the framework author expects will be needed. Often times a client programmer wants to do something that was not anticipated by the framework author, and is therefore difficult to accomplish with the provided callbacks.

When such mismatch occurs between the client programmer's intended design and the available callbacks, it is the design, not the framework, that must be changed. A client programmer with incomplete knowledge of the framework does not yet know if what he wants to do is simply impossible, if he needs to continue searching for an appropriate callback, or if he needs to creatively design using the known callbacks. The client programmer can spend a considerable amount time deciding which of these three alternatives is correct.

An example of something that is simply impossible is found in early versions of the Enterprise Java Beans framework, which lacked a means to schedule a Bean to execute at a particular time, since Beans could only respond to external stimuli. If a client program needed to generate a report at 2 A.M. every day, it was impossible to do this completely within the EJB framework. A common workaround was to create a program that would send a message to the EJB program at 2 A.M., and at that point it could respond to the message and generate the report.

2.3.3 Gain Confidence of Compliance

With perfect knowledge of how the framework works, a programmer may still have a tedious time verifying that the client program is correct. The client program may respond to many callbacks from the framework and this state space may be large. For example, a program may receive lifecycle callbacks like `init` and `destroy`, but also register for events like mouse clicks. The programmer may wonder about the interleaving of these events, such as “Will I ever receive a mouse click event before I have completed `init`? ” Or wonder, “Is it always legal for me to call this service method from this callback? ”

Unfortunately, since framework documentation is not always complete and not all interactions have been foreseen, this tedious problem becomes even more difficult. And as a program is evolved by various programmers, their confidence that all framework constraints have been satisfied can easily erode.

2.3.4 Represent Solutions

It is difficult to find a way to make the framework do what you want, so once you have discovered it you would like to represent that understanding and share it with other programmers. It is possible to ask other programmers to look at your source code, but they may make mistakes inferring the solution because the client-framework interactions are non-local. Even when the example is copied correctly, the canonical solution may change, but since there is no path to follow from the original to the copies, it is difficult to propagate the new solution.

Writing a tutorial or a book is an effective way to describe your solution, but not everyone has a book publisher. These solutions are documented in natural language, so they will not be easily readable by tools. In contrast, important parts of API documentation are readable by tools, which enables programmers using IDEs to click on a class or method and navigate, without fail, to its documentation. Providing programmers with immediate, contextual assistance within IDEs is helpful because it minimizes their disruption when presented with a problem. Nor does it require them to remember what they read in a book some time in the past.

Because programmers lack an effective way to represent solutions, client programmers continue to repeatedly re-learn and re-infer what is already known by others.

2.3.5 Encode Design Intent

Client programmers still have access to their standard informal techniques for encoding design intent, including “intention revealing” method and field names. The necessary non-locality of client-framework interactions, however, reduces the effectiveness of these techniques. It is not possible to change the name of the framework callbacks to something that better reveals its intention within the client program. It may also not be possible to refactor the code and co-locate parts of an algorithm, providing that method with an intention revealing name.

Comments, too, are harder to use, unless the client programmer is diligent to place them in every part of the non-local interaction. Our examination of client programs did not reveal any that were so diligent. As a consequence, programmers reading the source code are left to infer the client program’s original intent simply from the client-framework interactions themselves.

The phrase “design intent” can refer to relatively low-level details like whether or not a parameter can be null, or range as high-level as architectural design. In the context of client programs, the design intent that is difficult to express is what the collection of client-framework interactions is expected to accomplish; that is, the goal in interacting with the framework. Such design intent is approximately at the same level as that of a design pattern [23]. An example might be, “It is my intent with this client program to respond to right-click mouse events by popping up a context menu based on the item that the user has selected.”

2.3.6 Connect with External Files

The earliest object-oriented frameworks and client programs were written purely in an object-oriented language. When the client program wanted to influence the framework, for example to let the framework know that a new drawing shape had been defined for the graphical editor, the client would call a framework service method to let the framework know about this new class. One downside of this style is that since such logic is procedural, it is difficult to reason about what shapes will be available when the tool runs.

To enable easier reasoning about such questions, frameworks are increasingly relying on external declarative configuration files. The object-oriented language would still be used to define the new shape for the graphical editor, but its existence would be declared to the framework in the external file. This is a boon to frameworks such as Eclipse that use it to enable lazy-loading of classes, and to frameworks such as EJB and Struts that use it to express connections between components. All three of these frameworks use XML to express their configuration files.

These external files, however, are a difficulty for client programmers. Their object-oriented source code may be perfect but it will be completely ignored by the framework until the appropriate declarative statements exist in the configuration file. While modern IDEs understand the semantics of particular object-oriented languages, like Java, and provide warnings when the client program tries to call methods that are not defined, they have been slow to understand the connections between the configuration files and the source code. The Eclipse IDE has recently started providing warnings if classes specified in the configuration file cannot be found in the source code. The reverse is not the case, and it is difficult to provide when design intent is absent.

2.4 Differences Compared to Libraries

Both libraries and frameworks offer large scale reuse of objects. Libraries are like servants ready to do the bidding of the client program; the client program always decides when to call the library. Libraries may require that requests from the client have the right kinds of parameters or that the requests are sequenced correctly. But libraries never require that the client program has a particular structure, nor do they ever call the client program. Frameworks do call the client program, and it is this *inversion of control* that is unique to frameworks.

A library publishes a list of *provided services* that it provides for client programs. A client program is said to *require services* that are defined by the library. A client program of a framework, however, both provides and requires services of a framework. It provides services by implementing framework callback methods. It requires services by calling framework service methods.

While inversion of control is the essential difference between libraries and frameworks, other differences exist between them. It is possible for libraries to require clients to call their methods in a particular sequence, or protocol, but this is uncommon. Such a restriction requires the library itself to be stateful in order to track the protocol, and this too is less common than with frameworks, which are nearly universally stateful. A stateful library with a protocol to its methods is likely still easier to use than a corresponding framework because, with the library, a client can often localize its calls to the library. For example, a client method might call in sequence `open`, `read`, and `close` on a library that manipulates files. A corresponding framework client could be forced to call similar methods in three different callback methods, causing validation of the protocol to be more difficult. Finally, while libraries encode domain knowledge in objects, for example a `Date` class that handles leap years, such encoding is more extensive in frameworks.

2.5 Current Best Practice

Client framework programmers are provided with a variety of types of information intended to help them use a framework. This information includes: framework source code, general framework documentation like books and courses, method- and class-level API documentation like JavaDoc, targeted tutorials like HowTo articles, and example client programs. This variety of resources exists because none is a perfect match for every client programmer's problem.

```

/**
 * A listener which is notified when a viewer's selection changes.
 * @see ISelection
 * @see ISelectionProvider
 * @see SelectionChangedEvent
 */
public interface ISelectionChangedListener {
    /**
     * Notifies that the selection has changed.
     * @param event event object describing the change
     */
    public void selectionChanged(SelectionChangedEvent event);
}

```

Figure 2.2: JavaDoc for `ISelectionChangedListener`

2.5.1 Sources of Information on Frameworks

Framework source code is definitive in that it represents the implementation that the client programmer's code will be interacting with. It can answer questions related to how the framework works and, as a consequence, how the framework will interact with client code. This information on interaction policy must be teased out of code that is full of other implementation details that do not help answer the client programmer's questions. There is also the danger that a client programmer may accidentally come to depend upon private implementation details that could change later.

The framework source code can be consulted in a “first principles” fashion by the client programmer. It can be used to completely understand how the framework operates, which then enables the client programmer to devise solutions compatible with it. In principle, the framework source code can be used to verify that he has used the framework correctly but it requires significant mental energy to do so. A programmer trying to understand a client program could use the framework source code to gain understanding, but it provides no help in disentangling code that is doing multiple things.

The framework source code is often not available, especially for commercial frameworks like Microsoft .NET and Enterprise Java Beans, but in these cases the framework's API documentation may be of higher quality to compensate.

API documentation is written by the framework authors and attempts to describe just the callbacks and service methods of the framework, providing sufficient detail for client programmers to use the framework correctly, but omitting framework implementation details.

It is difficult to write high quality documentation for the traditional reasons, but additionally so because it must describe not only the particular class or method, but also provide references to other classes or methods that must be employed by the client programmer to arrive at a solution. Using a framework usually entails coordinating many classes and interfaces from the framework, and it is not obvious where in the API documentation such coordination information should live.

An example of documentation that fails to provide the needed coordination information is shown in Figure 2.2, which is JavaDoc for an Eclipse framework interface. Following the links to

the references (`ISelection`, `ISelectionProvider`, and `SelectionChangedEvent`) yields similarly terse descriptions. It fails to inform the client programmer about the traditional use of the `createPartControl` callback method and the `ViewPart` superclass. It is missing information about how to register for notifications with `addSelectionChangedListener`. It is also missing the essential but non-obvious fact that when using a `TreeView` it is safe to downcast the `ISelection` to an `IStructuredSelection`. Things are not always so difficult to deduce, however, and if the programmer were learning about resource change events then the `IResourceChangeListener` JavaDoc shown in Figure 2.3 provides a much better description of the bigger picture.

Even when the API documentation is well written there is the problem of whether or how to describe canonical examples of usage. Use of a framework means employing many non-local framework resources, so there is often no appropriate place to put the example. The documentation for these methods would be cluttered with many “if you want to do this, then ...” clauses, so in practice it is not done. This problem is often less pronounced in libraries because they usually do not require coordination of many classes and methods to accomplish a single goal. Even when they do, there may be an obvious place to place the example.

Given this difficulty, API documentation is not an ideal answer for programmers seeking to accomplish a goal with the framework. When the API documentation is complete, client programmers can use it to decide if their code uses the framework correctly, but then they still wonder if the API documentation is complete. Though tedious, they can also read the source code and cross reference the API documentation to understand what client code does.

Books and courses are traditionally used to convey general principles or philosophy regarding the framework. Generally, they cannot be searched using electronic means since, as they say on the internet, “you cannot grep a dead tree”. Books and courses are quite useful at giving the client programmer an understanding of what the framework can do, its architecture, and its commonly employed patterns, thus priming him for more detailed information later such as API documentation or HowTo articles.

Books, in particular, will have chapters that will directly help client programmers to accomplish their goals, but courses are less likely to do so. Both may provide a understanding of the framework that can help in evaluating if client code is correct, but are unlikely to provide much help at a line-by-line level of detail. Books may provide a big-picture understanding that guides a programmer’s intuition and helps him infer what a client program does.

Targeted tutorials describe a specific goal and provide instructions on how to achieve it, occasionally providing some discussion of context or points of variability. Where they exist, and are updated, they are quite effective at instructing client programmers on how to use a particular part of a framework. However, since tutorials are often found on the internet, and do not have the quality control offered by a book editor, their quality is variable and may be outdated. Some tutorials may provide the client programmer with guidance on how to confirm that his code works correctly, but others may not. Tutorials provide little help to programmers trying to understand a client program. It is straightforward to cross-reference between source code and API documentation because API documentation is organized to facilitate it, but no similar means exist for tutorials except using internet search engines.

Example client programs are known to work with the framework since the programmer can compile them himself and watch them work. As such, they provide a source of known-good

```

/**
 * A resource change listener is notified of changes to resources
 * in the workspace.
 * These changes arise from direct manipulation of resources, or
 * indirectly through resynchronization with the local file
 * system.
 *
 * Clients may implement this interface.
 *
 * @see IResourceDelta
 * @see IWorkspace#addResourceChangeListener(
 *         IResourceChangeListener, int)
 */
public interface IResourceChangeListener extends EventListener {
    /**
     * Notifies this listener that some resource changes are
     * happening, or have already happened.
     *
     * The supplied event gives details. This event object (and the
     * resource delta within it) is valid only for the duration of
     * the invocation of this method.
     *
     * Note: This method is called by the platform; it is not
     * intended to be called directly by clients.
     *
     * Note that during resource change event notification,
     * further changes to resources may be disallowed.
     *
     * @param event the resource change event
     * @see IResourceDelta
     */
    public void resourceChanged(IResourceChangeEvent event);
}

```

Figure 2.3: JavaDoc for `IResourceChangeListener`

	Understand non-local client-framework interactions	Design solutions without owning the architecture	Gain confidence of compliance	Represent solutions	Encode design intent	Connect with external files
Framework source code	Maybe	No	No	No	No	No
API documentation	Maybe	No	No	No	No	Maybe
Books and courses	Yes	Yes	No	Yes	No	Yes
Targeted tutorials	Maybe	Yes	No	Yes	No	Yes
Example client programs	No	Maybe	No	Yes	No	Maybe

Table 2.1: Helpfulness of information sources to overcome difficulties

examples. Sometimes simple internet searches will yield appropriate examples, but other times the client programmer may know, for example, that he wants to receive events about Eclipse framework file changes, but will have a hard time finding example code because a file is called a “Resource,” and he must search for that keyword. Once he finds some example code, it may be difficult to extract the essence of the interaction between client and framework, and consequently he may copy too much or too little. And of course real code has real bugs, so he may copy those as well. Despite these problems, copying examples from code has been promoted by leading framework experts [22] as the best way to learn how to use a framework. The example code, however, provides no assurance to the programmer that the code he has written is correct, nor does it help him to understand a new client program.

2.5.2 Help in Overcoming Difficulties

The effectiveness of the information sources at helping client programmers with the six difficulties is summarized in Table 2.1. Most help client programmers to translate their goals into a plan for interaction with the framework, though they vary in how directly this information is conveyed, and whether known-good examples or generalities are conveyed. These information sources also vary in the time and place in which they are available. Some, like courses, are available days or months before the client programmer sits down to write a program. Others, like books, are available when the program is being written but are printed on paper and do not afford tool assistance. An ideal solution would address all six of the difficulties and be available to the client programmer while the program is being developed and evolved.

2.6 Summary

Frameworks are a large-scale reuse mechanism, distinct from libraries in that they use inversion of control. Client programs of frameworks must implement the callback methods that the framework requires, and in turn require service methods from the framework.

Frameworks are difficult for client programmers for a number of reasons. Interactions between the client and framework are forced to be non-local, so it is difficult to find all the interactions and relate them. Since the framework owns the architecture, client programmers struggle to find a solution that is within the framework's constraints. Once they have written code, they still have difficulty gaining confidence that the code conforms to the framework constraints. If they have discovered a solution, they find it difficult to represent it so others can learn from it. In their source code, they face difficulties encoding their design intent since standard techniques like intention revealing method names and comments are hard to use. And modern frameworks increasingly require them to coordinate their object-oriented code with external configuration files in order for the client program to work correctly.

Existing sources of information about frameworks have enabled programmers to write client programs but not without difficulty. An ideal solution would address all six of the difficulties and be available to the client programmer while the program is being developed and evolved.

Chapter 3

Thesis and Hypotheses

The preceding chapters have described some of the difficulties faced by client programmers when using frameworks. This thesis proposes that an improved form of specification, called a design fragment, can help overcome the difficulties. A design fragment specifies a single known-good way to use a framework to accomplish a goal. Because design fragments are tool-readable, analysis tools can check conformance between the client program and the design fragment. Furthermore, we believe that design fragments can be used in real programming conditions.

3.1 Thesis

The thesis of this dissertation is:

We can provide pragmatic help for programmers to use frameworks by providing a form of specification, called a design fragment, to describe how a client program can correctly employ a framework, and by providing tools to assure conformance between the client program and the design fragments.

While it is our desire to directly validate that design fragments provide “pragmatic help,” it is challenging to demonstrate that claim, since a single objection could eliminate its pragmatism. It is likely that only the voluntary, widespread use of design fragments by many client programmers would represent irrefutable validation of pragmatic help. Since validation of that scope is not possible in this dissertation, we have instead identified the largest concerns about the pragmatism of design fragments, and have collected evidence to mitigate those concerns.

The first concern regards how well design fragments can be applied to real programming conditions. This includes current programming languages like Java, actual frameworks, and client programs that we find on the internet. We wish to eliminate the chance that design fragments only work on toy examples.

The second concern regards the example-based nature of design fragments. We know that frameworks can be quite large, and we also know that client programmers can be quite creative in their solutions. So our concern was that the variety of examples of client-framework interaction, and consequently the variety of design fragments, will be impractically large to enumerate.

The third concern regards our opportunity and ability to provide help through assurance tools. It was not known what kinds of constraints would be present and if we could build analyses to

check them.

To address these concerns, we have created three corresponding hypotheses, subordinate to the main thesis. Since each hypothesis is smaller than the main thesis, each can be directly supported by evidence. Taken together, the three hypotheses bolster the case that design fragments and assurance tools can provide pragmatic help.

We performed a case study using the rather small Applet framework. Its small size allowed us to directly collect some metrics, like the final size of the design fragment catalog, that for a larger framework would be too time consuming to perform during this dissertation. We have paired it with a case study on the rather large Eclipse framework to ensure our findings are replicated at larger scales.

In the following sections we state each hypothesis and summarize the evidence, presented in the remainder of this document, that supports it.

3.2 Hypothesis: Real Conditions

To demonstrate that design fragments are pragmatic, we demonstrate their use on frameworks and client programs that are used by real programmers. The first hypothesis is:

Design fragments can be used with existing large commercial frameworks, real programming languages, and off-the-shelf code.

We support this hypothesis with the following evidence:

1. Catalogs of reusable design fragments for the Applet and Eclipse frameworks (Chapters 6 and 7). Both frameworks are implemented in the Java programming language, both are commercial, and the Eclipse framework is very large.
2. Case studies of the Applet and Eclipse frameworks that include bindings between client programs found on the internet and the design fragment catalogs (Chapters 6 and 7). The client programs for the frameworks represent off-the-shelf code, not code written by researchers or for the purpose of this dissertation.
3. An inventory showing that the design fragment language can express the observed client-framework interactions (Chapter 4). Additionally, an empirical study demonstrating that the constraints expressed in design fragment catalogs cover the majority of observed client-framework interactions in a Java client program (Chapter 7).

This evidence shows design fragments to be practical in real conditions: Java frameworks and client programs found on the internet. It also shows that the design fragments can specify the client-framework interactions actually found in those client programs.

3.3 Hypothesis: Design Fragment Variety

A design fragment expresses a single known-good solution to accomplish a client programmer's goal in interacting with a framework. If there are many such solutions for each goal, then a catalog of design fragments would be impractically large. However, if there are few such solutions for each goal, then, as we survey client programs, we would expect to find fewer and fewer

new design fragments over time. The size of the design fragment catalog would be small, and therefore pragmatic. The second hypothesis is:

The variety of design fragments to accomplish a given goal is limited, so a small catalog of design fragments can have good coverage of the code seen in practice.

We support this hypothesis with the following evidence:

1. A case study of the Applet framework demonstrating the small size and asymptotic growth of the design fragment catalog (Chapter 6). That is, with each new client program whose interactions with the framework are encoded as design fragments, the chance is reduced of needing new design fragments in the catalog.
2. A case study of the Eclipse framework demonstrating that client programs show low variety in the client-framework interactions (Chapter 7). While the size of the Eclipse framework prevents us from creating a complete catalog, this measurement suggests that a complete catalog would have similar asymptotic growth as the Applet catalog.
3. Evidence that programmers copy examples, which serves to reduce the variety of observed client-framework interactions (Chapters 6 and 7).

This evidence shows that design fragments are practical because it is possible to create a small catalog of design fragments that covers a framework. We created a catalog for the Applet framework and showed it was small, and for the larger Eclipse framework we collected metrics that suggest the final size of the catalog will be small.

3.4 Hypothesis: Assurance

Simply by collecting design fragments into a catalog we have provided programmers who are designing client programs with knowledge about how to interact successfully with a framework. Such knowledge is pragmatic help at design time but, with analysis tools that ensure conformance, the aid provided by design fragments can continue through the evolution of the client program, and even to other programmers. The third hypothesis is:

Analysis can provide programmers with assurance that their code conforms to the constraints of the framework as expressed in design fragments.

We support this hypothesis with the following evidence:

1. An implementation of simple static analysis for conformance assurance (Chapter 5). These analyses include *required method call*, *required new instance*, *required class reference in XML*. These analyses are practical in that they run quickly enough to be used interactively inside a modern integrated development environment.
2. A description of what percentage of design fragment constraints can be checked through analysis (Chapter 5).
3. An empirical study that describes additional behavior specifications that should be possible to check through analysis (Chapter 4).

This evidence shows that assurance tools can provide pragmatic help because we built some tools to check some constraints, and there are opportunities for more constraint checking analyses.

3.5 Integration

When turned into a simple question, the main thesis becomes “Can design fragments and assurance tools provide pragmatic help to client programmers?” All three hypotheses address the “pragmatic help” part of the thesis, each either demonstrating a particular way that help is provided, or by mitigating a concern about the pragmatism of the provided help.

Our intent with a broad thesis statement is to push our research to devise a realistic and practical solution to the challenges faced by client framework programmers, knowing that the validation of such a thesis could not be complete. These three hypotheses taken together do not remove all concerns about whether design fragments and assurance tools can provide pragmatic help. We discuss the relationship between the hypotheses and the main thesis in detail in Section 9.1. However, the validation of each hypothesis provides valuable knowledge on its own, and the sum of the hypotheses substantially supports the conclusion that design fragments provide pragmatic help.

The next several chapters of this document provide the evidence that supports the hypotheses presented above. Chapter 4 describes the design fragment language. Chapter 5 describes the assurance tooling that plugs into the Eclipse integrated development environment. Chapters 6, and 7 describe case studies on Applet and Eclipse frameworks.

Chapter 4

Design Fragments

Previous chapters have explained the problems that client programmers face when using frameworks, motivating the need to provide help. In this chapter we present the design fragment language, a language which can express ways for a client program to successfully use a framework to accomplish a goal. Design fragments can be bound to the client program, which captures design intent and enables conformance assurance. We also discuss how design fragments can be used in the software lifecycle, and the benefits they provide.

4.1 What a Design Fragment Is

A design fragment is a specification of how a client program can use framework resources to accomplish a goal. From the framework, it identifies the minimal set of classes, interfaces, and methods that should be employed. For the client program, it specifies the client-framework interactions that must be implemented. The structure of the client program is specified as roles, where the roles can be filled by an actual client program's classes, fields, and methods. A design fragment exists separately from client programs, and can be bound to the client program via annotations in their source code. These annotations express design intent, specifically that it is the intention of the client programs to interact with the framework as specified by the design fragment.

Design fragments are partial specifications, and do not attempt to fully specify either the framework or the client program. They rely on the existence of framework API documentation and the competence of client programmers. As partial specifications, their value lies in their identification of the required framework resources and their description of how the client program can use them to accomplish a goal. Without design fragments, finding a solution can be difficult and error-prone, but once the needed resources and solution have been identified, client programmers can accomplish their goal using their existing programming skills.

To provide assurance that client programs conform to the specification, design fragments are machine-readable. The bindings between design fragments and client programs preserve the intentional connections between non-local parts of the client program, enabling analysis tools to check conformance both immediately and as the program evolves.

A design fragment provides a programmer with a “smart flashlight” to help him understand

the framework. This smart flashlight illuminates only those parts of the framework he needs to understand to accomplish his goal. Without the smart flashlight, a programmer browsing the framework classes can be swamped with implementation details or unable to differentiate the relevant from irrelevant. Even simple parts of a framework have enormous complexity [34]. In the Java Swing user interface framework, the JButton class has 160 methods and fields, while JTree has 336. A design fragment provides a programmer with a mapping from a goal to a handful of framework resources, focusing his attention on what is relevant.

Each design fragment describes how a client programmer can accomplish a goal, and this goal is usually a simple task. There are two special sub-types of design fragments with special goals. The first is a design fragment whose goal is to facilitate event bookkeeping. Client programmers may have difficulty understanding which events the framework can emit, and further difficulty knowing when to register and deregister for those events. This type of design fragment connects the two by identifying both a framework event and the lifecycle callback methods where registration and deregistration are conventionally done. Such a design fragment is additionally helpful because event deregistration is easy to forget, non-local in the source code, and can be hard to find during testing.

The second sub-type of design fragment helps a programmer to work around conflicting constraints. Sometimes a goal becomes difficult to achieve only because of constraints imposed by the framework, since the client programmer is able to easily solve the problem if this were a standalone program. One example of this is in the Applet framework where client programmers are forced to use threads to update their display; another example is in the Enterprise Java Beans framework where client programmers are forced to create an external clock program that emits periodic events because EJB's cannot act spontaneously. The client programmer must understand the constraints of his domain, and the constraints of the framework, then find a solution within the intersection, a solution that is unconventional in one or both of the domains. Here, design fragments are valuable since the programmer's existing intuition on how to solve problems may not quickly guide him to a solution.

4.2 Contents of a Design Fragment

A design fragment is a specification that expresses a conventional solution to how a client program interacts with a framework to accomplish a goal. It has the following four parts:

1. The name of the design fragment.
2. The goal of a client programmer that is accomplished by this design fragment.
3. A description of the relevant parts of the framework that interact directly with the programmer's code, including the callback methods that will be invoked, the service methods that are provided, and other framework classes that are used.
4. A description of what the programmer must do to accomplish the goal of this design fragment, including the classes, methods, and fields that must be present. This description also includes the behavior of the methods.

In this section, we describe the design fragment language. We begin with how the language expresses object-oriented structure, proceed to the expression of external configuration files, and

conclude with the expression of other constraints.

4.2.1 Object-Oriented Language

Object-oriented frameworks are built in object-oriented languages, so it is important to express that structure, including classes, interfaces, fields, and methods. The design fragment has separate sections for the parts of the framework and the parts of the client program, which we refer to as “framework provided” and “programmer responsibility.” In both sections, the specification simply declares that a class, interface, field, or method exists. Relationships between classes and interfaces are expressed with the *implementsinterface* and *superclass* declarations. Inheritance and interfaces in a design fragment work the same way as they do in Java, which means single inheritance for classes and multiple implementations of interfaces.

By way of example, Figure 4.1 shows the structure for the Background Continuous V1 design fragment. The horizontal bar in the diagram separates the framework provided classes from the ones that must be built by the programmer. Note that only the relevant subset of framework classes, interfaces, and methods are shown. A design fragment should not reveal implementation details of the framework, even if client programs have visibility to them because of their privileged subclass relationship. Consequently, we say that a design fragment respects the encapsulation boundary between the framework and the client program, describing only those framework resources the client program needs to reference to do its job.

The classes and methods in the programmer responsibility section are roles in the design fragment, and can be bound to the programmer’s classes, as we describe in section 4.3.

Entire design fragments can be marked as deprecated. Such a facility would be helpful when a bug is found in the design fragment, since it would enable clients to know that using that design fragment is not recommended. Our example design fragment, Background Continuous V1, is in fact a deprecated design fragment that has been replaced by V2, which has fixed a threading bug.

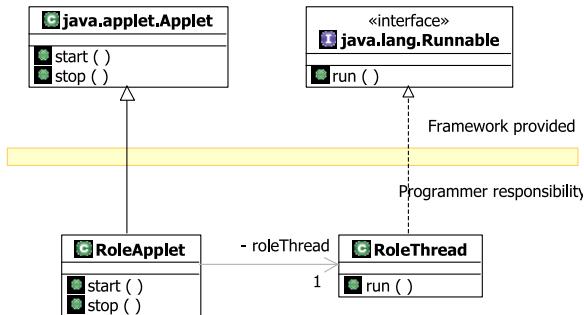


Figure 4.1: UML structure of Background Continuous V1 design fragment

Figure 4.2 shows the abstract syntax of the object-oriented parts of the design fragment language, and substantially follow the Extended Backus-Naur Form (EBNF) [20]. To make the definitions representing unstructured text more readable, we have shown them in italics. We will subsequently add in the parts for external configuration files and specifications. The parts shown in bold are forward references and are defined in Figures 4.3 and 4.5.

```

design-fragment = name-text , goal-text , [ is-deprecated ] ,
    framework-provided , programmer-required ;
is-deprecated = "yes" | "no" ;
framework-provided = { type } ;
programmer-required = { type } , [ { xml-file } ] ;
type = class | interface ;
class = name , [ superclass ] , [ implements-interface ] ,
    [ { field } ] , [ { method } ] , [ { class-spec } ] ;
interface = name , [ superclass ] , [ { field } ] , [ { method } ] ;
superclass = name ;
implements-interface = name ;
field = name , field-type-text , [ { field-spec } ] ;
method = name , return-type-text , [ { argument } ] ,
    [ is-abstract ] , [ source-code ] , [ { method-spec } ] ;
argument = name , argument-type-text ;
is-abstract = "yes" | "no" ;

```

Figure 4.2: Abstract syntax of the core design fragment language

The design fragment language's concrete syntax is XML, which was chosen because it is standardized and familiar to client programmers. We have defined a full XML schema definition [27] for the language in Appendix B. The concrete syntax closely follows the abstract syntax, as is shown in this short excerpt that shows the RoleThread class from our Applet example:

```

<class name="RoleThread">
    <implementsinterface name="java.lang.Runnable" />
    <method name="run" returnvalue="void" />
</class>

```

4.2.2 Configuration File Language

While the earliest frameworks were written entirely in a programming language, modern frameworks are increasingly combining a programming language (such as Java) with an external declarative configuration file (often XML¹). Examples of such frameworks are Eclipse, Enterprise Java Beans, and Struts. The configuration file is often used to specify architectural connections or to enable extension of the framework after its source code has been compiled and deployed. Elements of the source code, such as class names, must correspond to declarative statements found in the XML configuration file or else the source code will be ignored by the framework and fail to extend it as desired. Inaccuracies in a framework's configuration file can prevent a program from working, and so are no less serious than bugs in the source code.

The design fragments language can specify the required structure of an external XML configuration file. Figure 4.3 describes the abstract syntax, where the bolded part is again a forward reference to the specifications that will be defined in Figure 4.5. As you can see, **xml-file**

¹To be clear, the design fragment source files are themselves also expressed in XML. But here we are discussing how the design fragment language can refer to XML configuration files from a framework.

contains a single `xml-node`, and `xml-node` is recursively defined to allow for a tree of `xml-nodes`. Roles can be specified in the design fragment, which requires that programmers who bind this design fragment must also bind this XML role.

```
xml-file = name , xml-node ;
xml-node = name , [ { xml-node} ] , [ xml-role-name-text ] ,
[ { xml-spec } ] ;
```

Figure 4.3: Abstract syntax of the XML design fragment language

As an example, Figure 4.4 shows an excerpt from a design fragment that describes the `plugin.xml` file from the Eclipse framework. The entire excerpt is an `xml-file`, within which the name of the configuration file is specified, and the required parts of the configuration file are shown as `contents`. We have followed the convention that anything that is present in the design fragment must be found in the configuration file, but the configuration file may contain more items not mentioned in the design fragment. So, since an XML node named `propertyTab` with an attribute `category` equal to `acmeelements` is mentioned in the design fragment, it must also be present in the `plugin.xml` configuration file.

Not all of the nodes are equally interesting to us from a specification standpoint. We assign role names to the ones that we refer to elsewhere, as seen in the `<df:role name="Role-PropertyTab" />` node in our example. This specification means that the `propertyTab` XML node is assigned to the role named `RolePropertyTab`. In order to keep the syntax of references simple, the design fragment language only allows us to refer to role names, not to an arbitrary XML node. This example also demonstrates the use of an XML string matching specification, which we will return to in the next section, which describes the specifications.

Since we can refer to roles in Java code and in XML files, we can write specifications that refer to either the Java or the XML or both, as we discuss next.

4.2.3 Specification Language

Of course not all constraints can be expressed through a simple description of the structure, like the UML structure we saw in Figure 4.1. In that example, it is necessary to specify the threading behavior of the Applet and coordinate it with the framework callback methods. The behavior that the design fragment should specify is that in the `start` callback, the code should create a new `RoleThread` instance and assign it to the `roleThread` field in the `RoleApplet`. In the `stop` callback, the code should set the `roleThread` field to null. We use the null field as a signal to the thread that it should terminate. Then, in the `run` callback, the code should loop repeatedly while checking that the `roleThread` field has not been set to null.

Most of this behavior can be expressed in the design fragments language. The required creation of a new instance is expressed like this:

```
requirednewinstance target="java.lang.thread.Thread" arguments=""
```

A required method call is expressed like this:

```
requiredcallspec targetobject="roleThread" targetmethod="start"
arguments=""
```

```

<xmlfile>
  <name>plugin.xml</name>
  <contents>
    <plugin>
      <extension point="propertyTabs">
        <propertyTabs contributorId="MultiPageEditor">
          <propertyTab category="acmeelements">
            <df:role name="RolePropertyTab" />
          </propertyTab>
        </propertyTabs>
      </extension>
      <extension point="propertySections">
        <propertySections contributorId="MultiPageEditor">
          <propertySection>
            <df:role name="RolePropertySection" />
            <df:match reason="Matches our RolePropertyTab"
              p1="xpath://RolePropertySection@tab"
              p2="xpath://RolePropertyTab@id" />
            </propertyTab>
          </propertyTabs>
        </extension>
      </plugin>
    </contents>
  </xmlfile>

```

Figure 4.4: Design fragment with XML specifications

Additionally, freeform text can also be entered as a specification:

```
freeformspec text="By the end of this method, a new thread must  
be running and assigned to the roleThread field"
```

However, note that the looping behavior required in run cannot be expressed in the current design fragments language except as freeform text. Here we see the specification for the start method with the specifications in bold:

```
<method>  
  <name>start</name>  
  <returnvalue>void</returnvalue>  
  <requirednewinstance target="java.lang.Thread" arguments="" />  
  <requiredcallspec targetobject="roleThread" targetmethod="start"  
    arguments="" />  
  <b><freeformspec text="By the end of this method, a new thread must  
    be running and assigned to the roleThread field" /></b>  
</method>
```

The above example shows how we can specify the required behavior of the client program, but the client programmer also needs to know how the framework will behave. We can also put specifications on the framework classes that describe their behavior. Some framework methods are callbacks, while others are service methods. Some callback methods follow the lifecycle of the object, while others represent spontaneously generated events. In the case of the Applet example, the programmer would want to know that the start framework method is a callback, it can be called multiple times, it is a lifecycle callback, and it is invoked in pairs with the stop callback method. These framework specifications are purely descriptive, unlike the prescriptive specifications for the client program, so they would not be checked by analysis tools. We can document this framework behavior by providing the following specification for the Applet class:

```
<class name="java.applet.Applet">  
  <method name="start" returnvalue="void">  
    <b><freeformspec text="Callback method; invoked when framework  
      decides to initialize your applet." /></b>  
    <freeformspec text="invocation-cardinality '*' " />  
    <freeformspec text="invocation-lifecycle 'yes' " />  
    <freeformspec text="invocation-type 'callback' " />  
    <freeformspec text="invocation-pair 'stop' " />  
    <freeformspec text="invoked-before 'stop' " />  
  </method>  
  ...  
</class>
```

Figure 4.5 describes the abstract syntax for the specifications that we have defined. There are four types of specifications, corresponding to classes, fields, methods, and XML files. Each specification, such as a required call specification, is an instance of one of these four types. While it is possible within our language to define specifications for fields, we have not yet created one.

```

class-spec = class-referenced-in-xml-spec ;
class-referenced-in-xml-spec = xml-role-name ,
    attribute-name-text ;
field-spec = ; (* no field specs pre-defined *)
method-spec = freeform-spec | required-call-spec |
    required-new-instance-spec ;
freeform-spec = specification-text ;
required-call-spec = target-class-name ,
    target-method-name , arguments , purpose ;
required-new-instance-spec = target-class-name ,
    arguments , purpose ;
xml-spec = string-match-spec | attribute-exists-spec ;
string-match-spec = specification-purpose ,
    xpath-predicate-1 , xpath-predicate-2 ;
attribute-exists-spec = attribute-name ;

```

Figure 4.5: Abstract syntax of design fragment specifications

Returning to the example from Figure 4.4, we can see the use of an XML string match specification. We need the specification because the example declares both a *tab* and a *section* that goes on that tab. The programmer must declare that the section corresponds to the tab by giving the tab an ID, and referencing that ID when declaring the section. We specify this constraint with the `<df:match ... />` specification in our example. The specification says that the `RolePropertySection` has an attribute called `tab`, and that it must equal the `RolePropertyTab`'s attribute called `id`. The programmer must also show the correspondence between the tab and section declared in the XML file and the Java classes that implement them. This is done by putting the name of the Java class in the XML file. We can specify this correspondence with the class referenced in XML spec, like this:

```
<class-referenced-in-xml xmlrolename="RolePropertySection" at-
tributename="class" />
```

Table 4.1 summarizes the specifications in the current version of the design fragment language. It lists the two behavioral specifications, Required New Instance and Required Method Call, the structural specifications, and the Freeform Text specification. The column titled “Structured” indicates if the specification has internal structure defined in the XML; this structure is defined in the detailed description of the specifications, below. It is structure like this that makes the specifications parsable and analyzable by tools. The column titled “Checked” indicates if an implementation exists to check that specification.

The specifications and analysis implementations are described in more detail below. All of the specifications apply to methods in a design fragment, with the exception of the Class in XML Extension Point, which applies to a class.

Required new instance. This specification means that within this method the source code should create a new instance of the specified class/interface, or one of its subclasses. This specification is structured and has the following attributes: *target* (required), the class of the new instance; *arguments* (required), the arguments to the method; *purpose* (optional), a string describ-

Specification	Structured	Checked
Behavior specifications	Yes	Yes
Required new instance		Yes
Required method call	Yes	Yes
Structural specifications	Yes	Yes
Class in XML		No
String match in XML	Yes	No
XML element has attribute	Yes	No
Other structure specifications	Yes	No
Freeform text	No	No

Table 4.1: Specifications, checked and unchecked

ing the purpose within the design fragment for creating an instance; *text* (optional), a freeform string for any comments.

The implementation of this analysis is imprecise in that it does not check which constructor is invoked on the target class. Ideally the semantics of this specification would cover cases where the new instance is created in a subroutine, but this is not implemented. Our current convention is to use the unstructured *text* attribute to identify what field or role this new instance is stored in, but this informality limits our analysis.

Required method call. This specification means that within this method the source code should call the specified method on an instance of the specified class. This specification is structured and has the following attributes: *targetobject* (required), the class/interface of the object or role on which the target method is called; *targetmethod* (required), the method on the target object that is called; *arguments* (required), the parameters passed to the target method; *purpose* (optional), a string describing the purpose within the design fragment for calling the method; *text* (optional), a freeform string for any comments.

The implementation of this analysis is imprecise in that it only checks that a method with the same name as *targetmethod* is called, not that it has the same arguments or is on the right type of object. This has the effect that it will incorrectly accept, for example, `open` called on a `File` when the specification is to call `open` on a `Socket`. Despite this, in our experience almost all errors are ones of omission, not ones where a method with the same name is invoked on the wrong target, so this limitation on the implementation is of little consequence in detecting problems. A much greater inconvenience is the previously mentioned restriction that the semantics of this method should cover cases where the method call happens in a subroutine, as a simple refactoring of the source code can cause this specification to fail.

Class in XML. This specification means that this class must be referenced in the specified external configuration file as the specified attribute name. For programs that use the Eclipse framework, and many other modern frameworks, the deployed framework consists of both Java source code as well as external configuration files used in configuring the program. In the case of Eclipse, a program can extend the framework by writing Java code that implements the extension and binding that code through a declarative statement in a file called `plugin.xml`. This specification is helpful to client programmers since the traditional in-Java type checking cannot detect a missing declaration in the `plugin.xml`; instead the program will fail silently.

This specification is structured and has the following attributes: *xmlrolename* (required), the rolename found in the `plugin.xml` file; *attributename* (required), the name of the attribute in the specified *xmlrolename* where this class's name must be found. In the same way that programmers add bindings to Java files to bind their design fragments, they can also add bindings to XML files – this is the *xmlrolename* that must be found in the `plugin.xml` file. The implementation of this structural analysis is straightforward and has no known limitations.

String match XML. This specification means that within an XML file the two specified strings must match. The two strings are specified as XPath expressions (pointers into XML files) or constants. Design fragments can use this specification either to coordinate different parts of the same XML file to be consistent or to ensure that particular parts of the XML file have a particular constant value. This specification is structured and has the following attributes: *reason* (required), a string that explains why the design fragment believes these two strings should match; *p1* and *p2* (required), XPath expressions that point into the XML file and resolve into strings, or a constant. This specification was implemented and working but evolution of the tool has rendered it inoperable for now.

XML element has attribute. This specification means that an XML node must have the specified attribute, regardless of that attribute's value. Design fragments can use this specification to ensure that required attributes are present in XML files. This specification is structured and has the following attributes: *name* (required), the name of the attribute that must exist. This specification has no implementation yet.

Freeform text. This specification allows a design fragment author to write a freeform textual specification. A report on the kinds of constraints informally encoded with this specification is presented in Section 9.2.3.

4.3 Design Fragments and Client Programs

Design fragments exist separately from source code. They are an abstract expression of how a client program can interact with a framework. They can be connected with a particular client program by *declaring* an instance of the design fragment and *binding* that instance to the client program. In doing this, the client programmer expresses the design intent that the client program interacts with the framework in accordance with the design fragment's specification.

When a programmer wants to use a design fragment, he declares it using a Java 5 annotation [25] in the source code of the client program. Java 5's annotations are a standard mechanism in the Java language and the annotations can be typechecked using the Java 5, or subsequent, compiler. A new instance of a design fragment is created through the `@df.instance` annotation, providing a reference to the design fragment name, and giving this instance a unique name. For example:

```
@df.instance(df="TreeView", inst="tv1")
```

If the program uses the design fragment twice, there would be two such annotations, each with a different instance name, but both referring to the same TreeView design fragment. Design fragment instance declarations are placed in the special Java file `package-info.java` to share them across the whole Java package, rather than in any of the files corresponding to Java classes.

Once a design fragment instance has been declared and named, its roles can be bound to the corresponding classes, fields, and method in the client program. These bindings are also expressed as Java 5 annotations. Here we see bindings from the `tv1` instance's `RoleView`, `roleViewer`, and `createPartControl` roles to the corresponding class, field, and method in the source code:

```

@df.bindings({
    @df.binding(inst="tv1", role="RoleView")
}) public class SampleView extends ViewPart {

    @df.bindings({
        @df.binding(inst="tv1", role="roleViewer"),
    }) private TreeViewer viewer;

    @df.bindings({
        @df.binding(inst="tv1", role="createPartControl"),
    }) public void createPartControl(Composite parent) {
    ...
}

```

These bindings declare the programmer's intent that the roles in the `TreeView` design fragment are to be played by the classes, fields, and methods that he identifies with annotations.

Bindings in an external configuration file are made in a similar way to the bindings in Java source code. However, since XML does not have Java 5 annotations, we have used XML comments, which begin with `<!--` and end with `-->`. An example of an XML binding is:

```
<!-- @dfbinding inst="tv1" role="RolePropertyTab" -->
```

With a binding, the programmer is stating that a particular XML node corresponds to a role defined in the design fragment. Since XML has a hierarchical structure, the role binding is positioned as a child of the particular XML node.

A class, field, method, or XML node in a client program or configuration file may exist to satisfy more than one goal of the programmer. This condition is known as *tangling* of concerns [64] because, when a programmer tries to evolve the code, he finds that he must disentangle the client-framework interactions that exist. Similarly, the code that implements a single goal of a programmer may be *scattered* across many classes, fields, and methods. Design fragment annotations help to make tangling and scattering visible to programmers. When, for example, a class plays a role in more than one design fragment, the class will have more than one binding. Similarly, following the role bindings for a single design fragment may lead the programmer to many classes.

Each framework will have its own catalog of design fragments that act like a handbook, collecting conventional solutions to problems. Programmers can see examples of the design fragment in use by navigating from the catalog to the code that implements a given design fragment.

Authors collect their design fragments into catalogs for ease of management. A catalog contains design fragments for a particular framework or version of a framework. Catalogs can also be categorized, for example by how stable the design fragments are. We have catalogs for stable, testing, and unstable design fragments for the current version of the Eclipse framework. A

```

@df.instances({
    @df.instance(df="TreeView", inst="treeview1"),
    @df.instance(df="ContentProvider", inst="contentprovider1"),
    @df.instance(df="DoubleClickAction", inst="doubleclickaction1"),
    @df.instance(df="DrilldownAdapter", inst="drilldownadapter1"),
    @df.instance(df="LabelProvider", inst="labelprovider1"),
    @df.instance(df="RightClickMenu", inst="rightclickmenu1"),
    @df.instance(df="ToolbarMenuAction", inst="toolbaraction1"),
    @df.instance(df="ToolbarMenuAction", inst="toolbaraction2"),
    @df.instance(df="PulldownMenuAction", inst="pulldownaction1"),
    @df.instance(df="PulldownMenuAction", inst="pulldownaction2"),
    @df.instance(df="ViewerSorter", inst="viewersorter1"),
    @df.instance(df="SharedImage", inst="sharedimage1"),
    @df.instance(df="SharedActionImage", inst="sharedactionimage1"),
    @df.instance(df="SharedActionImage", inst="sharedactionimage2")
})
package eclipseTreeView.views;

```

Figure 4.6: Instance declarations reveal architectural understanding

client programmer can specify which catalog they prefer when instantiating the design fragment, like so:

```
@df.instance(df="TreeView", inst="tv1", branch="unstable")
```

Programmers with low tolerance for risk would choose the stable catalog, but those involved with the development of new design fragments could specify a newer, less tested version from the testing or unstable catalogs.

It is possible to understand how a client program interacts with a framework by reading its declarations of design fragment instances. This kind of architectural understanding is usually difficult to extract because the framework owns the architecture, and the client-framework interactions are not localized within the client program. Figure 4.6 shows the design fragment instance declarations on an Eclipse client program. A programmer who scans this list of design fragments used will be rewarded with a high-level understanding of how this program interacts with the Eclipse framework. In fact, because there are two uses of the ToolbarMenuAction design fragment, we would correctly guess that this menu has two items. In a way, this transparency of the design intent is very similar to the transparency seen in programs that use libraries.

4.4 Expressiveness

The core of the design fragment language allows the expression of a predefined set of structural constraints as predicates. The structural constraints correspond directly to what can be expressed in object-oriented languages, constraints such as the existence of classes, interfaces, fields, and methods; and features of them, such as deriving from a particular superclass, or implementing

Framework Interaction Type	Expressed by DF Language
Method invocation	Yes
Overrides method	Yes
New instance	Yes
Static class reference	Yes (Undifferentiated)
Static field access	Yes (Undifferentiated)
Extends superclass	Yes
Implements interface	Yes
Static method invocation	Yes (Undifferentiated)
Field holding framework object	Yes

Table 4.2: Expressiveness of design fragment language

an interface. Mainstream, commercial object-oriented programming languages like Java and C# are very close in their core elements, so the expressiveness of the core language may apply to languages other than Java.

In addition to expressing the core of object-oriented languages, we needed the design fragment language to express client-framework interactions. To do this, we performed a detailed syntactic examination of an Eclipse client program, and identified nine client-framework interaction types. This program and our process are described in Chapter 7. Table 4.2 lists the framework interaction types and shows that all of them can be expressed in the design fragment language. The three interaction types specific to static classes, fields, and methods are listed as “undifferentiated” in that the design fragment language does not distinguish static from non-static access. As will be discussed in detail in Section 7.4.1, each of these interaction types has the characteristic that it can be identified through simple syntactic examination of a client program.

As we discuss in Section 9.2.6, there are a number of areas where the design fragment language could be improved to increase expressiveness.

4.5 Using Design Fragments

In an effort to reduce the barrier to using design fragments, we have deliberately kept the language simple. As a consequence, client programmers should be able to quickly comprehend the first design fragment they see.

No large upfront investment is required to use design fragments. A client programmer can bind a single design fragment to a client program, yielding benefits for that client-framework interaction. The benefit grows linearly with the number of bound design fragments. As we will see in Section 7.4, our case study shows that a majority of client-framework interactions can be described with a fully populated catalog of design fragments. The benefits of design fragments follow a gentle slope with programmer effort, and provide immediate benefit with each investment of effort.

The simple design fragment language should encourage many design fragment users to become authors. An author would first identify the required framework resources, then describe what roles the client program must implement. The name and goal of the design fragment are

unstructured text, so they should be easy to write. Providing specifications is only slightly more complicated, since they can start out as *freeformspecs*, which are also unstructured text.

To add more precision, the unstructured specifications can be transformed into structured ones, either using our provided set of specifications, or by creating a new kind of specification. Defining a structured specification without automated checking allows the work of creating a checking tool to be deferred, while capturing precise design intent today. The final step would be to create the checking tool, whose implementation could be improved over time.

As with the use of design fragments, the benefits of authoring design fragments follow a gentle slope with author effort. The benefits jump in value with transitions from a design fragment with unstructured specifications, to one with structured specifications, and then checked specifications. Appendix A walks through the process of adding a new specification to the language and tools.

4.6 Design Fragments in the Software Lifecycle

Design fragments will be applied at various times in the software lifecycle. The following narrative serves to describe the contributions that occur. Our client programmer, Pat, wants to create a menu in her client Eclipse program. She scans the catalog of design fragments for the Eclipse framework and finds Toolbar Menu Action, whose goal is stated as “Add an action to a toolbar menu.” She binds this design fragment to her code, creating the necessary methods and implementing the necessary interfaces. Once she finished the binding, the analysis tools warn her that the code does not yet make the needed method calls to the framework and so she implements these, which makes the tool warnings go away. She further verifies that the textual specifications in the design fragment have been satisfied since the tools cannot yet check all the constraints imposed by the framework.

Mary, another programmer, later maintains the same code and sees the design fragment bindings in the code. She sees that the tool is happy with the checkable constraints. She navigates from the code to the design fragment catalog, where she can see this use of the Toolbar Menu Action design fragment as well as all the others. She also sees that Pat’s code implements other design fragments and gets a big-picture overview of how Pat’s code interacts with the framework.

Janice, the author of the Toolbar Menu Action design fragment, finds that it fails to handle a special case of one of the framework parameters. She updates the design fragment definition, meaning that some of the uses of this design fragment will now report warnings.

When Mary next looks at the code she sees that the code now has a warning. She reads the warning and any other text that Janice put in the design fragment and changes the code so that it is again in conformance. The code had originally passed testing but the testing had not caught this bug so she is thankful that the strengthening of the design fragment has helped her increase the code quality.

Art, the author of analysis tools for design fragments, puts the finishing touches on his new constraint analyzer. Previously, design fragment authors had put specifications into the code that could not be checked by tools and so client programmers were expected to check them manually. Even when the programmers did check these constraints, the code sometimes wandered out of compliance through evolution. Art’s new analysis tool is able to check these constraints. Since

many existing programs already contain enduring design intent, specifically that they implement the Toolbar Menu Action design fragment, these programs now show new warnings of non-compliance that are investigated and fixed by their programmers.

4.7 Benefits

Design fragments provide two immediate benefits to programmers. First, programmers can quickly scan a catalog of design fragments to find known-good solutions to their problems. Second, analysis tools can check conformance between the programmer's stated intent and his source code. This conformance checking provides programmers with immediate value for their investment of effort.

Once programmers have started using design fragments, long-term benefits arise. Since programmers have expressed their intent, such as this code follows the Toolbar Menu Action design fragment, it becomes possible to analyze the code with respect to that intent. Even if analysis tools are not available at the time the code is written, the expression of intent endures and can be checked by stronger tools available tomorrow. For example, our current tools cannot check for concurrency bugs today, but, in the future, static analysis tools from the Fluid project [26] could assure correct threading behavior.

The use of design fragments allows other programmers to comprehend the code more quickly. At a glance it is possible to see what design fragments a program is using. Design fragments convey architectural information that is complimentary to an Acme [24] architecture model.

Evolution of code is improved. With copy-and-paste reuse of example code, it is a maintenance burden to find and revise all of the copies when a bug is found in the original. A design fragment can be marked as deprecated, causing programmers to examine their code and fix the bug. Note that in the example there is no single method or class that can be deprecated, only the collection of methods and classes that are used in a particular way.

Unnecessary code diversity can be reduced. Instead of a goal being accomplished slightly differently by various programmers on the same team, they could standardize on a particular design fragment. This yields benefits in code comprehension, and may reduce bugs. Framework authors could deliver both example applications, as they do now, as well as a catalog of design fragments. The design fragments could act as seed crystals so programmers would use the conventional solution unless they had a good reason to deviate.

4.8 Summary

A design fragment is a pattern that encodes a conventional solution to how a program interacts with a framework to accomplish a goal. It has four parts: a name, a goal, a description of what the programmer must do, and a description of the relevant parts of the framework. Design fragments describe both structure and behavior. External XML configuration files can also be specified.

When a programmer wishes to use a design fragment, he declares a new instance of one, providing that instance a name. He then binds each role in the design fragment to the corresponding part of his program. Each binding provides the name of the design fragment instance,

the name of the role in the design fragment, and the part of the program to be bound. Bindings in source code are expressed as Java 5 annotations, and bindings in external configuration files are expressed using XML comments.

The design fragment language comes with several specifications that can be placed on methods, classes, and XML nodes. Informal specifications can be written in natural language. New specifications and specification checkers can be added by design fragment authors. Specifications can be written today that express design intent, allowing the possibility of future checkers to provide assurance.

Chapter 5

Tools

We built new tools into an integrated development environment to demonstrate how design fragments can provide help to client programmers. These tools manipulate design fragments, connect them with the source code, and check conformance. Five tools are user interface views that are visible to programmers: the Catalog View, the Instances View, the Problems View, the Context View, and the Slice View. Analysis tools check compliance between source code and the design fragments, but are invisible to the user except for their output that appears in the views.

The tools are built into the Java Development Tooling for the Eclipse integrated development environment (IDE) [12]. Programmers working in the Eclipse IDE can use the design fragment tools in a style that is consistent with the incremental compilation tools already present in the IDE.

In this chapter, we describe each of these views in detail and describe how they can be used to overcome the difficulties with frameworks we have identified. Before going into detail, however, we provide an overview of the tools in context. The full IDE screenshot in Figure 5.1 shows how a programmer might configure the views inside the Eclipse IDE.

In the top left, labeled (1), is the Catalog View. It has been opened to reveal the Right Click Menu design fragment. Below it, labeled (2), is the Instances view. It shows the instance of the Right Click Menu named `rightclickmenu1`, and the checkmark next to the specification indicates that it has been analyzed and found to be satisfied. The `rightclickmenu1` instance is declared in the `package-info.java` file shown in the Eclipse editor, labeled (3) in the figure. Another Eclipse editor is labeled (4) and displays the contents of the `createPartControl` method. One of the lines in this method has been commented out, resulting in the warning shown above it on the binding for the Drill Down Adapter design fragment. The same warning also appears in the Context View, labeled (5) and in the Eclipse Problems View. Since the programmer's cursor is in the `createPartControl` method, the Context View displays all of the specifications for that method, and the design fragment instances that contribute those specifications. On the right, labeled (6) is the Slice View. It shows the relevant source code for the Right Click Menu design fragment instance named `rightclickmenu1`, since that instance has been selected by the programmer in the Instances View.

These tools help client programmers to overcome the difficulties of using frameworks. The difficulties we identified in Chapter 2 are: understanding non-local client-framework interactions, designing solutions without owning the architecture, gaining confidence of compliance,

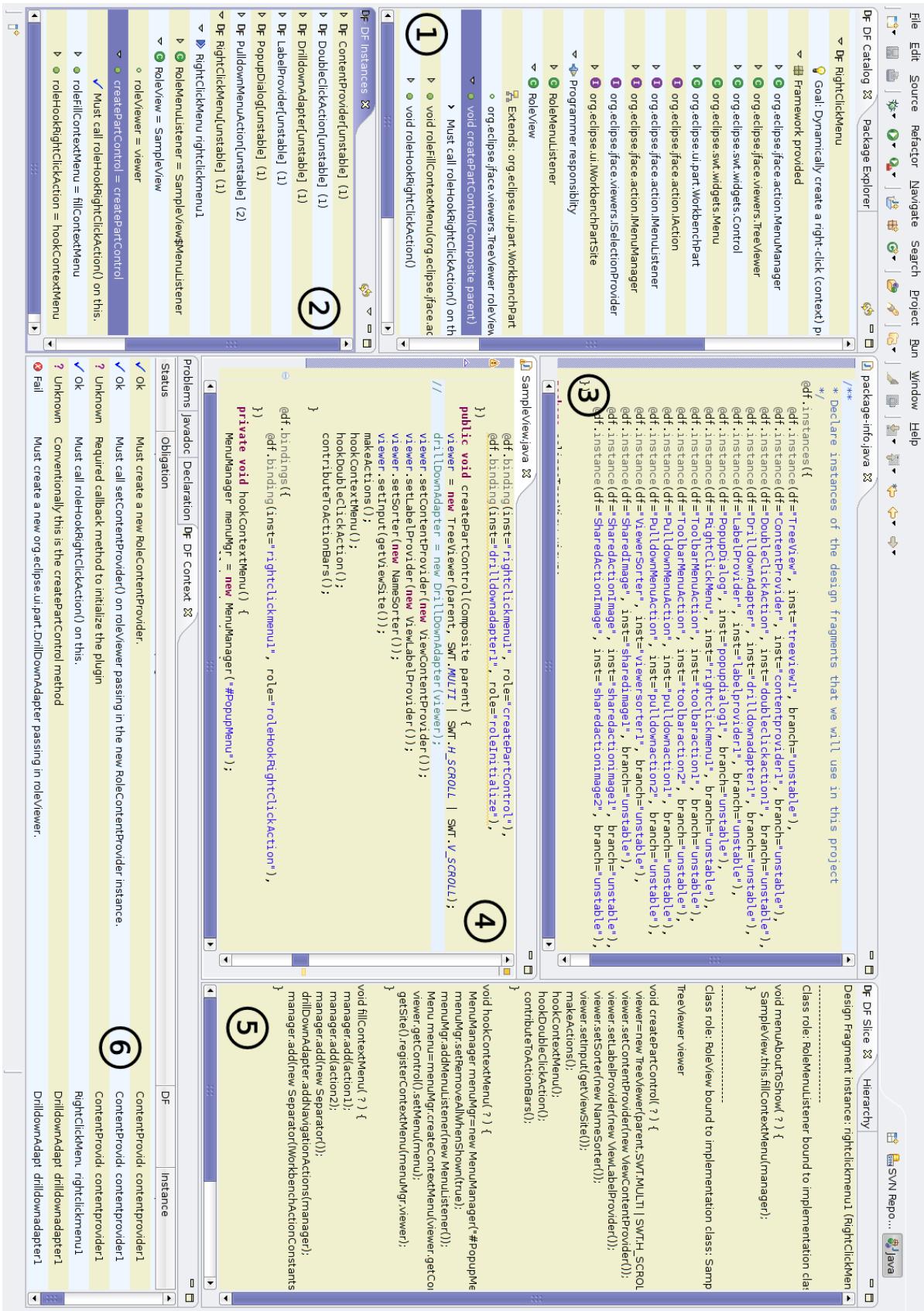


Figure 5.1: All tool views

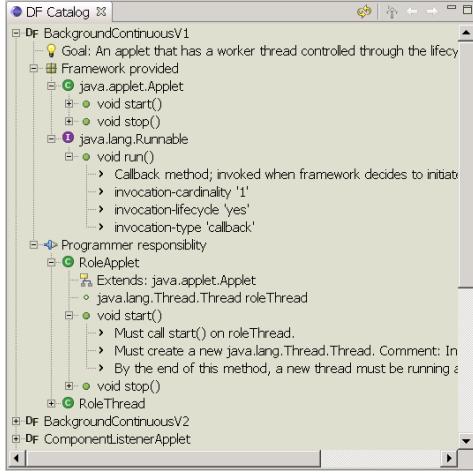


Figure 5.2: Catalog View

representing solutions, encoding design intent, and connecting with external files. Each of these difficulties is addressed in the following sections, with a description of how the tools help overcome the difficulty. We conclude with a set of use cases that the tool supports, organized around the software development lifecycle.

5.1 Understand Non-Local Client-Framework Interactions

Accomplishing a single goal usually requires the coordination of many client-framework interactions, such as calls to framework service methods, overriding of framework callback methods, and creation of new objects derived from framework classes. These client-framework interactions cannot be co-located in the source code, which causes difficulty for client programmers both when creating new source code and when evolving existing source code.

A client programmer creating new source code may have a clear idea what he wants to do, which we call his goal, but may not know how to accomplish it. Design fragments represent known-good ways to accomplish goals, so they help the client programmer by providing a means to move from goal to solution. The design fragment tools present a catalog of design fragments in the **Catalog View**.

Figure 5.2 shows the design fragments Catalog View, showing design fragments for the Java applets framework. The Background Continuous Task V1 design fragment has been opened, showing the text of its goal, the parts provided by the framework, and the parts the programmer must build. The definition of the `run()` callback method has been opened, showing the specifications of when and how often this callback will be invoked by the framework.

The section on programmer responsibility shows the two class roles, `RoleApplet` and `RoleThread`. The class role `RoleApplet` must be a subclass of `java.applet.Applet`, must have a field named `roleThread` of type `java.lang.Thread`, and must implement the two callback methods `start()` and `stop()`. The `start()` callback method has been opened, showing the specifications of what the programmer is expected to do in order to fulfill framework

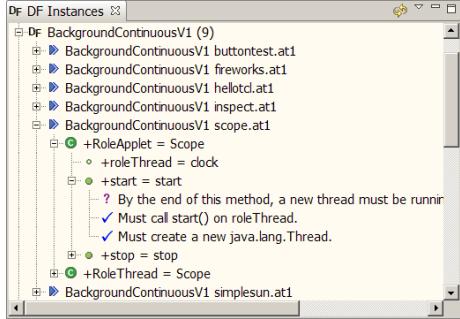


Figure 5.3: Instances View

obligations and this design fragment.

A client programmer using the Catalog View could scan through the design fragments and find one that matched his goal. The Catalog View describes the relevant framework classes and interfaces he needs to know about, and describes what his source code must do, which helps him move from his goal to a solution.

While the Catalog View shows only design fragments, the **Instances View** shows design fragments and their bindings to source code. Using the Instances View, the client programmer can see how other programmers have used this design fragment. Seeing the design fragment used in context should help him better understand how it is used, and comments in the source code may yield additional insights.

Each use of a design fragment, which we call a design fragment instance, is given a name. Usually a design fragment is used just once per class or package, but not always. For example, we analyzed a two-player Tetris game that used two background threads, so it used two instances of the Background Continuous Task V1 design fragment. The Instances View shows the design fragments in a tree with each instance as a child of its parent design fragment.

Figure 5.3 shows the Instances View with some of the nine instances of the Background Continuous Task V1 design fragment, the design fragment that was shown earlier in the Catalog View. This view shows where each role in the design fragment is bound in the Java source code. For example, the `scope.at1` instance has the `RoleApplet` and `RoleThread` roles bound to the `Scope` Java class, and the `roleThread` role bound to the `clock` field. The client programmer can navigate to the relevant parts of the source code itself by double clicking on these role bindings.

The Instances View will indicate binding problems, such as when a class role is not bound to any Java class. Any conformance analysis failures are also shown in this view. In the figure, the specifications for the `start()` method have been opened, revealing checkmarks indicating successful conformance. The user interface distinguishes three states: pass (a checkmark), fail (an X sign), or no analysis tool is available for this specification (a question mark).

So far we have discussed how a client programmer who is writing new source code would use the design fragment tools. We now switch our discussion to a client programmer who is reading some existing source code, trying to understand how it uses the framework. This programmer would see the design fragment bindings in the source code, and could navigate from the source code back to the Instance View and Catalog View. This link back to the design fragment reveals, through the stated goal of the design fragment, the original author's intention in writing the code.

Status	Obligation	DF	Instance
✓ Ok	Class must be referenced in xml file at role: RoleViewExt in attribute: class	TreeViewInitialization	treeviewinit1
? Unknown	Must create a new org.eclipse.jface.viewers.TreeViewer passing in parent.	TreeViewInitialization	treeviewinit1
? Unknown	Required callback method to initialize the plugin	TreeViewInitialization	treeviewinit1
? Unknown	Conventionally this is the createPartControl method	ToolBarAction	toolbaraction1
✓ Ok	Must call roleMakeActions() on this.	ToolBarAction	toolbaraction1
✓ Ok	Must call roleAddActionToToolbar() on this.	ToolBarAction	toolbaraction1
? Unknown	Must create a new RoleViewerSorter.	ViewSorter	viewsorter1
✗ Fail	Must call setSorter() on roleViewer passing in a new RoleViewerSorter.	ViewSorter	viewsorter1
? Unknown	Required callback method to initialize the plugin	ViewSorter	viewsorter1
? Unknown	Must create a new RoleLabelProvider.	LabelProvider	labelprovider1
✓ Ok	Must call setLabelProvider() on roleViewer passing in new RoleLabelProvider.	LabelProvider	labelprovider1
? Unknown	Required callback method to initialize the plugin	LabelProvider	labelprovider1
? Unknown	Must create a new RoleContentProvider.	ContentProvider	contentprovider1
✓ Ok	Must call setContentProvider() on roleViewer passing in the new RoleContentProvider.	ContentProvider	contentprovider1
? Unknown	Required callback method to initialize the plugin	ContentProvider	contentprovider1
✓ Ok	Must call roleHookRightClickAction() on this.	RightClickMenu	rightclickmenu1

Figure 5.4: Context View

It also reveals the non-local client-framework interactions, helping him understand how this part of the program connects to other parts of the program that he may not have found yet.

The Catalog and Instance Views display how single design fragments work, but a client programmer will also need to understand the intersecting demands of many design fragments. The **Context View** displays a list of the relevant specifications for what the programmer is editing. A programmer editing the `createPartControl` method, shown in Figure 5.4, would see a large number of design fragments intersecting, all of them contributing specifications. This intersection, or tangling, happens because multiple design fragments may provide specifications for the same callback method. In our experience, the busiest callbacks are the lifecycle callbacks that mark the initial creation or imminent destruction of an object. As can be seen in the figure, six different design fragments are contributing specifications in this example.

For each specification, the Context View shows the specification, its originating design fragment and instance, and the conformance status of the source code. At a glance, the client programmer can see all of the obligations this source code must satisfy, and whether those obligations have been satisfied.

Because a client program is filled not only with the client-framework interactions but also with its domain logic, it can be helpful to elide part of the program, and focus just on the implementation of a single design fragment. A client programmer trying to debug the Background Continuous Task V1 design fragment would benefit from seeing just the classes, fields, and methods in the client program that are bound to that design fragment. This is analogous to seeing an *aspect* [40] of the client program, or a *slice* [68] through it.

Figure 5.5 shows the **Slice View** displaying an Applet. The user has selected the `scope.at1` instance of the Background Continuous V1 design fragment. The view shows the two class roles from the design fragment, `RoleApplet` and `RoleThread`, which in the source code are both bound to the same class, `Scope`. `RoleApplet` has three roles: `roleThread`, `start`, and `stop` (by convention, the programmer is free to change the name of roles beginning with “role” but should keep the same name otherwise). `RoleThread` has one role: `run`. This view has elided all other classes, fields, and methods in the project, limiting the client programmer’s view to the parts that are relevant for this design fragment. It removes non-locality by combining source code from every class, field, and method into a single view.

In order to compute the view shown in the figure, the tool uses the design fragment instance that the user has selected in the Instance View, which in this case is the `scope.at1` instance.

```

DF DF Slice x
Design Fragment instance: scope.at1 (BackgroundContinuousV1)

Class role: RoleApplet bound to implementation class: Scope
Thread clock

void start(?) {
    if (clock == null) {
        if (stoppedAt != 0) lostTime+=System.currentTimeMillis() - stoppedAt;
        clock=new Thread(this);
        clock.start();
    }
}

void stop(?) {
    if (clock != null) {
        stoppedAt=System.currentTimeMillis();
        clock=null;
    }
}

Class role: RoleThread bound to implementation class: Scope
void run(?) {
    while (clock != null) {
        try {
            clock.sleep(100);
        }
        catch (Exception e) {
        }
        repaint();
    }
}

```

Figure 5.5: Slice View

The design fragments tool keeps data structures tracking the binding between that instance and the source code, so it is possible to extract just the source code corresponding to the bound roles. Note that, in our tool, entire methods are shown and no attempt is made to eliminate irrelevant statements within the method. Also, our tool has not extracted method parameters from the AST so they are shown as question marks.

5.2 Design Solutions without Owning the Architecture

A client programmer who designs a solution may find that it conflicts with the framework author's design of the framework. The framework author has envisioned how the framework will be used, and has provided service methods and callbacks to enable that use. The framework author's design choices decide how constrained the client programmer's design space will be.

Since design fragments represent known-good solutions, a client programmer who uses one of them need not worry about finding a solution that is compatible with the framework, since all of the design fragments are compatible. The design fragment Catalog View displays the list of design fragments, and, by scanning through that list, the client programmer can select one to use. The existence of the design fragment catalog helps the client programmer by taking away his old job, which was searching through a solution space where poorly understood framework constraints often prevent him from using his standard bag of programming tricks, and gives him a new job, which is scanning linearly through the design fragment catalog to find a match.

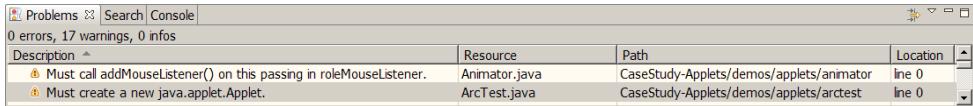


Figure 5.6: Problems View

5.3 Gain Confidence of Compliance

Once the client programmer has written the source code and bound it to a design fragment, he would like to be sure that his code correctly does what the design fragment says it should. Our analysis tools check conformance; their findings are visible to the client programmer in two views we have already discussed: the Instances View and the Context View. Findings are also visible in the **Eclipse Problems View**, a standard view in the Eclipse IDE that collects all problems found in the IDE.

We have augmented the standard Problems View to display the problems that are found by our analysis tools. As seen in Figure 5.6, when source code fails to meet the specifications defined in the design fragment, the problems are reported as warnings in the Problems View. Clicking on the problem will navigate the programmer to the line in the source code where the problem was detected.

Most problems are reported as warnings, including conformance analysis failures and incomplete bindings. A few problems are reported as errors, including the declaration of design fragment instances where the design fragment is not found in the catalog. Arguably every problem could have been reported either as an error or as a warning. However, since programmers can evolve a program such that it no longer conforms to a design fragment, yet correctly interacts with the framework, a warning seems more appropriate than an error.

In addition to appearing in the Problems View, errors are also presented in the source code editor. Figure 5.7 shows the `createPartControl` method and line 315 has been commented out. Consequently, a design fragment conformance error has been noted with a warning sign at line 305 where the design fragment is bound. Placing the mouse cursor on this warning sign reveals the text of the warning message.

The Eclipse IDE includes convenient features like incremental compilation upon the saving of source files so that the list of problems is always accurate. Programmers have grown accustomed to this style of interaction with their tools, so the design fragments tools are built to work in a similar way. Changes to the source files that define design fragments cause those files to be re-parsed and presented in the Catalog View. Any changes to the catalog propagate to the design fragment Instances View. Similarly, changes to the Java source code trigger re-analysis of the bindings to design fragments and these are displayed in the Instances View. Any problems detected during these steps are reflected in the Problems View.

Our initial implementation of the specification checkers analyzed the full project with every change, but, as the size of the case studies grew, the time required to re-analyze grew larger than a minute. We re-implemented the analysis to run in the background and analyze just what is needed to recalculate the specifications, not the entire project. The analysis runs in a separate thread so that the programmer does not wait; the design fragment views update within a few seconds. Our tools check specifications in an incremental, background style that is consistent with the way

The screenshot shows an Eclipse IDE window with a Java file named 'SampleView.java' open. The code is as follows:

```

302     @df.bindings({
303         @df.binding(inst="treeviewinit1", role="createPartControl"),
304         @df.binding(inst="toolbaraction1", role="roleActionSetup"),
305         Spec fails: Must call setSorter() on roleViewer passing in a new RoleViewerSorter.
306         @df.binding(inst="labelprovider1", role="createPartControl"),
307         @df.binding(inst="contentprovider1", role="createPartControl"),
308         @df.binding(inst="rightclickmenu1", role="createPartControl"),
309     })
310     public void createPartControl(Composite parent) {
311         viewer = new TreeViewer(parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
312         drillDownAdapter = new DrillDownAdapter(viewer);
313         viewer.setContentProvider(new ViewContentProvider());
314         viewer.setLabelProvider(new ViewLabelProvider());
315         // viewer.setSorter(new NameSorter());
316         viewer.setInput(getViewSite());
317         makeActions();
318         hookContextMenu();
319         hookDoubleClickAction();
320         contributeToActionBar();
321     }

```

A tooltip 'Spec fails: Must call setSorter() on roleViewer passing in a new RoleViewerSorter.' is displayed over the line 'viewer.setSorter(new NameSorter());'. The code is written in Java, and the IDE interface is visible around the code editor.

Figure 5.7: Errors appearing in editor

that current IDE’s check type and syntax errors, and this ability contributes to design fragments providing pragmatic help to client programmers.

All of the analyses use the Eclipse Abstract Syntax Tree (AST). By default, AST’s from Eclipse do not resolve bindings of names, which speeds up AST creation by a factor of 3. For example, from examining a .java file we can see that a class named B is defined but we may not yet know if it is a subclass of A until names are bound. In order to correctly implement some of the analyses we need to have the names resolved. The current engineering compromise is to leave the analyses turned off that require the name binding, and only occasionally turn them on. Other options include using a faster AST implementation, or formalizing the quick-versus-accurate modes of checking into the tool, rather than asking the user to toggle analyses manually.

5.4 Represent Solutions

Client programmers have difficulty representing their known-good solutions, and without design fragments would likely resort to HowTo articles or example code to communicate their knowledge with others. We propose that a better way to express this knowledge is to write a design fragment.

The design fragments tools represent a design fragment as an XML file with simple syntax. Once a client programmer has written a design fragment in a file, it is parsed and presented in the design fragment Catalog View. Any parsing errors are displayed in the Problems View.

Our tool’s implementation of the design fragment catalog makes it easy to share design fragments and to keep the catalog updated. The design fragment catalog is represented as an Eclipse project and the files in the project are the XML design fragment definitions. Eclipse can synchronize projects with source code control repositories, like CVS or Subversion, so programmers can stay updated with the latest design fragment catalog on the internet. Following the Debian [59] example of maintaining stable, testing, and unstable builds of their Linux distribution, each design fragment catalog has folders for stable, testing, and unstable design fragments. It is expected that most programmers would use the stable design fragments, which have been vetted by the

catalog maintainer, but programmers on the cutting edge could use the testing or unstable folders. Each catalog contains design fragments for just one version of a framework.

5.5 Encode Design Intent

Once a client programmer has learned enough about the framework and has implemented all the non-local client-framework interactions necessary, he may become concerned that other programmers will not understand his design intent. If he binds a design fragment to his code, however, the annotations he places in his code preserve his design intent.

```
...
@df.bindings({
    @df.binding(inst="scl1", role="RoleView")
})
public class SampleView extends ViewPart
...
}
```

The above example shows an annotation on a Java class. With the design fragments tools, this annotation connects the source code back to the design fragment definition, and from there to the related places in the source code.

5.6 Connect with External File

Since modern frameworks often require both object-oriented source code as well as external configuration files, client programmers need to coordinate their work across both. A design fragment can specify that parts of the source code are correlated to parts of an XML configuration file. When a client programmer is using a design fragment for the first time, the tools will guide him to implement both the object-oriented code as well as the external configuration file. Furthermore, during evolution of the source code, the tools will warn him when the name of a class in his source code changes without a corresponding change to the configuration file, and vice versa.

5.7 Use Cases

The design fragment tools were built to overcome the difficulties we identified with frameworks. The tools help programmers to cope with the difficulties by enabling specific use cases. Use cases are at a more detailed level than the difficulties, and more easily traced to the specific tools that we have built. We have organized these use cases by the phase of software development where they would be most helpful.

1. Comprehension and evolution of existing client code

- Understand the overall client use of the framework by reading the declared design fragment instances

- Disentangle code in method by understanding what goal(s) each interaction (e.g., a method call) supports
- Determine how a particular interaction contributes to the overall intention of the client program by tracing it to the design fragments that require it
- Learn a method's obligations to the framework
- Navigate from a particular mechanism to its goal, and from there to related mechanisms

2. Generating new client code

- Search for design fragment matching known goal
- Document design intent for the benefit of other programmers by declaring design fragment instances and bindings
- Check that specified obligations within a class or method have been met

3. Debugging existing client code

- Navigate from a particular mechanism to its goal, and from there to related mechanisms
- Focus attention on parts of code contributing to a single goal
- Detect when code evolution disrupts existing code

4. Learning about the framework

- Learn what framework can do by browsing the catalog of design fragments
- Navigate from goal to relevant framework API documentation via a design fragment

These use cases are directly supported by our tools today, but the underlying data structures enable the support of even more use cases, including wizards to write the bindings and additional visualizations of the design fragments and constraints. We discuss these possibilities in Section 9.5.

5.8 Summary

The design fragments tools are built into the Eclipse integrated development environment. Several views are visible to the user including a view of the design fragment catalog, a view of the design fragment instances, a view of the design fragment specification problems, a view of the specifications relevant to the currently edited method, and a view showing a slice through the program only revealing those classes, fields, and methods that pertain to a given design fragment. Analysis tools run in the background and incrementally check conformance between the specification and the source code.

These views and analysis tools enable design fragments to directly help programmers address the six difficulties we identified with frameworks. Help is available to client programmers who are beginning to write code. Since the views present design fragments that are known-good solutions, the solutions will conform to the framework architecture and other framework

constraints. Help is also available to programmers trying to understand or evolve existing code, since the bindings to design fragments reveal the intention of the client programmer, and allow easy navigation to other relevant parts of the source code.

In the future work section of the conclusion chapter, Section 9.5, we discuss some ways we would like to see the design fragment tools extended, specifically to reduce the client programmer's effort and to increase his understanding of how the source code works.

Chapter 6

Case Study: Applet Framework

To demonstrate that our technique is pragmatic, we must apply it to real frameworks. Early stage research such as this must be judicious in their investigations, so we have focused our attention on two case studies. The first is a thorough investigation of a small framework, and the second is a targeted investigation of a large framework. The two case studies are complimentary in that the first shows our technique working on a complete but small example, and the second demonstrates that the technique will work when the framework size is large.

This chapter presents the first case study, an investigation of the Applet framework [63]. We progressed through the complete use of our technique, from initial catalog creation to its conformance checking on client programs. We built a complete catalog consisting of twelve design fragments that describes the interactions between fifty-six Applets and the Applet framework, indicating that Applets were consistent in their use of the framework. The growth rate of our catalog was asymptotic, which supports the conclusion that our technique is pragmatic. We also discovered examples where programmers failed to comply with the framework, indicating that our tooling to check conformance would be helpful.

The following sections describe the Applet framework, our procedure, results, and discussion.

6.1 Applet Framework

The Applet framework allows Java code to run inside of a web browser. The framework presents the Applet with a canvas to draw on, and makes events like mouse clicks available to the Applet. The author of a web page can embed an Applet using an HTML tag like this, optionally passing in parameters:

```
<applet code=MyApplet.class param=5>
```

Sun has bundled demonstration Applets with the Java Development Kit (JDK) since its original version. The JDK today contains twenty demo Applets, and thousands more can be found on the internet with a simple search.

The Applet framework is small, yet it is built from the same object-oriented mechanics as other frameworks. Client programmers must understand, for example, the framework constraints,

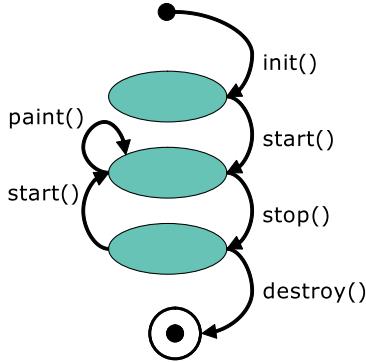


Figure 6.1: Applet callbacks

the framework callback methods, the framework service methods, and how non-local client-framework interactions can achieve a goal.

The Applet framework defines several lifecycle callback methods that are invoked on the programmer's Applet when the user starts the Applet in the web browser. As shown in Figure 6.1, the Applet first receives an `init` callback, then at least one `start` and `stop` pair of callbacks, then a single `destroy` callback. After receiving a `start` callback, it may receive `paint` callbacks until it receives the `stop` callback.

In addition to these callback methods, the Applet framework also defines service methods that the client program can invoke on the framework. Many of these service methods are for setting up additional callbacks that communicate events, such as `addKeyListener` and `removeKeyListener`. After calling `addKeyListener`, the client program will receive additional callbacks from the framework, `keyPressed`, `keyReleased`, and `keyTyped`, informing the Applet of these events as they occur.

The Applet framework is closely related to the Java Abstract Widget Toolkit (AWT). Any AWT program can become an Applet by subclassing from `java.applet.Applet`. As we discuss in Section 6.5.3, this complicated our procedure for finding Applets on the internet.

6.2 Hypotheses

This case study was designed to demonstrate that the variety of design fragments was low. We expected to find several design fragments by examining the first few Applets, and, as more Applets were examined, we expected to find fewer and fewer new design fragments. Ideal evidence of low design fragment variety would be a small catalog whose size grew asymptotically.

We also hoped to find evidence that client programmers were following an examples strategy. Therefore, the first Applets we examined were demo Applets, Applets that other client programmers may have examined as examples.

The following section describes our case study process.

6.3 Process

Starting with an empty catalog of Applet design fragments, we examined the demo Applets. As we recognized repeated patterns of client-framework interaction, we created new design fragments, bound them to the source code, and added them to our catalog. We then identified Applets from the internet to be examined. We attempted to bind our existing design fragments to the internet Applets, and added new design fragments to our catalog when needed. Finally, we used internet searches to collect aggregate demographic information on internet applets [14]. The subsequent sections describe this process in more detail.

6.3.1 Demo Applets

Since we initially had no design fragments for the Applet framework, we needed a source of example Applets. We selected the twenty demo Applets provided by Sun in the Java Development Kit. We expected that these Applets, like other example client programs, were provided to show what the framework could do and so were likely to exercise many parts of the framework.

Starting with the first demo Applet, we examined the source code looking for uses of framework callback and service methods. When we recognized repeated patterns we documented them as a design fragment.

At the end of examining the twenty demo Applets, we had created a catalog of design fragments for the Applet framework. Each Applet declared a set of design fragment instances, and their roles were bound to the source code. The constraint checking tool reported no errors of conformance except those resulting from limitations in the tool itself. An example of such a limitation is that the tool could not check if required method calls happened in subroutines.

6.3.2 Collecting Internet Applets

To evaluate our existing catalog of Applet design fragments with a set of Applets from the internet, we first had to identify these Applets. We searched the internet for Applets using this search string in Google:

```
filetype:java "import java.applet.Applet" -site:sun.com
```

However, as we discuss in Section 6.5.3, this search yielded mostly AWT programs that did not meaningfully engage in the Applet framework, meaning that they did not use any of the Applet framework callback methods or service methods.

The `filetype:java` part of the search string constrains the results to files ending with `.java`. Words in quotation marks must appear in sequence in the results. Words with a dash in front of them, like `-site:sun.com`, are negated. The words `site:sun.com` part restricts the search to the `sun.com` domain, but since it is prefixed with a dash, it returns results not from `sun.com`.

We augmented our search string to select Applets that implemented Applet callbacks or invoked Applet-specific service methods on the framework. For example, to find threaded Applets, we added `java.lang.Thread` to the search string; to find mouse listener Applets, we

added `java.awtMouseListener`; to find parametrized Applets, we added `getParameter`. We collected the first ten Applets that matched each search string, and, with duplicates from our searches eliminated, collected thirty-six Applets in total.

6.3.3 Apply Catalog to Internet Applets

To apply our catalog to the internet Applets, we proceeded similarly as with the demo Applets, except that we used the catalog of design fragments already created. When possible, we identified and bound existing design fragments to the source code of the internet Applets. We added new design fragments to the catalog as needed.

6.3.4 Collect Demographic Information

In addition to the detailed examination of individual Applets, we collected aggregate demographic information about Applets on the internet. We used the Google search engine to find Applets, as before, and we recorded the number of search results that matched our query. We attempted to exclude Applets written by Sun from our search, as these overlap with the demo Applets. Our basic query to find non-Sun Applets was:

```
filetype:java "import java.applet.Applet" -"Sun Microsystems, Inc.  
All Rights Reserved." -site:sun.com.
```

Queries appended additional search terms to this string. For example, to find Applets that override the `init` callback, we appended “`init`”.

6.4 Results

Here we present the results of our investigation of demo and internet Applets. The design fragments in our catalog can be divided into categories: threaded Applets, event handling Applets, and other Applets. We have organized our results to describe each category and the design fragments within it. We conclude our results with the aggregate demographic information.

From the twenty demo Applets we found ten design fragments. The set of demo Applets contained two design fragments not found in the internet Applets: One-time Init Task and the Timed Task. The set of internet Applets contained two design fragments not found in the demo Applets: Background Continuous V1 and Focus Listener. All other design fragments were found in both.

We added two new design fragments from our examination of the internet Applets, yielding a catalog of twelve design fragments. We added the Background Continuous V1 design fragment, a variant of an existing design fragment. These two design fragment variants have the same goal, and are very similar in structure, differing only in a single check that occurs in the `run` method. V1 checks that the `thread` field is not null, while V2 checks that the `thread` field is equal to the currently running thread. The other added design fragment, Focus Listener Applet, is structurally similar to the other listener design fragments except that it listens for user interface focus changes.

Design Fragment Name	Description	Instances from demos	Instances from internet
<i>Threading</i>			
Background Continuous v1	A separate thread used to execute an ongoing task	0	9
Background Continuous v2	Same as above, but with a race condition removed	6	3
One-time Init Task	A separate thread used to run a task at startup, once	2	0
One-time On-Demand Task	A separate thread used to run a task at a domain-specific time, once	1	3
Timed Task	A task that should be repeated regularly	1	0
<i>Event Handling</i>			
Component Listener	Listening for component events	1	1
Focus Listener	Listening for when the Applet gets focus in the GUI	0	1
Key Listener	Listening for keyboard events	1	2
Mouse Listener	Listening for simple mouse events	10	12
Mouse Motion Listener	Listening for complex mouse events	4	11
<i>Other</i>			
Parametrized Applet	An Applet that reads parameters from a web page	13	17
Manual Applet	An Applet that can be run from the command line because its main method manually invokes the Applet lifecycle methods	5	5

Table 6.1: Design fragment frequency

A tabulation of the design fragments, including a short description and how often they occurred, is shown in Table 6.1. The first column lists the design fragments by category and name. A short description of the design fragment is in the second column, and the number of times the design fragment was found in the demo and internet Applets are in the third and fourth columns.

Table 6.5 at the end of this chapter is a compilation of all of the Applets analyzed and the design fragments that were found within them. The following sections describe each of the identified design fragments in detail.

6.4.1 Threaded Design Fragments

There are five design fragments that we have categorized as related to threading: Background Continuous V1 and V2, One-Time Init Task, One-Time On-Demand Task, and Timed Task. These design fragments have the goal of coordinating a programmer-created thread with the

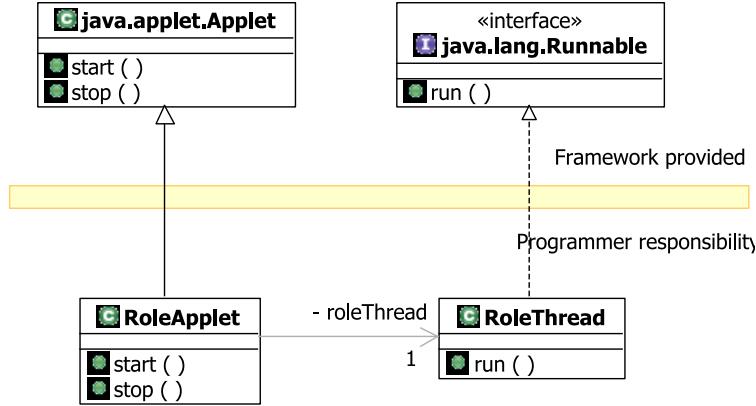


Figure 6.2: Background Continuous V1 and V2 structure

framework callback methods.

The Applet framework never explicitly requires programmers to create new threads, yet our searches reveal that 27% of Applets on the internet do. The Applet framework constrains the solution space for programmers and, in response, programmers have solved their problem using threads. The API documentation for the Applet framework does not instruct or suggest that the client programmer should use threads.

Figure 6.2 shows the structure of the Background Continuous V1 and V2 design fragments. The threads in these design fragments are intended to run for a long time, usually the duration of the Applet.

The One-Time Init Task design fragment uses a thread to perform a time-consuming startup task, such as establishing a connection with a server. The task is done in a background thread so as to keep the GUI responsive. This task is started in the `init` callback method. The One-Time On-Demand Task generalizes the One-Time Init Task. In it, the background task can be started at any time during the running of the Applet, such as when the user presses a key.

The Timed Task design fragment was only found in the demo Applets, but could have been applied in many places where Background Continuous was used. Timed Task uses a Java Timer instead of a thread and the Timer can be set to run every so many milliseconds. Many uses of the Background Continuous design fragments could have been written more simply and with less risk of a race condition if they had used the Timed Task instead, but none of the Applets we examined from the internet used this design fragment.

6.4.2 Event Handling Design Fragments

As we described in our description of the Applet framework in Section 6.1, the Applet framework allows client programs to request to be notified via a callback method of additional events from the framework. The design fragments that we have categorized as related to event handling all follow the structure shown in Figure 6.3, which shows the Mouse Listener design fragment.

To receive these additional callbacks, the client program must implement the appropriate listener interface, in this case `MouseListener`, and provide implementations for each of the required callback methods defined in that interface. In the `init` lifecycle callback method of the

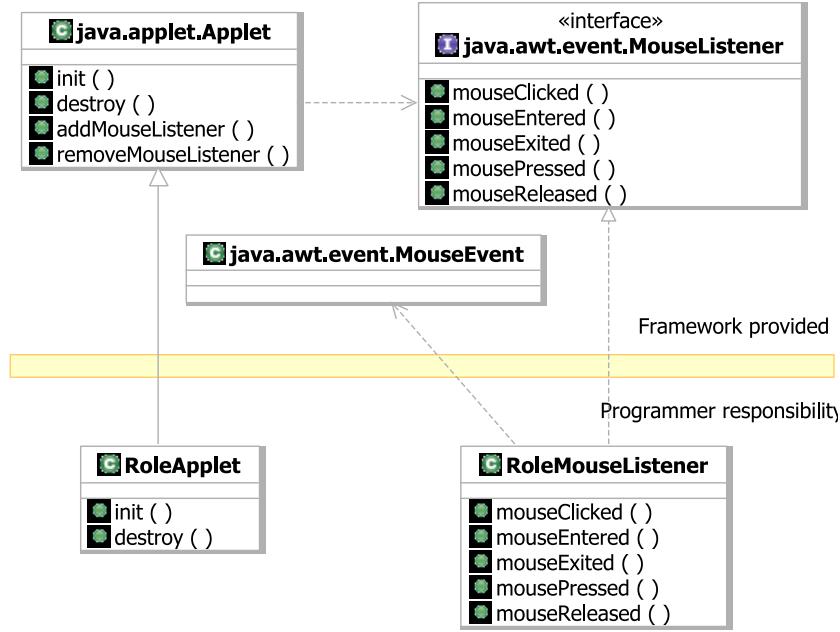


Figure 6.3: Mouse event handling structure

Applet, the client program must call a framework service method to register for callback events of this type, in this case `addMouseListener`. The corresponding `removeMouseListener` service method must be called from the `destroy` lifecycle callback method to de-register for events. As we discuss in Section 6.5.2, many of the Applets from the internet did not de-register for events, while the demo Applets generally did de-register.

6.4.3 Parameterized Applet Design Fragment

Parameterized Applet deals with how to obtain the textual parameters that can be passed into an Applet, and how to report to users what parameters can be passed in. Java Applets are often run from web pages and it is possible to pass parameters into the Applet via the HTML text, like:

```
<applet code=ArcTest.class width=400 height=400>
```

In this case, the parameters `width` and `height` are passed in as strings with values of 400. The Applet can read these parameters with the framework service method `getParameter(String parameterName)`. An Applet should let its users know what parameters they can pass in, and this is done with the non-lifecycle callback method `getParameterInfo`, which returns an array containing the parameters and their expected types. The structure of this design fragment is shown in Figure 6.4.

Ideally, every Applet that reads parameters would publish the fact that it reads them. Furthermore, the published list should match exactly the calls to `getParameter` made by the Applet. In practice, more than half of the Applets from the internet had a mismatch between the parameters they queried and the parameters they published. We speculate that it is easy for a client programmer to add a parameter check, but forget to update the list, breaking this non-local constraint.

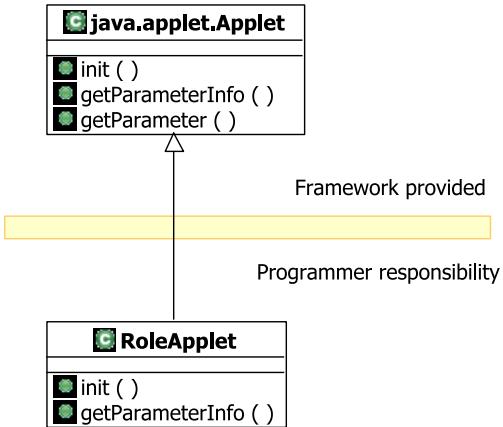


Figure 6.4: Parameter handling structure

We had trouble expressing the constraints for this design fragment using our language. The `getParameterInfo` method returns an array of arrays, where each row has information on a single parameter. The constraint is that every call to `getParameter(String parameterName)` that is made in the Applet must have a matching row for `parameterName`. Conversely, no row exists for a parameter that is not queried. This is difficult for the design fragment language because the contents of the array are not static. As we discuss in Appendix A, we could create a new design fragment specification and push the complexity of checking conformance into a difficult-to-write analysis routine.

In Section 9.5, we discuss how framework authors should choose constraints that are easier to check, now that design fragments are available. For example, the framework constraint could insist that both `getParameter` and `getParameterInfo` only refer to static final fields, which would make the conformance check easy.

6.4.4 Manual Applet Design Fragment

Manual Applet deals with how to provide a `main` method that simulates the callback structure of an Applet, so that the Applet can be invoked from the command line in addition to running with a web browser. In Java programs, the `main` method is the first one executed when the program starts. In order to execute the Applet by simulating the framework callbacks, calls to the Applet's `init` and `start` methods are invoked by the client program from its `main` method. The `main` method must also create a `java.awt.Frame` for the Applet to be run in. The structure of this design fragment is shown in Figure 6.5.

6.4.5 Demographics

We searched the internet for Applets using our basic search string to find non-Sun Applets:

```

filetype:java "import java.applet.Applet" -"Sun Microsystems, Inc.
All Rights Reserved." -site:sun.com.

```

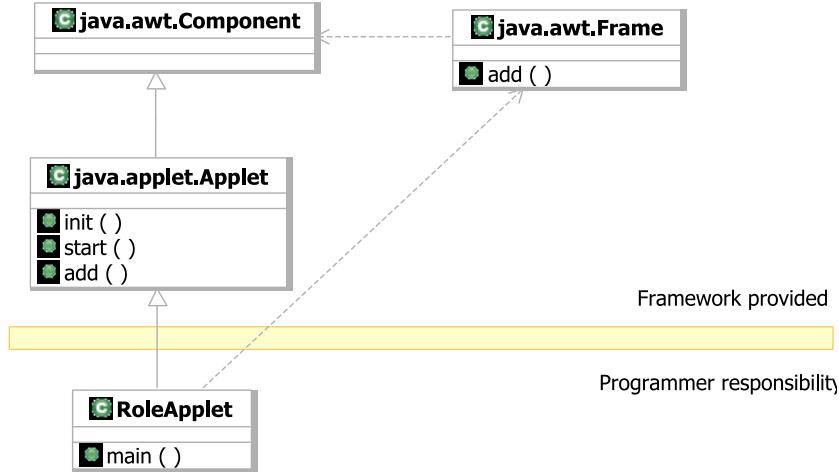


Figure 6.5: Manual Applet structure

This search yields 52,700 results, where each result is a Java file, so we interpret it to mean 52,700 Applets. All of our searches were conducted using the Google search engine in February of 2006.

From this set of Applets, we further queried to find out how often certain strings appeared, such as the names of the Applet framework callback methods or service methods. We did this by appending additional strings to the basic query, and the results are shown in Table 6.2.

This aggregate demographic information provides evidence that not all Applets conform to the framework constraints. For example, only 510 of the 14,600 Applets that use threads (4%) call `currentThread`, meaning that they are candidates for Background Continuous V1, the buggy one, but not Background Continuous V2, the correct one that requires a call to `currentThread`. Similarly, we see many calls to register for event callbacks, but few calls to de-register. We discuss this non-conformance in more detail in Section 6.5.2.

6.5 Discussion

A surprising outcome of this case study was the discovery of the widely reproduced threading bug, which we represented as a design fragment named Background Continuous V1. Applets that conformed to that design fragment had a threading bug, but the design fragment did respect the framework constraints. Not so other Applets, which for example, violated the framework constraints regarding event registration and de-registration.

We will also discuss a possible selection bias in our sampling of internet Applets, and the likelihood that other authors would create slightly different design fragments. We conclude our discussion with an examination of our hypotheses: that the variety of design fragments is small, and that client programmers are following the examples strategy.

	Internet count
All Applets	52,700
with paint	33,400
with start	21,400
with stop	15,300
... with both	14,700
with init	41,600
with destroy	597
... with both	579
with getParameter	9,440
with getParameterInfo	221
... with both	180
with MouseListener	837
with MouseMotionListener	448
with KeyListener	385
with ComponentListener	108
with “void main”	566
with Thread	14,600
... also with “currentThread”	510
with Timer	851
with addComponentListener	118
with removeComponentListener	1
... with both	1
with addMouseListener	573
with removeMouseListener	49
... with both	48

Table 6.2: Internet Applet demographics

6.5.1 Threading Bug

As was described earlier, we identified two design fragments that have identical goals, but differ in regards to a threading bug in one of them: the Background Continuous V1 (buggy) and V2 (correct) design fragments. The original Java Development Kit (JDK) contained a demo Applet called `NervousText`, shown in Figure 6.6, that repeatedly painted text onto the screen, offsetting it each time by a pixel or two, which had the effect of making the text look “nervous.” Unfortunately this very simple demo Applet also contained a bug in how it coordinated the Applet callbacks with thread creation and termination.

```
...
public class NervousText
    extends java.applet.Applet
    implements Runnable {
...
Thread killme = null;
...
public void start() {
    if (killme == null) {
        killme = new Thread(this);
        killme.start();
    }
}
public void stop() {
    killme = null;
}
public void run() {
    while (killme != null) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {}
        repaint();
    }
    killme = null;
}
...
}
```

Figure 6.6: `NervousText` JDK 1.0 Applet

The programmer’s intent during the `stop` callback is to signal to the running thread that it should terminate. The signal is that the `killme` field is set to `null`. The race condition occurs when the framework invokes both `start` and `stop` before the thread checks the value of the `killme` field. When that happens, the old thread continues executing, since it missed the signal to terminate, and a new thread executes too, since it was started in the second call to `start`. The bug was fixed in the next release of the JDK, and the check for `killme != null` was replaced with the more precise `killme == Thread.currentThread()`.

Note that while there was indeed a race condition in `NervousText` that could yield two or

more active threads, there was no user-perceived error. Since the original Applet just wiggled text on the screen, having multiple threads meant it wiggled a bit faster. When this buggy pattern is reused elsewhere, however, the extra thread may be perceived by the user, for example as a double-fast opponent in a video game.

It is tempting to use the Java mechanism of deprecating individual methods to solve this problem. Deprecation can be successfully used to inform client programmers that a method in an API should no longer be used. Deprecation does not work in this example because the error does not lie in any framework method, but instead in the pattern of the client programmer's use of those framework methods. Specifically, `NervousText` implements the `start`, `stop` and `run` callbacks and does not violate the contract for any of them.

Concurrency is notoriously difficult, so it is tempting to dismiss this bug as just another threading bug. However, it is possible that this bug resulted from a programmer's misunderstanding the framework callback sequence. The JDK 1.0 method documentation for `start` reads:

This method is 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 (II-\$3.1.22) to suspend the animation.

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

See Also: `init` (II-\$3.1.13) `destroy` (II-\$3.1.2).

Figure 6.7 shows both the correct callback state machine and a simplified, but incorrect, version. Such a state machine was not delivered with the Applet framework but it can be deduced through careful reading of the documentation. The way it can be deduced is by recognizing that “each time the Applet is revisited in a Web page” is not clear, therefore the safe assumption is that the Applet might get lots of `start/stop` calls. A quick reading can yield the incorrect reasoning that “I do not cause my Applet to be revisited (whatever that means), so I will just get one `start` and `stop` callback”. Indeed, normal testing of the Applet may never yield multiple `start/stop` callbacks since the user must do something like minimize the browser while the Applet is running to trigger an additional pair of `start/stop` callbacks. While it cannot be determined at this point how the bug in `NervousText` originated, it is consistent with both the hypothesis that “threading is hard” and with the hypothesis that “framework callback sequences are hard to understand”.

This bug, once introduced in the demo Applets provided with JDK 1.0, has spread widely. The revised JDK containing the corrected Applet was delivered in 1997, yet collecting Applets from the internet today reveals more Applets with the bug than without. In our examination of thirty-six Applets from the internet, we found that just 3 of the 12 Applets used the corrected

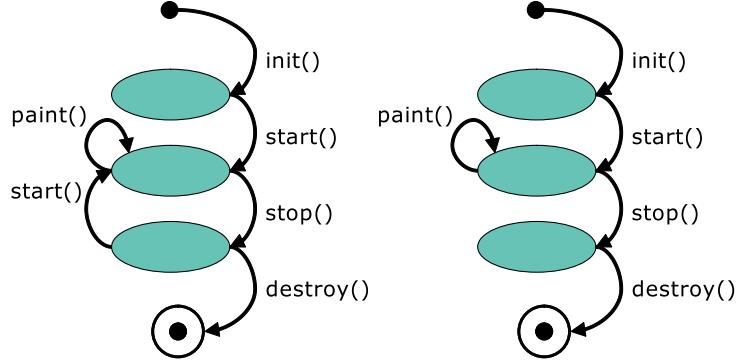


Figure 6.7: Applet callbacks - correct (left) and incorrect (right)

version of the design fragment, as seen in Table 6.1. Fixing the bug requires testing to see “Am I the thread that should be running?”, which requires a call to `currentThread`. An internet search, as shown in Table 6.2, reveals that just 510 of 14,600 Applets that use threads also call `currentThread`. Consequently, our finding that 75% of the Applets using the buggy V1 of the design fragment is likely too low, and a more accurate number may be 96%.

The prevalence of this buggy code derives from the copying of a buggy example, and the inability of the example author to influence the copies. An Applet bound to the buggy V1 of the design fragment, however, could be notified by the tools when the design fragment is marked deprecated.

6.5.2 Non-Conformance

We observed substantial non-conformance to framework constraints. We will focus on two constraints. The first constraint is that an Applet that registers for events should also de-register for them. The demo Applets appear to be trying to adhere to the register/de-register constraint, and only 3 fail to do so. In some of the Applets that do conform, comments in their change logs indicate that missing calls to de-register have been added.

The second constraint is that Applets should publish and check the same set of parameters. An Applet can ask the framework if the user has passed it a parameter by calling the `getParameter` service method, and it can publish the set of parameters it takes by overriding the `getParameterInfo` callback method. The demo Applets were unerring in conformance to the Parameterized Applet design fragment, in that every parameter they checked with a call to `getParameter` was paired with the results of `getParameterInfo`.

Evidence. The Applets from the internet often did not conform to these constraints. In the Applets we examined, almost all of the listening Applets failed to de-register for events, and about two-thirds failed to pair the parameters they read via `getParameter` with the parameters they reported via `getParameterInfo`. The observed non-conformance of the Applets we examined is shown in Table 6.3.

Our aggregate demographic information obtained via internet searches (see Table 6.2) supports our detailed examinations. Searches reveal that less than 10% of Applets de-register for mouse listener events, and less than 1% de-register for component events. The parameter con-

	Demo Applets	Internet Applets
Failure to define getParameterInfo	0 / 13	12 / 17
Failure to de-register for events	3 / 16	26 / 27

Table 6.3: Non-conformance to design fragment

straint is worse, with searches revealing that only 2% of Applets that use parameters both use, and report using, parameters. Our aggregate searches cannot reveal if the checked and reported parameters were paired up in that 2%, but this was a common problem in the Applets we examined.

We note these deviations not as compelling evidence that design fragments can reduce bugs in code, but rather as evidence that even in debugged, released code it is possible to find violations of framework constraints because of non-local client-framework interactions.

Framework encapsulation. It is likely that with such widespread non-conformance the Applet framework authors cannot change the framework implementation in a way that it requires Applets to observe these constraints. This raises two questions. First, what is the impact of the non-conformance? And second, was the non-conformance deliberate?

As the Applet framework is currently implemented, there is no user-perceived error when an Applet fails to de-register for events. One perspective is that it is unimportant because the Applet is being destroyed anyway, and so has no impact. Our perspective is that the failure to de-register violates framework encapsulation. The assumption that the framework will ignore the violation is itself a dependency on the framework's internal implementation details.

In other frameworks, notably the Eclipse framework, a failure to de-register can yield errors, including null pointer exceptions. This happened to us during our development of tools in Eclipse. We failed to de-register when one of our views was closed, so subsequent events were still sent to our view. Garbage collection did not delete our view since there were still references to it, but internal invariants no longer held in this state of limbo. Errors occurred when the view tried to process the event, leading to null pointer exceptions.

Through their descriptions of classes, interfaces, and methods, the Applet framework authors specified details that clients could depend on, while attempting to keep their implementation details changeable. Client programs, by depending on implementation details of the Applet framework, have forced a new constraint on the framework designers, and have limited the evolvability of the framework.

One interpretation of this data is that programmers incorrectly inferred the constraint, and another interpretation is that they were knowingly ignoring it. It is unfortunate if programmers incorrectly inferred the constraints from the demo Applets. But it is equally unfortunate if programmers correctly inferred how to use the framework, but then deliberately chose to violate its constraints.

Whether the non-conformance is evidence of incorrect inference or evidence of deliberate dependence on framework internals, it is evidence that the standard strategy of using method signatures and demo programs to specify client-framework interactions is deficient. Furthermore, the use of design fragments would reveal to framework users that they are depending on a safe and supported use of the framework.

```

public class Simple
    extends java.applet.Applet {
    public Simple() {
        add( new java.awt.Label("Hello") );
    }
}

```

Figure 6.8: Simple AWT Applet

6.5.3 Selection Bias

Because of how Applets evolved out of the Abstract Widget Toolkit (AWT), not all Applets meaningfully engage in the Applet framework. Figure 6.8 shows an example of a simple AWT program that extends `java.applet.Applet` yet does not meaningfully engage with the Applet framework. Every legal AWT program is also a legal Applet so long as it derives from `java.applet.Applet`. The engineering advantage to this choice was that any AWT program could be run in a web browser simply by subclassing from `java.applet.Applet`.

Our search process eliminated simple programs, as above, by requiring that the Applet override a framework callback, or invoke a framework service method. We did this by adding the names of these callbacks and service methods to our search string.

As a result, we were sure to get Applets that used the features we searched for, but our collection of Applets no longer represented a neutral sampling of Applets on the internet. A possible concern about our process is that our searches preferentially targeted specific kinds of Applets, specifically those using threads, engaging in event listening, and reading parameters. Our internet searches indicate that 27% of Applets use threads, 3.5% use events, and 18% use parameters.

In addition to bias introduced because of our targeted searches, there is bias in the ordering of search results from any search engine. We used the Google search engine, which tries to sort the most relevant results to the top using proprietary algorithms and heuristics. We may have uncovered this bias when we saw 75% conformance to the (buggy) Background Continuous V1 design fragment from the top Google results (see Table 6.1) versus a 96% conformance based on raw search results (see Table 6.2).

6.5.4 Authoring Consistency

Early versions of our design fragments were often overly specific, but we were able to generalize them when they were recognized in subsequent Applets. For example, the first Applet might register *itself* for framework event messages, but the second Applet registered *a separate object* for the same messages. This can be generalized by creating two roles in the design fragment, one role for the object that registers and another role for the object that receives the messages.

When the differences between Applets were irreconcilable, two versions of the design fragment were created. While we expect that irreconcilable differences could result from inventiveness of the client programmers in developing solutions, in this case study the only irreconcilable difference found was a result of a threading bug that was later fixed, yielding a design fragment

Background Continuous V1. Must have a field holding reference to thread, Must create thread in `start`, assigning field to thread, Must set field to null in `stop`, Must implement `run` and loop continuously until field is found to be null.

Background Continuous V2. Must have a field holding reference to thread, Must create thread in `start`, assigning field to thread, Must set field to null in `stop`, Must implement `run` and loop continuously until field is not the same as currently running thread.

One-time Init Task. Must create thread in `init`, Thread must execute `run` to completion just once.

One-time On-demand Task. May create thread at any time, Thread must execute `run` to completion just once.

Timer. Must create a new TimerTask in `start` and call `schedule` on it, Must call `cancel` on the TimerTask in `stop`.

Event Handling (all kinds). Must register in `init` using `addZZZListener`, Must implement relevant interface, Must implement interface methods, May fail to de-register in `destroy` using `removeZZZListener`.

Parametrized Applet. Must call `getParameter`, perhaps not from `init`, May fail to define `getParameterInfo`, May fail to match `getParameter` calls with `getParameterInfo` data.

Manual Applet. Must have `main` method that calls `init` and `start` on Applet

Figure 6.9: Conformance criteria for design fragments

for the buggy pattern and another for the fixed one.

Recognizing a design fragment is equivalent to defining a category. The design fragment author must examine source code and recognize that a subset of that code is repeated elsewhere. The author then encodes this pattern as a design fragment and binds it to the source code. Some of our initial attempts to define design fragments were overly broad (e.g., an Applet that paints to the screen) and others overly narrow (e.g., an Applet that has a background thread for running a control panel). The selection of an appropriate scope became easier after defining a dozen or so design fragments. However, it is natural that different authors would create different design fragments in same way that different authors would create different code libraries.

6.5.5 Hypothesis: Small Catalog

The creation of a catalog of design fragments is an investment of time and energy that must be recouped by usage afterwards. The faster the catalog can be built, the faster the investment can be recouped. Ideally, the demo Applets would reveal all the design fragments seen elsewhere.

Our examination of thirty-six internet Applets led to only two design fragments being added to the catalog. One of the two added design fragments, however, is the deprecated Background Continuous V1, a buggy version of Background Continuous V2. It is marked as deprecated in our catalog, and our tools warn any client programs that bind to it.

The second design fragment we added was Focus Listener, one of the five event handling

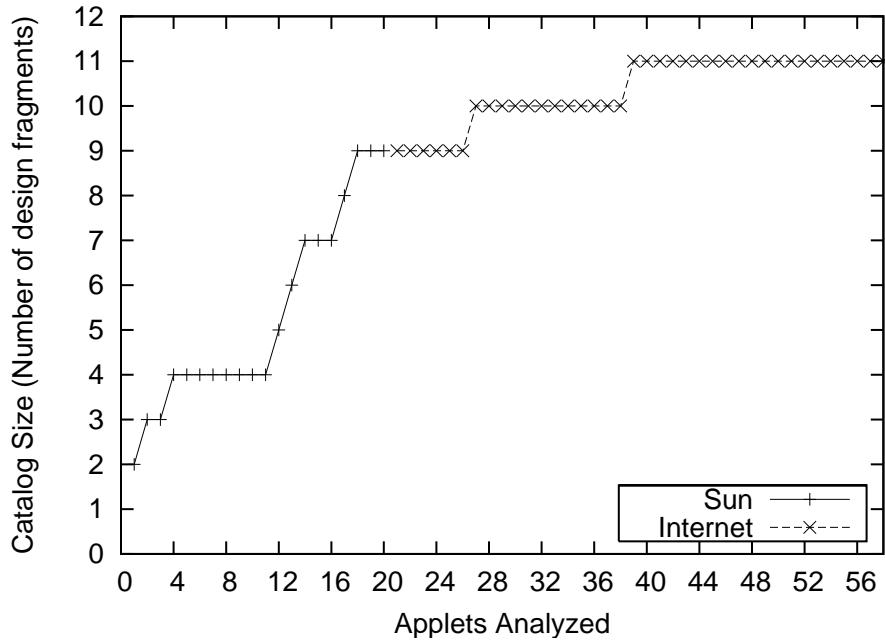


Figure 6.10: Applet catalog growth

design fragments, and was structurally the same as the others. The API documentation for Applet reveals that the following nine events can be requested: component, focus, hierarchy bounds, input method, key, mouse, mouse motion, mouse wheel, and property change. Presumably there should be identically structured design fragments for all nine of these. In the fifty-six Applets examined, we found Applets that used five of the nine (component, focus, key, mouse, and mouse motion).

Overall, the observed rate of catalog growth appears to be asymptotic, as is shown in Figure 6.10. This graph shows that most of the design fragments were discovered quickly, with ten of the twelve present in the demo Applets. The internet Applets interacted with the framework with high consistency, and as a consequence we were able to bind the existing design fragments already in our catalog instead of defining new ones.

It is worth considering the perspective of an Applet programmer. If the Applet framework authors had delivered the demo Applets, and had bound them to a catalog of design fragments, how much would this have helped an Applet programmer? By looking at the graph and assuming that the programmers use the correct Background Continuous V2 instead of V1, it would appear that thirty-six new Applets could be written, requiring just one additional design fragment. But the prospects are even better, because the complete data, shown in table 6.5, shows that the one additional design fragment had a single use. So, thirty-five of the thirty-six new Applets could have been created from the original catalog, indicating that the original catalog would have been quite helpful to programmers writing new Applets.

Thread field name	Demo Applets	Internet Applets
<i>killme</i>	1	1
<i>engine</i>	1	1
<i>runner</i>	2	4
<i>kicker</i>	1	1
<i>timer</i>	1	0
<i>marcher</i>	0	1
<i>my_thread</i>	0	1
<i>_helloThread</i>	0	1
<i>aniThread</i>	0	1
<i>clock</i>	0	1
<i>tetris1, tetris2</i>	0	1
<i>artist</i>	0	1
<i>loader</i>	0	1

Table 6.4: Frequency of thread field name

6.5.6 Hypothesis: Examples Strategy

Before starting this case study, our hypothesis was that programmers trying to use a framework sought out example code, studied that code to discern the patterns of framework use embodied in the code, and then used those patterns in their own code. We believed that few programmers were returning to “first principles”: studying the framework API and inventing a solution that worked within the constrained solution space provided by the framework. The results of this case study support these beliefs.

It is likely that programmers were referencing and copying the demo Applets. The first indication is the duplication of the thread field names found in both the demo Applets and the internet Applets. Table 6.4 shows the thread field names from the demo and internet Applets. Note that the highlighted field names *killme*, *engine*, *runner*, and *kicker* are found in both the demo Applets and the internet Applets. Overall, seven of the fifteen field names used in the internet Applets originated in the demo Applets.

The second indication that programmers were referencing and copying the demo Applets is the prevalence of the threading bug originally found in the earliest demo Applets. 75% (9 in 12) of the threading Applets we directly investigated, and as many as 96% (14,090 in 14,600) of the Applets found via an internet search, exhibit this bug. Both items together strongly suggest that programmers looked to the demo Applets to learn how the framework should be used.

Based on the evidence from this case study, it appears that the variety of ways that programmers use the framework is limited. As we discuss in more detail in Section 9.3.2, we believe that the demo Applets acted as seed crystals, and programmers who could have solved problems in many ways instead chose to use the solution found in the demo Applets. 27% of Applets found on the internet are using a thread, yet all 15 threaded Applets we examined in detail employed either the Background Continuous or One-Time On-Demand Task design fragments. While there is ample opportunity for variety, Applet programmers appear to have reused existing solutions.

6.6 Summary

This case study of the Applet framework has supported our hypotheses that there is limited variety in how client programs interact with frameworks, and that programmers follow an example strategy when building client programs. We examined twenty demo Applets and thirty-six internet Applets, and built twelve design fragments to specify the client-framework interactions we observed. Most of the design fragments were found in the demo Applets, and one of the design fragments found in the internet Applets contained a bug that had been fixed nearly a decade ago, but whose fix had not propagated to the Applets that had copied it.

		Background Continuous V1	Background Continuous V2	One-time Init Task	One-time On-Demand Task	Timed Task	Component Listener	Focus Listener	Key Listener	Mouse Listener	Mouse Motion Listener	Parameterized Applet	Manual Applet	Total DFs	Sun Total DFs	Internet Total DFs	New DFs Found	
Sun demo	animator													3				
Sun demo	arctest	1												1				
Sun demo	barchart													1				
Sun demo	blink													2				
Sun demo	cardtest													1				
Sun demo	clock	1												2				
Sun demo	ditherest	1												3				
Sun demo	drawtest													1				
Sun demo	fractal													3				
Sun demo	graphicstest													1				
Sun demo	graphtayout													1				
Sun demo	imagemap	1												4		6	1	
Sun demo	jumpingbox													3		7	1	
Sun demo	moleculeviewer													4		8	1	
Sun demo	nervoustext	1		1										3		8		
Sun demo	simplegraph													0		8		
Sun demo	sordemo													3		9	1	
Sun demo	spreadsheet													3		10	1	
Sun demo	tictactoe													1		10		
Sun demo	wireframe		1											4		10		
Internet	anbutton	1												2		10		
Internet	antacross	1												3		10		
Internet	antmarch	1												3		10		
Internet	blinkhello			1										1		10		
Internet	brokeredchat			1										3		10		
Internet	bsom													1		10		
Internet	buttontest	1												2		11	1	
Internet	cte													0		11		
Internet	demographics													2		11		
Internet	dotproduct													3		11		
Internet	envelope													0		11		
Internet	fireworks	1												2		11		
Internet	gammabutton													1		11		
Internet	geometry													1		11		
Internet	hellotcl	1												3		11		
Internet	hyperbolic													2		11		
Internet	iagtager													1		11		
Internet	inspect	1												1		11		
Internet	kbdfocus													3		12	1	
Internet	linprog													2		12		
Internet	mousedowndemo													2		12		
Internet	myapplet2													4		12		
Internet	nickcam													1		12		
Internet	notprolog													1		12		
Internet	scatter													1		12		
Internet	scope	1												2		12		
Internet	simplepong													1		12		
Internet	simplesun	1												2		12		
Internet	smtp													1		12		
Internet	superapplet													2		12		
Internet	tetris	2												2		12		
Internet	ungrateful													1		12		
Internet	urccalendar													1		12		
Internet	urlexample													2		12		
Internet	webstart	1												4		12		
Internet	ympyra													1		12		
Total		9	9	2	4	1	2	1	3	22	15	30	10	108				

Table 6.5: All Applets with origin and counts of design fragments

Chapter 7

Case Study: Eclipse Framework

The previous chapter presented a case study on the Applet framework, investigating whether the use of design fragments was practical on a small framework. Our results showed that a catalog of twelve design fragments could describe the interactions of the fifty-six client Applets studied. However, since the Applet framework is small and ten years old, we were concerned that the results could be different on a larger and newer framework.

This chapter presents a case study on the Eclipse framework, a large framework undergoing current extensive development. To be clear, we have dealt with Eclipse in many ways in this thesis. Eclipse is a Java Integrated Development Environment (IDE). Our design fragment tools work within that IDE, and those tools are built on the Eclipse framework. This chapter presents a case study on other programs that use the Eclipse framework.

In this case study, we set out to investigate whether, in a large framework, the patterns of client-framework interactions were limited. We created a catalog of design fragments by examining a demo Eclipse client program. Our results indicate that the design fragments in our Eclipse catalog are close matches to the client programs from the internet, supporting the hypothesis that clients are following consistent patterns when interacting with the framework.

We also set out to investigate whether design fragments could cover a majority of the client-framework interactions. To do this, we tallied all client-framework interactions in our demo client program. This case study reveals that our design fragment catalog covered 98% of such interactions.

This chapter details our case study procedure and results, and then discusses the evidence for our hypotheses.

7.1 Eclipse Framework

The Eclipse framework allows software tools, as client programs, to be written and interoperate. It is composed of many smaller frameworks that build upon each other. For example, the popular Java Development Tooling framework, which is used by programmers as a Java IDE, is an extension of the Eclipse Platform framework, which itself extends the Eclipse Equinox component model framework.

Eclipse is quite large for a framework at nearly two million lines of code [11]. It has been

under development since 1999 [35], initially by a team of forty IBM and OTI developers who had previously written IBM’s VisualAge development environment. Now the Eclipse Foundation, a not-for-profit corporation supported by 115 commercial companies, maintains stewardship of the framework and the source code is extended by the constituent Eclipse companies via open source collaboration.

7.2 Hypotheses

This case study was designed to investigate whether the variety of design fragments was low. We hoped to find that the design fragments in our catalog were also seen in client programs from the internet. Ideal evidence would be 100% conformance between the design fragments we defined and the client-framework interactions seen in the internet client programs.

We hoped to find evidence that client programmers were following an examples strategy. Our catalog was created from a demo Eclipse client program in the expectation that other client programmers would also have referenced it when creating their client programs.

The following section describes our case study process.

7.3 Process

Our primary concern in the design of our process was how to collect convincing evidence regarding the variety of client-framework interactions while keeping the effort manageable. In the Applet case study, each Applet was examined fully, but most Applets used only a few design fragments. Our prior experience programming within the Eclipse framework provided us the intuition that many Eclipse client programs would use a dozen or more design fragments. We chose to examine in fine detail a single demo client program, and create a catalog of design fragments from it alone. To find out if other client programs interacted with the framework in the same way, we conducted targeted searches to find client programs on the internet. We did not fully examine each of these, but instead evaluated each to see if it used design fragments from our catalog.

While we were examining our client program in detail, we collected data to compute a coverage metric, which indicates how many client-framework interactions in the client program were specified in our design fragments. We also took the opportunity to create a sufficiently complete list of client-framework interaction types. The following sections describe our process in detail.

7.3.1 Choose Client Program

As the subject of our investigation, we selected a client program for the Eclipse framework. Like in the Applet case study, we wanted to choose a client program provided by the framework authors so that the design fragments we found were most likely to represent canonical client-framework interactions.

Unlike the Applet framework, where the demos are simply included in a directory, the Eclipse framework has a “wizard” that guides the user through a series of dialog boxes to create some

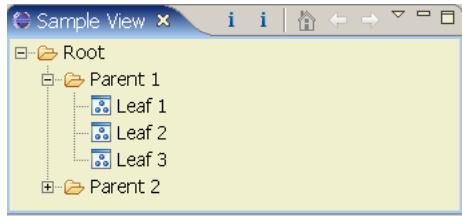


Figure 7.1: Screenshot of our client program

example code with features that the user selects from a limited list. Eclipse client programmers are as likely to have seen code generated by this wizard as Applet client programmers are to have seen the demo Applets.

We created a client program with this wizard, choosing options for a client program that puts a tree view on the screen. It is a sophisticated “Hello, world” program. A screenshot of the program running is shown in Figure 7.1.

Our selected client program is heavy on framework interaction, and relatively light on domain-specific processing. About 10% of the program is devoted to setting up a simple parent-child data structure to be presented in the tree view. The overall size of the program including whitespace and comments is 451 lines, and about 245 lines when whitespace and comments are omitted, but leaving end curly brackets on their own lines.

7.3.2 Identify Client-Framework Interaction Types

Starting with our chosen client program, we identified all of the client-framework interaction types. We insisted that each interaction be observable from the syntax of the source code. We categorized the client-framework interactions and recorded the frequency of occurrence of each of them in our client program.

7.3.3 Create Design Fragment Catalog

At the beginning of this case study, we had a pre-existing catalog of Eclipse design fragments that we had created two years earlier. We started with that catalog, and modernized it to accommodate changes to the Eclipse framework, adding or revising the design fragments that were found in our client program. The new and revised design fragments were created by examining our client program until we understood it and could identify the relevant non-local client-framework interactions that contributed to a single goal.

We declared instances of the design fragments from our catalog, and bound them to our client program.

7.3.4 Identify Required and Optional Interactions

We examined each client-framework interaction in our client program, and tagged it as being *required*, *optional*, or *no interaction* for each design fragment. For example, on the Pulldown Menu Action design fragment, it is required for clients to call the `getMenuManager` method,

and for them to include an `Action` in the menu. But it is optional for them to add a `Separator` to the menu, or to set the text displayed when the user hovers the mouse over that menu item.

While the choice of what was required or optional was subjective, few parts of the design fragments were optional in our catalog. The criterion for an interaction to be required was, “Is it essential to the design fragment?”, and for optional parts, “If it is not essential, is it likely that clients will often do this part too?”

From this data we computed coverage metrics, indicating how much of the client-framework interaction could be explained through the design fragments.

7.3.5 Examine Internet Use

Our process so far had yielded a catalog of design fragments that we had derived from just one client program, a client program provided by the framework authors. To see if our design fragments were also seen in other client programs, we looked for their occurrence in client programs found on the internet that were trying to accomplish the same goal. We set an acceptance threshold of 30%, meaning that for us to accept that a design fragment was genuine, it should appear in at least 30% of the internet client programs.

Our intent was similar to the one used in the Applet case study. We chose appropriate search strings, searched the internet for client programs, and compared the client-framework interactions in the resulting client programs to those specified by our design fragments. When analyzing the Applets, we fully examined each, and added new design fragments to our catalog as needed. This was impractical on the Eclipse framework, given the size of the framework and the size of the client programs. Here, we focused our attention on how the internet client programs used the design fragments we already had in our catalog, and did not define any new design fragments.

The first step was to find Eclipse client programs on the internet. Our base search string to find these client programs was:

```
filetype:java org.eclipse -IBM
```

This yields only Java programs that use the Eclipse framework, but excludes client programs written by IBM.

The second step was to focus this search to yield client programs that might be following one of our design fragments. We wanted to find client programs that had the same goal as one of our design fragments, then to examine them to see if they accomplished that goal consistently with the design fragment we had already defined.

The challenge with this approach is to target the search, but not narrow it so much that it only yields occurrences of our design fragments. From each design fragment we extracted a subset of keywords that were sufficient to find appropriate uses, but not so specific that they would only find exact matches to the design fragment. We explore this possibility in section 7.5.1. The search strings are shown in Table 7.1.

Each of the 12 search strings was entered into Google and the top 10 results were recorded, for a total of 120 results. These results were client programs that used the Eclipse framework, and we hoped had goals matching our design fragments.

We analyzed each client program to see if it matched the design fragment we had in our catalog. We scored each comparison as one of three choices: an exact match, a basic match, or

Design Fragment	Search String
Content Provider	ITreeContentProvider ViewPart setContentProvider
Double Click Action	IDoubleClickListener addDoubleClickListener ViewPart
Drilldown Adapter	DrillDownAdapter addNavigationActions ViewPart
Label Provider	LabelProvider setLabelProvider ViewPart
Popup Dialog	openInformation
Pulldown Menu Action	ViewPart getMenuManager add
Right Click Menu	IMenuListener ViewPart addMenuListener
Shared Action Image	getImageDescriptor getSharedImages setImageDescriptor
Shared Image	getImage getSharedImages
Toolbar Menu Action	ViewPart getActionBars getToolBarManager add
Tree View	“new TreeViewer” setInput ViewPart
Viewer Sorter	setSorter ViewerSorter ViewPart

Table 7.1: Internet client program search strings

not a match.

Exact Match. An exact match meant that the client program was required to conform to every part of the design fragment as it was already written.

Basic Match. When it was possible to have revised the design fragment such that it could accommodate both the usage seen in our original client program and the internet client program, we scored it as a basic match. A basic match can be thought of as an over-specified design fragment. We discussed our process for reconciling design fragments in Section 6.5.4.

Not A Match. Everything else was scored as not a match.

We did not require the results to be unique, so some client programs were examined for more than one design fragment.

7.4 Results

The results of our case study include the following artifacts: a list of client-framework interaction types, a catalog of Eclipse design fragments, bindings between client programs and our design fragments, and a spreadsheet with line-by-line tagging of client-framework interactions by type and the design fragments that relate. We were able to compute some aggregate metrics from these artifacts, including coverage of client-framework interactions by our design fragment catalog.

7.4.1 Client-Framework Interaction Types

We created a list of the types of client-framework interaction found in our client program. Our goal was to identify as many framework interactions as possible, while ensuring that identification was unambiguous from the syntax of the program. The items in the list can be unambiguously identified from simple syntactic analysis of the client source code. The following list is

sorted, and begins with the most common interaction type.

Method Invocation. The client invokes a method on a framework class. Only a subset of the framework methods can be invoked by a client. These are known as “service methods”. Service methods line up quite closely with Java’s *public* methods on the framework classes but there are cases where framework methods must be made *public* for implementation reasons, yet are not intended for use by framework clients.

Overrides Method. The client class is a subclass of a framework class and overrides (creates a method with the same name as) a method on that superclass. This is usually done with a method that has been chosen for this purpose by the framework designer, a so-called “template method.” Java methods designated as *private* cannot be overridden but methods may be left as *public* or *protected* for reasons other than allowing overriding.

New Class Instance. The client creates a new instance of a framework class or subclass. This interaction will rarely do anything useful since creating a new instance usually does not change the state of other objects in the framework but it is a prerequisite for other stateful changes.

Static Class Reference. The client makes a named reference to a framework class, for example, by referring to `org.eclipse.SomeClass`. This is in contrast to using object references passed to the client as parameters. Note that a static class reference usually implies either a static field access or a static method access.

Static Field Access. The client reads a static field on a framework class. Theoretically, the client could also read non-static public fields on non-static framework objects, but such a design leads to poor encapsulation and consequently we did not find any examples of it in the case studies.

Extends Superclass. The client defines a new class that is a subclass of a class (i.e., extends) provided by the framework.

Implements Interface. The client defines a new class that implements an interface provided by the framework.

Invokes Static Method. The client invokes a static method on a framework class.

Field Holding Framework Object. The client assigns to one of its fields a reference to a framework object. This has the effect of saving some state provided by the framework.

While we would prefer that this list includes every type of client-framework interaction present in our client program, our subjective identification process used in developing the list does not provide assurance of completeness. Additionally, we know that our client program does not contain every type of client-framework interaction. For example, our client program did not reference any public fields from framework objects, and such client-framework interaction is usually discouraged by framework authors, but it is known to be missing from our list.

Analysis of the data reveals the relative frequency of each identified type of client-framework interaction. As seen in Table 7.2, the most common was a method call from the client to the framework, accounting for 43% of all interactions. Also common were overridden methods and creation of framework objects, accounting for a quarter of all interactions. While subclassing and implementing interfaces are commonly thought of as essential parts of interacting with a

Framework Interaction Type	Count	%
method invocation	50	43%
overrides method	15	13%
new instance	15	13%
static class reference	10	9%
static field access	8	7%
extends superclass	6	5%
implements interface	4	3%
static method invocation	4	3%
field holding framework object	4	3%
Total	116	100%

Table 7.2: Framework interaction counts

framework, the combined frequency is relatively low, at 8%, possibly because one subclass overrides and calls many methods. There were 116 total interactions between the client code and the framework in the roughly 245 source lines, indicating dense framework interaction.

7.4.2 Design Fragment Catalog

The following is a list of the design fragments in our Eclipse catalog. The number in parentheses following the name of the design fragment is the number of required client-framework interactions specified in the design fragment.

Content Provider (13). Provides the content to be displayed by a view.

Double Click Action (8). Adds double-click behavior to a view.

Drilldown Adapter (6). Adds navigation controls for drilling into a tree view.

Label Provider (8). Provides the labels corresponding to content displayed by a view.

Popup Dialog (4). Pops up a modal dialog box to the user.

Pulldown Menu Action (13). Adds an action to a pulldown menu on a view.

Right Click Menu (16). Adds a right-click “context” menu to a view.

Shared Action Image (5). Use shared images from the workspace using Image Descriptors.

Shared Image (3). Use shared images from the workspace.

Toolbar Menu Action (12). Adds an action to the toolbar menu on a view.

Tree View (8). Creates the skeleton of a tree view.

Viewer Sorter (5). Sorts the content in a tree view.

7.4.3 Coverage by Design Fragments

After creating the catalog, we tagged the client program line-by-line to reveal the connection between each client-framework interaction and our design fragments. Table 7.3 shows an excerpt from this effort. In the table, an “R” represents a client-framework interaction that is required by

the design fragment, “O” an optional interaction, and “-” no interaction. For example, line 352 should be interpreted as meaning that a New Instance interaction, where a new TreeViewer object is created, is required by one design fragment, the Tree View design fragment.

Since there can be multiple framework interactions on a single line of source code, we have reformatted the source code shown in Table 7.3 so that there is just one interaction per line. To improve readability, we have omitted the Static Class Reference interactions from the table. They are present in line 352 where the class named SWT is explicitly named three times.

This excerpt is derived from a spreadsheet whose rows were the reformatted lines of source code, and whose columns were the design fragments. The spreadsheet also counted the R’s and O’s to reveal how many design fragments mentioned each interaction. We derived the data shown in Table 7.4 from the spreadsheet.

Most interactions, 67%, were required by a single design fragment, indicating that the interaction existed in support of a single goal. However, some framework interactions existed to support more than one goal. 11% of the interactions were required by two or more design fragments. Since each design fragment has a single goal, this means that 11% of the client-framework interactions existed to support multiple goals. We discussed this condition, called tangling, in Section 4.3. An example of tangling of goals is seen in line 351 in Table 7.3, where the overriding of the `createPartControl` method is required by multiple design fragments. The design fragments all require that the client program do something in the body of this method. Most of the tangling that we observed was on callback methods, like `createPartControl`, or on fields.

Our line-by-line examination enables us to calculate what percentage of the client-framework interactions in our client program are covered by design fragments. For our definition of “coverage,” we include any interaction that is mentioned as a required or optional part of a design fragment. Based on this definition, 98% of the interactions between the client and framework were covered, with just 2 in 116 interactions left uncovered. A tighter definition that includes just the required parts from the design fragment yields 78% coverage.

The two client-framework interactions that were not covered by a design fragment dealt with extracting some data out of the current selection parameter so that the data could be presented on the screen. The two method calls not covered by design fragments are bolded in the following code snippet:

```
ISelection selection = viewer.getSelection();  
Object obj = ((IStructuredSelection)selection).getFirstElement();  
showMessage ("Double-click detected on "+obj.toString());
```

It would be possible to write a design fragment to express the intent of these client-framework interactions, perhaps one called Debug Selection Change with Popup Message. We did not create such a design fragment because it appears to us to be a special case, and not generally usable. It would also be possible to create a Get First Selection design fragment, which could be more reusable.

7.4.4 Internet Use of Design Fragments

One question that remains after looking at the extensive coverage of client-framework interactions by design fragments is whether or not the design fragments in our catalog resemble what

Line	Source Code	Interaction Type	Content Provider	Double Click Action	Drilldown Adapter	Label Provider	Popup Dialog	Pulldown Menu Action	Right Click Menu	Shared Action Image	Shared Image	Toolbar Menu Action	Tree View	View Sorter
351	public void createPartControl(Composite parent) {	overrides	R	O	-	R	-	R	-	O	R	R	-	-
352	viewer = new TreeViewer(parent, SWT.MULTI SWT.H_SCROLL SWT.V_SCROLL);	new instance static field static field static field	-	-	-	-	-	-	-	-	-	R	-	-
353	drillDownAdapter = new DrillDownAdapter(viewer);	new instance	-	-	R	-	-	-	-	-	-	-	-	-
354	viewer.setContentProvider(new ViewContentProvider());	method invocation	R	-	-	-	-	-	-	-	-	-	-	-
355	viewer.setLabelProvider(new ViewLabelProvider());	new instance	R	-	-	R	-	-	-	-	-	-	-	-
356	viewer.setSorter(new NameSorter());	method invocation	-	-	R	-	-	-	-	-	-	R	-	-
357	viewer.setInput(getViewSite());	new instance	-	-	-	-	-	-	-	-	-	R	-	-
358	makeActions();	method invocation	-	-	-	-	-	-	-	-	-	O	-	-
359	hookContextMenu();	method invocation	-	-	-	-	-	-	-	-	-	R	-	-
360	hookDoubleClickAction();	method invocation	-	-	-	-	-	-	-	-	-	-	-	-
361	contributeToActionBar();	method invocation	-	-	-	-	-	-	-	-	-	-	-	-
362	}													

Table 7.3: Excerpt of code coverage

Framework Interaction	Count	%
All interactions	116	100%
Required by 1 or more DFs	91	78%
Required by exactly 1 DF	78	67%
Required by 2 or more DFs	13	11%
Optional by 1 or more DFs	24	21%
Required or optional by 2 or more DFs	17	15%
Not covered by any DF	2	2%
Covered by some DF	114	98%

Table 7.4: Design fragment coverage of framework interactions

most client programs do. We searched the internet for similar client programs. For us to believe that our design fragment was real, we required it to express the client-framework interactions seen in 30% of the client programs from the internet that were selected to have the same goal. We believed that this threshold provided evidence that our design fragments reflect actual usage, and are not distorted in order to yield high coverage metrics in the previous analysis.

As described in Section 7.3.5, we searched the internet to find client programs whose goals matched those of each of our design fragments. We collected ten client programs for each of the twelve design fragments, except for the Drilldown Adapter, because there were only eight unique results from our query. Many programs were examined more than once because they appeared as the top results for multiple searches. We examined forty-two different client programs.

We graded each client program as an exact match if the design fragment specified the client program exactly, a basic match if we could have revised the design fragment to specify both the demo and the internet client programs, and no match if not, as described in Section 7.3.5. Figure 7.2 shows a chart of the conformance.

Every design fragment in our catalog exceeded our acceptance threshold of 30% conformance. The exact match conformance ranged from 30-100%. When basic matches were included then the range was 50-100%. The low 50% conformance was on the Label Provider design fragment. Further examination revealed that our search string was overly broad, and yielded code for label providers of tree data and table data, yet each used a different set of client-framework interactions. Had there been design fragments for Tree Label Provider and Table Label Provider, they would have both been at 100% conformance.

Three client programs did not match our design fragment for Double Click Action. The design fragment describes how to register for a double click event and, when notified of a double click event, execute an Action. The three non-conforming client programs did not execute an Action, but instead did their work inside the callback method informing them of the double click event.

Our conjecture is that conformance was related to the complexity of the design fragment. It is plausible that programmers make verbatim copies of the most complex code since it can be the most difficult to understand. However, our analysis shows no correlation between the number of framework interactions in a design fragment and the conformance seen in the client programs from the internet.

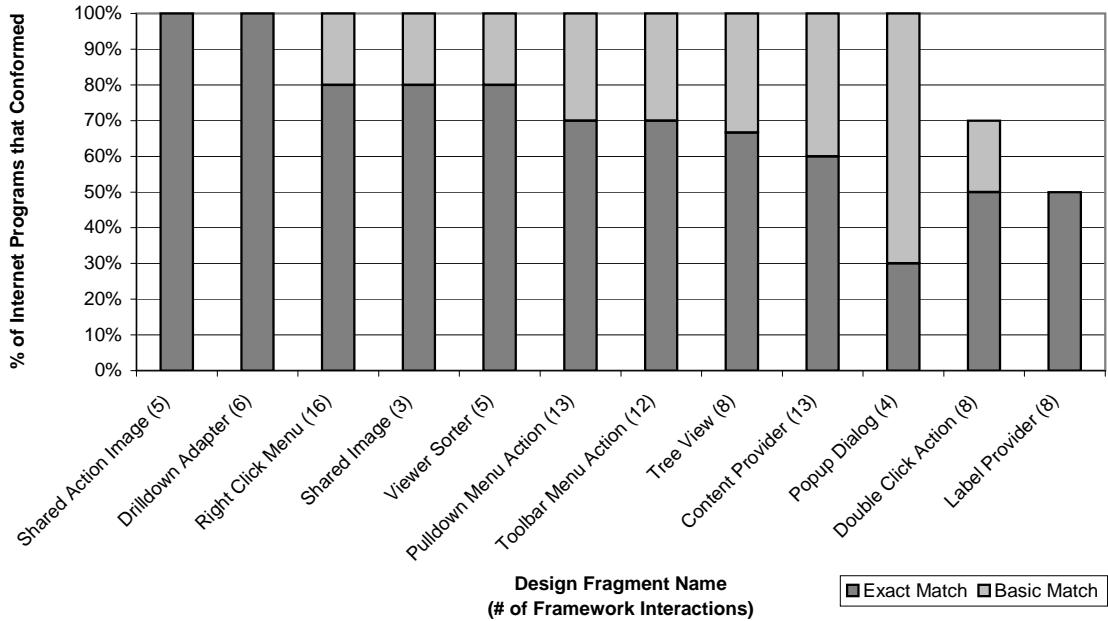


Figure 7.2: Design fragment catalog conformance

7.5 Discussion

A surprising outcome of this case study was how well our design fragment catalog, derived from a single demo client program, could specify client-framework interactions found in internet client programs. In this section, we discuss whether this could have been caused by a bias in our selection of internet client programs. We were able to create design fragments that covered 98% of the client-framework interactions in our demo client program, and so we discuss the significance of that coverage and how it may benefit programmers. We conclude our discussion with an examination of our hypotheses: that the variety of design fragments is small, and that client programmers are following the examples strategy.

7.5.1 Selection Bias

A possible concern about our process is that we unintentionally searched the internet for client programs that were too similar to our selected demo client program. We could have done this by choosing our search terms excessively narrowly, which would yield results that were predetermined to conform to our design fragment.

To provide some quantitative data regarding the narrowness of our search, Table 7.5 reprises the previously described search strings for our design fragments, and this time has added columns for the lengths. We considered the length of the design fragment to be the number of distinct client-framework interactions it required. The length of the search string was the number of words in it. Most of the search strings had three or fewer words, and the two search strings with four words corresponded to design fragments that were at least twice as “long.”

Most of the search strings contained the word “ViewPart,” which corresponds to the Eclipse

Design Fragment	Length	Search String	Length
Content Provider	13	ITreeContentProvider setContentProvider ViewPart	3
Double Click Action	8	IDoubleClickListener addDoubleClickListener ViewPart	3
Drilldown Adapter	6	DrillDownAdapter addNavigationActions ViewPart	3
Label Provider	8	LabelProvider setLabelProvider ViewPart	3
Popup Dialog	4	openInformation	1
Pulldown Menu Action	13	getMenuManager add ViewPart	3
Right Click Menu	16	IMenuListener addMenuListener ViewPart	3
Shared Action Image	5	getImageDescriptor getSharedImages setImageDescriptor	3
Shared Image	3	getImage getSharedImages	2
Toolbar Menu Action	12	getActionBars getToolBarManager add ViewPart	4
Tree View	8	“new TreeViewer” setInput ViewPart	4
Viewer Sorter	5	setSorter ViewerSorter ViewPart	3

Table 7.5: Internet client program search strings with lengths

framework class `ViewPart`. As described in its JavaDoc, it is the “Abstract base implementation of all workbench views.” We included that word in our search strings because our client program was also a workbench view, and not a dialog box, or a wizard. It is possible that removing this term would yield lower conformance. However, if transient dialog boxes do not interact with the framework in the same way as more-or-less permanent views do, then the better path forward may be to create another catalog of design fragments that represent the client-framework interactions in dialog boxes.

Upon review, we do not find our search terms to be overly narrowing. For example, to our knowledge, any client program that wants to listen to double click events must use the `IDoubleClickListener` interface and call `addDoubleClickListener`. More likely, as we discuss in Section 7.5.4, is that client programmers are either starting with the wizard-generated code themselves, or copying from other client programs that have.

7.5.2 Coverage

At the onset of the case study, our belief was that design fragments would cover a majority of the interactions. But the data indicates that essentially all of the interactions (98%) were specified in one or more design fragments. We expect that the coverage of design fragments on other programs will be lower than we found here. This program had little domain-specific logic and its purpose was to use the framework in a straightforward manner. Other client programs may also use the framework similarly, but with others the quirks of their domains may push them to ask for less common services, which may not even be noted as optional parts of design fragments.

A more fundamental question relates to what the coverage metric means, regardless of its score. What has been shown here is that we have examined a client program with 118 client-framework interactions, and we have documented the rationale for 116 of those interactions to exist. That is, we know the larger design intent that underlies each interaction.

Perhaps more importantly, those 116 individual client-framework interactions have been abstracted into 15 instances of design fragments. Figure 7.3, previously shown in Chapter 4, shows

```

@df.instances({
    @df.instance(df="TreeView", inst="treeview1"),
    @df.instance(df="ContentProvider", inst="contentprovider1"),
    @df.instance(df="DoubleClickAction", inst="doubleclickaction1"),
    @df.instance(df="DrilldownAdapter", inst="drilldownadapter1"),
    @df.instance(df="LabelProvider", inst="labelprovider1"),
    @df.instance(df="RightClickMenu", inst="rightclickmenu1"),
    @df.instance(df="ToolbarMenuAction", inst="toolbaraction1"),
    @df.instance(df="ToolbarMenuAction", inst="toolbaraction2"),
    @df.instance(df="PulldownMenuAction", inst="pulldownaction1"),
    @df.instance(df="PulldownMenuAction", inst="pulldownaction2"),
    @df.instance(df="ViewerSorter", inst="viewersorter1"),
    @df.instance(df="SharedImage", inst="sharedimage1"),
    @df.instance(df="SharedActionImage", inst="sharedactionimage1"),
    @df.instance(df="SharedActionImage", inst="sharedactionimage2")
})
package eclipseTreeView.views;

```

Figure 7.3: Instance declarations reveal architectural understanding

the instance declarations for our client program. The use of design fragments has provided these declarative statements regarding the client program's goals. These declarations of the client program's intent are fairly comprehensible to a novice, and would be exceedingly comprehensible to an expert familiar with the catalog. For this client program, design fragments have provided a compact, declarative explanation of 98% of its client-framework interactions.

7.5.3 Hypothesis: Limited Design Fragment Variety

The Applet case study provided evidence that Applets followed a limited number of patterns when interacting with the framework. This low variability enabled us to make a small catalog of design fragments whose growth rate was asymptotic. If the client programs had not conformed to a limited number of design fragments, then the size of the catalog would have grown much too quickly to be pragmatic. Before the Eclipse case study, we did not know if the larger Eclipse framework would work with the design fragments technique. It was possible that Eclipse client programs did not follow patterns when interacting with the framework.

This case study revealed that client programs for the Eclipse framework were similar to client programs for the Applet framework, in that both had limited variety in how they interacted with the framework. This limited variety could be represented as design fragments. This evidence allays our concern that client programs of larger frameworks behaved differently, and that the design fragments technique would not work on larger frameworks.

After creating design fragments derived from our client program, we hoped their specifications would also match the client-framework interactions for a majority of the internet client programs. We found that, for ten of the twelve design fragments, all of the internet client programs were “exact” or “basic” matches. An exact match meant that the design fragment, without any changes, explained both client programs. A basic match meant that the design fragment was

over-specified, but that it could be revised to match both client programs.

These results support our hypothesis that the variety of design fragments to accomplish a goal is limited. Looking at Figure 7.2, the shaded bars represent client use of frameworks that we have explained with our design fragments. The white triangle of space in the top right of the graph are the cases we could not explain. This white triangle represents just 8 of the 118 times we tried to bind our design fragments. As noted in Section 7.4.4, correctly distinguishing tree label providers from table label providers eliminates five of those eight failures.

7.5.4 Hypothesis: Examples Strategy

In this case study, we found evidence that the internet client programmers had reviewed the example code provided by the framework authors. The internet programs often used exactly the same method names as our example client program. Examples include `makeActions`, `hookContextMenu`, `hookDoubleClickAction`, and `contributeToActionBar`, which are seen in the excerpt in Table 7.3. Note that the existence of these methods is not required by the framework, nor does the framework constrain the names of the methods. These methods were also called in the same order, even when order did not matter.

Two of the internet programs conforming to the Pulldown Menu design fragment followed the design fragment exactly, except that they did not actually add any actions to the menu. This appears to be a case where the code from the demo was copied “just in case”.

Several of the internet programs conforming to the Right Click Menu design fragment used slightly different method names than the design fragment, but those names were consistent with each other. Examples of these method names are `createContextMenu` and `fillToolbar`. It seems likely that these share another common ancestor besides our example program.

We believe that the high conformance rates seen in Section 7.4.4 are due to the internet programs often having a shared ancestor. If programmers had followed a “first principles” strategy and examined the framework API to design their client-framework interaction, then we would expect lower conformance rates.

An alternate hypothesis is that, for any of the goals, there is only one easy way to accomplish it with the framework. This would explain why we see such consistent use of the framework by the different client programs. It would also support the use of an *examples strategy*, like design fragments, since there would be few examples to collect. This one-way-to-do-it hypothesis is at least partly true, in that the framework authors had some use scenarios in mind and designed the framework to support them. However, client programmers are clearly copying the demo programs since that is the only hypothesis that can explain the identical method variable names.

Our analysis suggests that there has been substantial copying of code from the examples provided by the framework authors into the client code from the internet. This copying likely contributes to the success of the design fragment technique on this framework by reducing variability in client programs.

7.6 Summary

We performed a case study on the Eclipse framework to investigate if the variety of design fragments was small, and to investigate if programmers were following the examples strategy. To do this, we created a catalog of design fragments that covered 98% of the client-framework interactions of a demo client program. We then searched the internet for client programs whose goals matched those of our design fragments. We found that our design fragments were an exact, or basic, match for almost all of the client programs.

Chapter 8

Related Work

This work builds upon previous work in a number of areas. Researchers in the 1980's and 1990's documented the use of software frameworks that they observed being used in industrial and academic settings. Frameworks were described as a new reuse mechanism that differed from class libraries. Natural language design patterns were suggested as a way for programmers to understand frameworks [36]. Most design patterns are based on the ideas of role modeling, where a given class can play various roles and its responsibilities are the superset of the responsibilities of its roles. Research into design patterns led to tools that could model patterns precisely and compare them with source code. Some modeling tools and techniques shifted their emphasis from the whole pattern, which may describe how the framework is implemented, to instead describe how the program interacts with the framework. Cookbooks and recipes followed a similar path starting from unstructured text through a precise representation.

This chapter reviews the techniques that have been used to help client programmers work with frameworks. We organize the existing techniques into four categories and present a brief summary of each. Finally, we situate design fragments in relation to these works and provide a detailed comparison with the most similar techniques.

8.1 Organization of the Research

Table 8.1 provides a categorization of the work on framework documentation. It divides the work along two dimensions describing the style of documentation. The first dimension is whether the documentation provides examples or helps the programmer to design a solution from first principles. The second dimension is whether the documentation prescribes how clients should use the framework or describes the implementation of the framework.

Relatively few projects have set out to describe the internals of frameworks and most of these projects were relatively older, from a time when frameworks were themselves just beginning to become popular. Most research has focused on documenting how to use the framework rather than documenting how the framework is designed. Techniques such as JavaDoc and Design Patterns can be applied to almost any documentation problem and have been used to describe how frameworks work internally.

Techniques in the example-based dimension do not claim to document every possible and

	Prescribes how clients should use the framework	Describes the implementation of the framework
First-principles-based	Inscape Programmer's Apprentice Helm Contracts FCL Constraints UML-F Profile Declarative Metaprogramming OOram Riehle frameworks FSML JavaDoc	JavaDoc OOram Riehle fwks
Example-based	Decl. Metaprogramming Design Patterns Hooks JavaFrames Cookbooks JavaDoc Design Fragments	Design Patterns JavaFrames Utrecht Tool JavaDoc

Table 8.1: Organization of the research

correct use of the framework but instead provide known-good examples. This simplifies the task both for the documentation author and for the reader.

Conversely, techniques in the first principles-based dimension claim to cover all correct use in much the same way that functional specifications for methods should cover all cases of inputs and outputs. For example, the FCL Constraints work presents a case study on Microsoft Foundations Classes where all subclasses of `CWnd` must call one of three window creation methods defined in the framework.

8.2 Specific Techniques

Many techniques have been tried for helping programmers understand and use frameworks. In addition to the two dimensions from Table 8.1, a technique's degree of formality can be considered as an orthogonal dimension, and it influences the level of tool support that can be provided. We will consider each of these dimensions in this section.

We begin this section with a survey of the related work, and then continue in Section 8.3 with a detailed comparison of design fragments with the related work.

8.2.1 Role Modeling

Object Oriented Role Analysis Modeling (OOram) is a software engineering method developed by Reenskaug that focuses on collaborating objects (role models) instead of classes [53]. Each role model consists of a number of roles with assigned behavior. Classes are created by composing these roles. A tool for the Smalltalk language was created for authoring and composing these role models. Reenskaug recalls Cox's metaphor [10] of the surface area of components, that is, the things that must be understood about the component for a client to use it correctly, and applies it to frameworks. He notes that the surface area of a framework should be kept as small as possible, can be described with role models, and should not be changed for fear of breaking existing applications. Reenskaug is credited with the creation of the Model-View-Controller pattern, whose implementation in Smalltalk may be considered the earliest framework [38].

In his thesis [55], Riehle extends the role modeling concepts from OOram to treat frameworks as first-class concepts, calling it “role modeling for framework design.” Role models describe the interface between the framework and the programmer’s code; “free roles” represent the roles the programmer can implement to use the framework. Programmers should find frameworks with associated role models easier to comprehend since the complexity of the class models has been explained in terms of cross-cutting role models.

8.2.2 Precise Design Patterns, Code Ties

The Utrecht University design pattern tool [17], implemented in Smalltalk, allowed the creation of prototype-based design patterns and binding of these design patterns to source code. Conformance checking between the pattern and source code could be performed and predefined fixes to repair non-conformance. Modeling focused on design patterns and application of the tool to frameworks was not specifically explored. Conformance checking was limited to static program structure.

8.2.3 Cookbooks and Recipes

Confronting the challenge of communicating how to use the Model-View-Controller framework in Smalltalk-80, Krasner and Pope [41] constructed an 18 page cookbook that explained the purpose, structure, and implementation of the MVC framework. The cookbook begins with text but increasingly weaves in detailed code examples to explain how the framework could be used to solve problems. This cookbook was designed to be read from beginning to end by programmers and could also be used as a reference but every recipe did not follow a consistent structure nor was it suitable for parsing by automatic tools.

The work on Hooks by Froehlich et al. documents the way a framework is used, not the design of the framework [21]. Hooks are similar in intent to cookbook recipes but are more structured in their natural language. The elements listed are: name, requirement, type, area, uses, participants, changes, constraints, and comments. The instructions for framework users (the changes section) read a bit like pseudo code but are natural language and do not appear to be parsable by tools.

8.2.4 Patterns

Johnson appears to have been the first to suggest documenting frameworks using patterns. He notes that the typical user of framework documentation wants to use the framework to solve typical problems [36] but also that cookbooks do not help the most advanced users [37]. Patterns can be used both to describe a framework's design as well as how it is commonly used. He argues that the framework documentation should describe the purpose of the framework, how to use the framework, and the detailed design of the framework. After presenting some graduate students with his initial set of patterns for HotDraw, he realized that a pattern isolated from examples is hard to comprehend.

Design patterns themselves can be decomposed into more primitive elements [51]. Pree calls these primitive elements metapatterns and catalogs several of them with example usage. He proposes a simple process for developing frameworks where identified points of variability are implemented with an appropriate metapattern, enabling the framework user to provide an appropriate implementation.

The declarative metaprogramming group from Vrije University [65, 66] uses Pree's metapatterns [51] to document framework hotspots and defines transformations for each framework and design pattern. Framework instances (plugins) can be evolved (or created) by application of the transformations. The tool uses SOUL, a prolog-like logic language. The validation was done on the HotDraw framework by specifying the metapatterns, patterns and transformations needed. The validation uncovered design flaws in HotDraw, despite its widespread use, along with some false positives. The work does not detail how evolution of client code would be supported, instead focusing on the evolution of the framework.

A UML profile is a restricted set of UML markup along with new notations and semantics [19]. The UML-F profile provides UML stereotypes and tags for annotating UML diagrams to encode framework constraints. Methods and attributes in both framework and user code can be marked up with boxes (grey, white, half-and-half, and a diagonal slash) that indicate the method/attribute's participation in superclass-defined template patterns. A grey box indicates newly defined or completely overridden superclass method, a white box indicates inherited and not redefined, a half-and-half indicates redefined but call to `super()`, and a slashed box indicates an abstract superclass method. The Fixed, Adapt-static, and Adapt-dyn tags annotate the framework and constrain how users can subclass. Template and Hook tags annotate framework and user code to document template methods. Stereotypes for Pree's metapatterns (like unification and separation variants) are present, as are predefined tags for the Gang of Four [23] patterns. Recipes for framework use are presented in a format very similar to that of design patterns but there is no explicit representation of the solution versus the framework. The recipe encodes a list of steps for programmer to perform.

The FRamework EDitor / JavaFrames project [28, 29, 30] is a result of collaboration between The University of Tampere, the University of Helsinki, and commercial partners starting in 1997. They have developed a language for modeling design patterns and a tool that acts as a smarter cookbook, guiding programmers step-by-step to use a framework. With the 2.0 release of JavaFrames, the tool works within the Eclipse IDE. Their language allows expression of structural constraints, and their tool can check conformance with the structural constraints. Code can be generated that conforms to the pattern definition, optionally including default im-

plementations of method bodies. Specific patterns can be related to general patterns; for example a specific use of the Observer pattern in a particular framework can be connected to a general definition of the Observer pattern.

8.2.5 Languages

The Framework Constraint Language (FCL) [33] applies the ideas from Helm's object-oriented contracts [31] to frameworks. Like Riehle's role models, FCLs specify the interface between the framework and the user code such that the specification describes all legal uses of the framework. The researchers raise the metaphor of FCL as framework-specific typing rules and validate their approach by applying it to Microsoft Foundation Classes, historically one of the most widely used frameworks. The language has a number of built-in predicates and logical operators. It is designed to operate on the parse tree of the client program's code.

The Framework Specific Modeling Language (FSML) [3] is a domain-specific language targeted at the domain of frameworks. The framework is described as a collection of features with constraints between them. For example, feature A might be required for Feature B but incompatible with Feature C. Client programmers manipulate a program written FSML and the corresponding Java source code is generated by a tool.

8.2.6 Aspects

Aspect oriented programming seeks to improve the modularity of source code by localizing programmer-chosen concerns into their own input files [40, 48, 64]. For example, the parts of a program that deal with logging could be extracted to a new source file so they do not clutter up the main code. Design fragments and aspects share a similar desire to localize related parts of a program.

While design fragments are specifications, aspects are implementations. It may be possible to use aspects to provide default implementations for design fragments. Aspect languages such as AspectJ do not provide any specific mechanisms or advice on how to apply their technology to the problem of interacting with frameworks.

8.2.7 General Programming Assistance

The complexity of programming has long been recognized and attempts to help programmers manage that complexity have been researched. The Inscape Environment [49] focused on the challenges of evolution and scale in procedural programs that use libraries. It addresses these challenges in part through specification of interfaces, much like design fragments. The specification language was deliberately impoverished in order to avoid the tar pit of verification, again much like the desire of design fragments to maintain simplicity in its language to encourage adoption.

The Programmers Apprentice [54] was an attempt to apply artificial intelligence to the problem of programming by providing an intelligent agent to support the programmer. Cohesive collections of program elements are bound together into a cliché, similar to a design fragment

based on syntactic code structure, encoding roles and constraints. These clichés are used by the tool to aid the programmer.

8.2.8 Pattern Mining

Researchers have investigated the identification of design patterns in source code and models. Early techniques such as DP++ [4] created a set of structural templates and searched for matches. This was successful for structural patterns but could not distinguish behavioral differences, such as between the State and Strategy [23] design patterns, nor could it identify novel patterns.

SOUL (Smalltalk Open Unification Language) [43] added an enhanced predicate language to define the patterns, which enables more flexible querying through unification and backtracking operations. The patterns could refer to metadata, such as Smalltalk protocols (method categories). Scripts could be written to refactor code matching one pattern into another.

Newer techniques relax the need to statically define the patterns and are better at analyzing behavior. SPQR (System for Pattern Query and Recognition) [18, 58] can recognize pattern variants that had not previously been seen by using a theorem prover. Ferenc et al. [16] use machine learning and a corpus of manually tagged examples to reduce false positives and distinguish State and Strategy patterns. Streitferdt [60] has recently surveyed and evaluated pattern mining tools with respect to precision and recall.

8.3 Comparison to Design Fragments

Cookbook recipes, hooks, and design fragments are similar in that they all provide example-based descriptions of how to use a framework. Hooks added structure to recipes but was still natural language. Design fragments regularize hooks to make them tool-readable and enable tool-based assurance of constraints.

The declarative metaprogramming approach to modeling framework hotspots appears to have significant up-front investment before payoff in order to provide its guarantees about correct use of the framework. It may additionally assume a higher level of accuracy or correctness in frameworks than will commonly be found in practice. In [66], the authors comment that their approach specifically avoids design patterns in favor of metapatterns because there could be many design patterns. While this makes their technique generally applicable and composable, it will likely be difficult to add pattern-specific semantics and behavior checking to their approach because of this choice.

Though targeted at describing the full variability of the framework interface instead of known-good examples, the Framework Constraint Language could be used to improve the design fragment tool. A tool that used both could provide a rich constraint language, where design fragment authors who wrote their constraints in FCL could avoid writing a checker in Java code.

At first glance design fragments appear very similar to Riehle’s role models. But following the categorization from Section 8.1, it is clear that role models are first principles-based, while design fragments are example-based. A role model strives to describe the complete and abstract protocol that every set of classes conforming to it must follow, while design fragments describe

just a single legal use of that protocol. The difference is stark when considering a popular callback method, like `createPartControl` in Eclipse. A role model would describe how that callback could be used in every possible situation, while a design fragment describes just how it is used to accomplish a single goal.

In addition to this fundamental difference, there are differences of focus and intent. First, design fragments often span multiple framework role models. An example of this is the coordination of the role model for event registration (such as listening for mouse events) with the role model for lifecycle callbacks (such as registering for events during a particular lifecycle callback). Second, design fragments often encode actions outside of the framework, such as in the thread coordination design fragments, which would not be covered by any framework role model. Third, design fragments are asymmetric, so they define only what the programmer must do and only provide a programmer-centric view of what the framework roles are doing. However, the intent of both techniques is to aid programmers in using frameworks and in practice their strengths are complimentary.

Design fragments are very similar to JavaFrames in that both encode structural patterns that programmers use to engage with a framework and both enforce this through static analysis. Design fragments extend JavaFrames by adding a description of the relevant parts of the framework. This minimal description of framework resources respects the encapsulation boundary between the framework and the client, but informs the client programmer about the framework resources that client code must interact with.

This added description of framework resources enables two things. First, analysis tools that check for errors can take advantage of the descriptions of the framework – for example the protocol of framework callbacks can be specified and checked. This would be especially valuable in cases where direct analysis of the framework code is impossible because its source is unavailable, multiple implementations exist, or because its codebase is too large for the analysis tools to handle. Second, knowing why a recipe works enables the programmer to go beyond the recipe. Functionality demands from his problem domain will cause him to push on the limits of the pattern and he must be given an understanding of how this can be done. With exposure to many design fragments, the programmer will build up a mental model of how the framework acts on his code.

Chapter 9

Conclusion

This chapter integrates the evidence gained in the case studies and applies it to the three hypotheses we are investigating. These three hypotheses address the ability of design fragments to provide pragmatic help to programmers who use frameworks. We show how the evidence supports our main thesis, and discuss remaining concerns.

Design fragments provide a number of contributions to software engineering, including a new abstraction that represents client-framework interaction, a systematic technique to improve code quality, and an empirical understanding of client-framework interaction.

9.1 Validation

This dissertation has set out to demonstrate that design fragments and analysis tools can provide pragmatic help to client programmers who use frameworks. Pragmatism is a high standard, since a single tragic flaw can defeat it. Within the scope of this thesis, we cannot enumerate and discharge every possible challenge to practicality, so we have instead addressed the most visible challenges. We articulated these challenges in three hypotheses, each of which is directly testable.

Our first hypothesis is that design fragments can be used in real conditions, and the second is that there are limited numbers of design fragments. The third hypothesis is that conformance tools can be helpful.

In the following sections, we discuss each hypothesis and the evidence we have gathered to support it. We then turn to the main thesis and, reprising the six difficulties faced by client programmers, we show how design fragments help to overcome these difficulties.

9.1.1 Hypothesis: Real Conditions

Before our case studies were completed, we worried that the complexities of real code would render design fragments impractical. Consequently we set out to test the following hypothesis:

Design fragments can be used with existing large commercial frameworks, real programming languages, and off-the-shelf code.

We applied the design fragment technique by building catalogs of design fragments primarily for the Applet and Eclipse frameworks, and to a lesser extent for the AWT framework, Acme Studio, and our own tooling. In total we defined 39 design fragments, each representing a known-good way for a client to interact with a framework. Each client program that we looked at either contained an existing design fragment from our catalog or was the source of a new design fragment.

None of the frameworks or source code we examined was written for the purpose of exercising design fragments, so everything that we analyzed was off-the-shelf code. All of the source code was Java, a popular commercial object-oriented programming language. The Eclipse framework, in particular, is a large commercial framework, and the Applet framework, while smaller, was widely used in the late 1990s.

We bound the design fragments to client programs that we collected. We used client programs provided by the framework authors, and client programs that we found via internet searches. Over fifty Applets were examined and every use of a design fragment in them was bound or added to the catalog. Across the Eclipse client programs, over a hundred design fragment instances were analyzed from more than fifty client programs. This application of design fragments to over a hundred off-the-shelf client programs demonstrates that they can be used on real source code.

To verify that our design fragment language could express the client-framework interactions, we created a classification system for client-framework interactions by analyzing a selected Eclipse framework client program. We identified nine types of interaction, and found that all nine could be expressed in the design fragment language.

Knowing that our language could express these interactions still left open the possibility that client programs bound to design fragments would only have a small fraction of their interactions explained. After fully binding a selected Eclipse client program to design fragments, we found that 78% of the client-framework interactions were required by the design fragments. This rose to 98% when both required and optional interactions were counted. Either number indicates that design fragments can explain the majority of client-framework interactions.

Despite our positive experiences, we did find room for improvement in our language, tools, and conformance assurance. These desired improvements, detailed in Section 9.3, are not large enough to render design fragments impractical.

In summary, we successfully built design fragment catalogs for some commercial frameworks, bound client programs to design fragments in the catalogs, demonstrated that all of the structural client-framework interactions can be expressed, and that the design fragments in our catalogs can cover the majority of the client-framework interactions. These results support the pragmatism of design fragments in that they can work outside of laboratory conditions, and they can handle the complexities of commercial frameworks.

9.1.2 Hypothesis: Design Fragment Variety

Before starting our case studies, we were concerned that it would be difficult to create a catalog of design fragments that covered the different ways that clients used frameworks. Lacking empirical data on how client programs interacted with frameworks, we feared that each program would

interact slightly differently. High variety could mean that there were too many design fragments to count, rendering our approach impractical.

We hoped that our empirical studies would show that the variety was low, and that client programs followed repeated patterns of client-framework interaction. Consequently our second hypothesis is that:

The variety of design fragments to accomplish a given goal is limited, so a small catalog of design fragments can have good coverage of the code seen in practice.

A catalog of design fragments is built by examining several client programs, recognizing that they are similar in how they interact with the framework, and then documenting that interaction as a design fragment. Catalogs grow in size as more client programs are analyzed, revealing new design fragments to be documented.

The earlier a catalog reaches a useful size, the more pragmatic it will be to build it. For the catalog, an asymptotic growth rate indicates that the variety of client-framework interactions is low, since fewer new design fragments are being discovered as client programs are examined.

To validate this hypothesis, we created a full catalog for the relatively small Applet framework. We populated the Applet catalog using design fragments found in the twenty demo Applets provided by the framework authors. Using just these revealed 81% of the eventual size of the catalog. We surveyed an additional three dozen Applets from the internet and found just two more design fragments. One of those two was a design fragment that has a threading bug, something that was present in older versions of the demo Applets, but had been fixed in the demo Applets we examined. We conclude that the rate of growth of the Applet catalog was asymptotic, and that creating a complete catalog of design fragments was pragmatic.

We also built a catalog of design fragments for the Eclipse framework. The large size of the Eclipse framework made the creation of a full catalog too time consuming within the scope of this dissertation, so we collected other metrics to demonstrate that design fragment variety was low. We created design fragments from a client program provided by the framework authors, and then examined how closely client programs from the internet conformed to those design fragments. Three quarters of the design fragments were completely consistent between the demo and the internet client programs, and the remaining quarter of the design fragments were over 50% consistent. This data indicates that, as with the small Applet framework, design fragment variety for the large Eclipse framework is low.

We believe that this low variety is, in part, due to a *seed crystal* effect: programmers reference the provided demo client programs, and copy or recreate the solutions they find there. We found evidence for this behavior both in the Applet case study, where structure and variable names were copied without changes, as well as in the Eclipse case study, where structure and method names were copied unchanged.

After examining two different frameworks and over a hundred client programs, the evidence strongly suggests that client programmers are repeating patterns of client-framework interaction, which is sufficient evidence that a pattern-based approach like design fragments is pragmatic.

9.1.3 Hypothesis: Assurance

The previous two hypotheses supported the pragmatism of design fragments. Our third hypothesis focuses on the ability of analysis tools to help programmers:

Analysis can provide programmers with assurance that their code conforms to the constraints of the framework as expressed in design fragments.

We implemented three simple static analyses to demonstrate the principle of conformance assistance. The first of these, *required method call*, was the most common client-framework interaction in our survey, accounting for 43% of all such interactions. The *required new instance* analysis was also common, accounting for 13% of such interactions. The third analysis, *required class reference in XML*, provides an important bridge between Java source code and required external XML configuration files.

All three of these specifications were added to the set of design fragment specifications using our standard extension mechanism. Furthermore, the checkers for them are implemented using the tool's standard interface for analysis extensions. Each analysis has access to the data structures maintained by the design fragments tool plus the abstract syntax tree of the client program. Because of this, we believe that other analyses should be able to extend the language and tooling in the same way.

In addition to these three implemented analyses, we have extended the design fragment language with other specifications but have not yet implemented the analyses. Through our case studies we have collected other examples of constraints imposed by the frameworks that we believe could be checked through analysis tools. These include protocol, concurrency, post-conditions on state, return values, and other constraints.

Overall this evidence shows that it is possible to assure conformance between the client's source code and the specification of the design fragment. Furthermore, it shows that there are opportunities, in the form of additional framework constraints, for analysis tools to provide assurance beyond what has been implemented.

9.1.4 Thesis

The main thesis of this dissertation is:

We can provide pragmatic help for programmers to use frameworks by providing a form of specification, called a design fragment, to describe how a client program can correctly employ a framework and by providing tools to assure conformance between the client program and the design fragments.

In Chapter 2, we identified six difficulties that client programmers face when using frameworks: they find it difficult to understand non-local client-framework mechanics, design solutions when they do not own the architectural skeleton, gain confidence that they have engaged with the framework correctly, represent their successful engagement with the framework in a way that can be shared with others, and ensure their design intent is expressed in their source code. Design fragments help overcome these difficulties.

Understand non-local client-framework mechanics. A client programmer with a goal can look up an appropriate solution defined as a design fragment. This design fragment would de-

scribe all needed client-framework interactions despite any non-localities. A client programmer examining source code can use the bindings and design fragment tools to navigate to the various parts of the code that implement parts of the design fragment and can of course look at the bound design fragment itself, both of which help the client programmer to understand what the code does and locate all the relevant client-framework interactions.

Design solutions without owning the architecture. By defining the architecture of the program, the framework constrains the programmer's solution space and it therefore can be difficult to find a solution. Based on our case studies, in most cases the client programmer should be able to find a known-good solution in the catalog, which avoids the search for a solution, replacing it with the easier job of scanning the design fragment catalog.

Gain confidence of compliance. Client programmers using design fragments have increased confidence that their code interacts with the framework correctly, first because they know they are reusing a known-good solution, and second because analysis tools assure them that their code has the needed client-framework interactions. Client programmers who invent their own solutions worry that they have not handled all the necessary cases or, through an unusual interaction, have revealed obscure framework bugs. The assurances provided by the conformance tools will likely never be complete, but more sophisticated analyses will yield increasing assurance, and some categories of errors can be eliminated completely.

Represent solutions. Design fragments are a compact and tool-readable representation of a known-good solution and as such are an effective way for programmers to communicate their solutions to others. Programmers could additionally provide an example program that is bound to the design fragment, which has all of the benefits of current best practice along with greater assurance that other programmers will find all non-local interactions and conform to framework constraints. Example programs alone run the risk that other programmers will infer the solution incorrectly, as was seen in the Applet demos where the demos deregistered for events in the `destroy` method, but client programs from the internet failed to conform to this non-local constraint.

Encode design intent. Design fragment bindings in the source code are an enduring expression of the client programmer's design intent. He is saying that his intent was to use the known solution defined by the design fragment he names. Other programmers who evolve the code will not have to guess what his code is intended to do and will feel more confident in removing unneeded cruft from the program rather than leaving it in for fear that it breaks something they do not understand.

Connect with external files. Design fragments allow the direct expression of what should be in external XML files. Three specifications that apply to XML files are provided, and the specification language can be extended. Programmers applying a design fragment can be confident that they are creating both the required object-oriented code as well as configuring external files.

We have shown that design fragments are pragmatic in that they work on real code, real languages, and real frameworks. Client programmers follow regular patterns of client-framework interaction, which means an example-based approach like design fragments is pragmatic since the size of design fragment catalogs grows asymptotically. Design fragments are helpful in that they document how to interact with a framework, but will become additionally helpful as analysis tools check the constraints specified in the design fragments. Design fragments and tools help overcome the difficulties faced by client programmers when using frameworks.

9.2 Pragmatism Concerns

Despite the evidence gained in the case studies, concerns remain about pragmatism of design fragments. In the following sections, we discuss concerns that could limit the pragmatism of design fragments. They are:

- Design fragments will not be useful to all programmers, since different programmers solve problems in different ways
- Domain-specific languages would provide more value
- There may be little opportunity for more specifications
- The expression cost to bind design fragments is high
- Design fragments offer no composability guarantees
- The design fragment language has unfortunate expressability limits
- A design fragment catalog may have a long tail
- Design fragments only work for the Java language

Some concerns may be addressed through improvements in the tools or language, others will require more information, gained from programmers who use design fragments and report on their experiences.

9.2.1 Programmer Differences

Not all programmers solve problems in the same way. A usability group at Microsoft has invented three programming personas to help them understand which features in their development tools will appeal to which programmers [8]. These three personas are defined in [7] as follows:

- The “opportunistic” programmer, representing the majority of all programmers, likes to create quick-working solutions for immediate problems and focuses on productivity and learns as needed.
- The “pragmatic” programmer likes to create long-lasting solutions addressing the problem domain and learns while working on the solution.
- The “systematic” programmer likes to create the most efficient solution to a given problem, and typically learns in advance before working on the solution.

In Table 9.1, we have summarized how we expect these three personas would value the different kinds of information on frameworks.

We expect that novices of all types would find design fragments useful both as a bridge to becoming experts, and as a tool to ensure correct implementation of their client-framework interactions. While the opportunistic programmers may be satisfied with this level of help, pragmatic and systematic programmers will likely seek out the other sources of information. Experts who understand the framework well may already know the pattern expressed in the design fragment, but will value the conformance assurance and the consistency that comes with using it.

	Opportunistic	Pragmatic	Systematic
Framework source code	No	Maybe	Yes
API documentation	Maybe	Yes	Yes
Books and courses	Maybe	Yes	Yes
Targeted tutorials (HowTo's)	Yes	Yes	Maybe
Example client programs	Yes	Yes	Maybe
Design fragments	Yes	Yes	Maybe

Table 9.1: Helpfulness of information to personas

9.2.2 Domain-specific Languages

Domain-specific languages [6] (DSL's) enable the succinct expression of abstract concepts particular to the domain. A DSL can be designed such that all programs written in it conform to the framework constraints. The value of the DSL is unlocked when tooling enables programs written in the DSL to be compiled into client programs of the framework. DSL's are unlike tutorials and design fragments in that DSL's are of the “first principles” style, rather than the “example” style, in that they do not encode known-good examples. DSL's for frameworks can benefit the entire software lifecycle, yielding high reward.

Since the DSL encodes the constraints particular to a framework, a new DSL and new tooling would need to be created for each framework. A DSL for a framework would be intellectually challenging to build because it requires knowledge both of language design and of the framework. Consequently, DSL's are a high-value, high-cost choice.

Tutorials, or other forms of documentation, are inexpensive to produce. They are expected to contain accurate instructions but these instructions can be written informally in natural language. They may or may not describe why the instructions work or explain the constraints the framework places on the resulting client program. They only benefit the client programmer during the software construction time since their instructions cannot be read by analysis or automation tools. Consequently, tutorials are a low-value, low-cost choice.

Design fragments sit in a sweet spot between DSL's and tutorials because they provide many of the benefits of a DSL with its tooling, at a cost that is closer to the tutorial. Design fragments are tool-readable and consequently tolerate typos less well than tutorials, but it is easy to imagine a design fragment editing tool that catches many of these syntactic typos. More of an intellectual challenge is the recognition and expression of the pattern, but this challenge is substantially lower than that of designing a DSL. We expect that, as with tutorials, all framework authors and most client programmers would be able to write a design fragment. Unlike DSL's, where each framework's DSL would require custom tools, the design fragments language and tools can be used across frameworks with few changes to either. Some frameworks may spur new kinds of design fragment specifications or analysis tools, but once these are written they are likely to be reusable on other frameworks. So, for about the same cost as writing tutorials, framework authors can instead write design fragments and get benefits close to a custom DSL tool chain.

One reason that this sweet spot exists is that a design fragment can afford to be an incomplete specification. Design fragments are not a full functional specification that might be usable by a tool to generate source code. Instead, design fragments connect many existing dots, dots that

include method- and class-level API documentation, the client programmer’s experience with other frameworks, and the client programmer’s general programming skill. The design fragment provides enough detail for the client programmer to create a working program and for tools to check that constraints have been met.

Unlike some other formalizations, including DSL’s, design fragments have a gentle adoption slope and provide value quickly, with just one design fragment in a catalog. It is possible to “dip your toe in the water” and see how the technology works with a small commitment of time and energy. Furthermore, anyone with access to the source code can do this and see the results. Each incremental investment in either a new design fragment or a binding to source code yields value. It does not require deferred gratification that only comes when an entire program is annotated or the catalog is complete.

9.2.3 Specifications Opportunities

We have already created some specifications and written some analysis tools to check them, and we have claimed that programmers will benefit as more specifications are written and more checkers are built. However, what if frameworks impose few additional constraints? In that case, there would be little additional benefit to be unlocked.

To find out what other specifications were possible, but not yet included in the design fragment language, we examined the freeform specifications that currently exist as unstructured text in the design fragments we have written. We collected data from all of our design fragment catalogs, including the partial ones: AcmeStudio, Applet, AWT, DesignFragmentSpecExtension, Eclipse 3.2, and Runnable. We then grouped the specifications into categories, discarding the ones that were in essence comments, not specifications. The result was five categories of specifications: protocol compliance, concurrency, state post-conditions, return values, and other.

This list should not be regarded as complete since it was derived from a sampling of the freeform specifications, and those freeform specifications represent only the constraints that were noticed by the design fragment author. Please note that I was the primary author of the design fragments, and also the one who later analyzed the freeform specifications, though the analysis was performed as much as three years after the specifications were written.

Protocol Compliance. Somewhat surprisingly, we noted just one framework interface where the client must observe a sequence when invoking framework service methods. In an implementation of the Visitor design pattern [23], the Eclipse framework provided a state checking method that could only reasonably be called after the client had first invoked `visit` on the visitor.

Again in the Eclipse framework, the sequence of callbacks can be changed by the client based on the return value of a prior callback. The framework invokes `hasChildren` on the client and only if the client returns true does the framework subsequently invoke `getChildren`.

Concurrency. Since the Applet demos dealt heavily with concurrency we have a number of examples from that framework. The Eclipse framework also works with concurrency but the programs we examined did not use this part of the framework. The freeform specifications from the Applet demos gave two kinds of advice to client programmers. The first regards the nature of the coordination between the threads; the second regards the looping or execution of the background thread.

Some threaded Applets communicated with the background thread by manipulating the values stored in fields which were then read by the background thread. Other Applets did not so the threads could not communicate. We saw freeform specifications advising the client on the availability and nature of this communication channel.

When the background thread is invoked, it may need to execute once, or loop, or execute only if certain conditions are satisfied. We saw freeform specifications advising the client on this expected behavior, for example, *Only run when roleThread == Thread.currentThread ()* and *Do domain-specific thread processing once only (no while loop forever)*.

Post-conditions on State. Frameworks provide callback methods to provide clients with changed or new objects and the client often stores these objects for future use. We also saw cases where Applet fields must be set to a particular value to ensure correct communication with background threads.

Return Value. We found more freeform specifications in this category than any other. In order to correctly implement a design fragment it was often necessary for the client to return particular values to the framework. Although these specifications follow the standard format of a method's functional specifications, they are only a subset of what the method must do, in effect identifying just one case that must be handled.

Other. In the Applet framework, every method call that checks a parameter value must be paired with an entry in an array describing which parameters are checked. We discuss this simple specification in depth in Appendix A, where we walk through how a specification extension could be built.

Another piece of advice involved when downcasting was safe. As seen in the example from Chapter 1, in the Eclipse framework, a client can register for `ISelection` events but in some cases it is safe to downcast the object to an `IStructuredSelection`. In the context of a particular design fragment, such a downcast might always be safe.

Some of these specifications may be checkable using existing analysis implementations like Fluid [26] and ESC/Java [9], in particular the concurrency, state post-conditions, and return value specifications.

From this analysis, it appears that frameworks impose many constraints beyond those already encoded as specifications in the design fragments language, so there is still considerable opportunity beyond what has been implemented already.

9.2.4 High Expression Cost

The expression cost of design fragments is high, specifically the effort required to bind design fragments to code. A design fragment has multiple roles and each role must be bound to the source code using a verbose Java annotation. Logistically, writing binding annotations is a burden for programmers. Design fragment authoring is also time consuming, but that burden is borne by relatively few compared to binding. The concern is that this high expression cost will deter programmers from using design fragments.

Conceptually, the job of binding a design fragment to source code is simple, since it is just showing correspondence between the two. To reduce the burden, we need to make the programmer's job no more difficult than showing this correspondence. We anticipate three ways to reduce this burden.

First, we can improve the design fragment language to reduce the number of needed bindings. For example, if a method must have an exact name because it overrides a superclass method or implements a method from an interface, no binding should be needed but currently the programmer must provide it. We chose Java annotations because they are the standard way of annotating Java programs but this has led to verbose bindings. We could instead put the bindings in Java comments and reduce the verbosity and complexity of the bindings. We should also allow the annotations to be placed into a separate file when it is inappropriate or impossible to put the annotations into the source code.

Second, tools can write the bindings for the programmer. We envision a “binding wizard” where each role in the design fragment is displayed, and the programmer can select from a list the corresponding part of the program to bind to. The wizard should be able to suggest methods that conform to the constraints of the design fragment, or ones that match the required return value and signature of the method role. The JavaFrames tool [29] can create skeleton code where no class or method yet exists, and our binding wizard should be able to do this too.

Third, tools can be built to analyze an existing code base and offer to bind the design fragments that it finds. Such a tool is conceptually related to design pattern mining [58]. A mining tool would allow large existing projects to quickly begin using design fragments without a long and uninteresting period spent binding.

To quantify the costs and benefits of annotations and specifications, we have compiled data on what client programmers must provide and what they are provided. Table 9.2 lists the design fragments from the Applet and Eclipse catalogs with the number of required annotations and provided specifications. The number of required annotations is divided into three categories. The *current* category represents the number of annotations programmers must write with the design fragment tools today. The *forced* category assumes that programmers should not have to write annotations where the choice is forced, for example the `createPartControl` callback must have that name exactly, so the annotation can be omitted. The *inferred* category assumes that we can further reduce annotations by guessing based on provided type information, for example a `roleViewer` field must be of type `TreeViewer` and there is only one field that matches that type. Specifications are divided simply into *checked*, which includes specifications like required method calls, and *total*, which additionally includes freeform specifications.

This analysis of the costs and benefits of providing annotations is quantitative, but flawed. It is easy to count the number of provided specifications, but difficult to count the benefit of captured design intent, or of focused attention. This analysis also fails to credit learning how to do something correctly: the Background Continuous V1 and V2 design fragments look equivalent in this analysis, but benefit has clearly been provided in the transition from buggy V1 to bug-free V2.

Despite these flaws, we can extract some general trends from this data. One is that for the “listener applet” design fragments, there is a large reduction in annotation burden in the *forced* category. In these design fragments, the programmer is asked to implement a specific interface and its methods in order to receive events from the framework. In the current system, annotations must be put on each of those methods, but since the programmer is forced to use those exact names, we can omit those annotations. This reduction in work is welcome, because each of these design fragments has just two checked specifications, indicating during what callback to register and de-register.

	Number of Annotations			Specifications	
	Current	Forced	Inferred	Checked	Total
Applet					
Background Continuous V2	7	4	3	3	6
Component Listener Applet	9	3	3	2	2
Focus Listener Applet	7	3	3	2	2
Key Listener Applet	8	3	3	2	2
Manual Applet	3	2	2	5	5
Mouse Listener Applet	10	3	3	2	2
Mouse Motion Listener Applet	13	7	7	2	2
One Time Init Task	6	3	3	2	5
One Time On Demand Task	6	5	4	2	5
Parameterized Applet	4	2	2	1	2
Timed Task	7	4	3	2	4
Applet Total	81	39	36	25	37
Eclipse					
Content Provider	11	4	3	3	7
Double Click Action	10	8	7	6	6
Drilldown Adapter	7	6	4	3	3
Label Provider	7	4	3	2	2
Popup Dialog	3	3	3	3	3
Pulldown Menu Action	9	7	7	11	11
Resource Listener	8	4	3	5	6
Right Click Menu	8	6	5	12	13
Shared Action Image	3	3	3	4	4
Shared Image	3	3	3	3	3
Toolbar Menu Action	9	7	7	11	12
Tree View	8	6	5	5	6
Tree View Selection	6	4	2	2	3
Viewer Sorter	5	4	3	2	2
Eclipse Total	97	69	58	72	81

Table 9.2: Cost and benefit of annotations

Another trend is that the benefit derived from annotations corresponds roughly to the effort. There are some outliers, for example Mouse Motion Listener Applet requires seven annotations but provides just two specifications, and Right Click Menu requires just five annotations but provides twelve specifications. Overall, the Applet total is 36 annotations to get 37 specifications, and the Eclipse total is 58 annotations to get 81 specifications.

9.2.5 Composability

Design fragments offer no composability guarantees. That is, a programmer who uses design fragments A and B has no guarantee that they do not conflict in subtle ways. It is also true that even a single design fragment is not guaranteed to work – the programmer could have written an infinite loop, or another debilitating bug, in another part of the program that prevents the correctly implemented design fragment from working. Furthermore, within a method, a design fragment gives no guidance as to how to interleave the required parts of two design fragments.

Design fragments are partial specifications. They rely on the skill of programmers to implement solutions and resolve conflicts at a local level. Design fragments identify the relevant places in the source code a programmer should look to resolve a bug, making it easier for them to find the places to look, and making it easier for them to understand the non-local interaction. In our experience, these composability concerns have been minor because programmers are able to resolve any conflicts at this local level. Design fragments have the same composability problems as any recipe-like solution, including natural language tutorials and books.

Additionally, while no guarantee exists, design fragment authors should strive to create catalogs where all the design fragments are indeed composable, much the same way that class libraries are built with classes that are known to work together.

9.2.6 Expressiveness Limitations

The design fragment language has a number of expressiveness limitations, some of these can be worked around using comments in the design fragment. This approach is unsatisfactory from a specification perspective, but may be sufficient from a usability perspective.

Over-specifying a superclass. A design fragment may specify that the client must subclass from the `ViewPart` class, or it could specify `ViewPart`'s parent, `WorkbenchPart`. If most of the time clients will be subclassing from `ViewPart` then authors will be tempted to write this constraint into the design fragment. The design fragment could specify the more accurate constraint of subclassing from `WorkbenchPart`. Being more accurate, however, works against a design fragment's ability to point the programmer to relevant resources, which in this case would be `ViewPart`. It is possible to add a comment in the design fragment advising the programmer about the existence of possibly more appropriate subclasses.

Over-specifying fields or parameters. In a design fragment, it is easy to specify that the client must have a field, or use a parameter, but it is not possible to allow either in an interchangeable way. The intent might be “the source code must hold onto this object from the framework for later, either in a field, a local variable, or just pass it around as a parameter”. Consequently there are cases where the design fragment over-constrains the client programmer.

Over-specifying code blocks. While the set of constraints is open ended, it is not straight-forward to specify the case where large blocks of code are part of a design fragment. While it is easy to say that the client programmer should invoke methods A, B, C, D on the framework in sequence, this will over-constrain the client programmer when some interleavings are also allowed, such as B, A, C, D. Trying to encode all such legal variations is contrary to the main idea of design fragments: to encode a single known-good solution. Perhaps the pragmatic choice would be to encode a tightly constrained set of solutions, with the sequence of methods encoded using a path expression.

Optional parts. Design fragment authors often want to advise client programmers that some items, while not required, are commonly present. An example of this occurs when creating a `Menu` item in Eclipse: it is required that an `Action` be initialized and placed into the `Menu`, but it is optional that the `Action`'s icon be specified. Currently the design fragment language cannot express such optionality, but we encode it informally using comments.

Binding roles multiple times. Programmers may desire to use a design fragment multiple times without rebinding every role. For example, a `Menu` may have dozens of `Actions` in it. It is tempting declare just one design fragment instance, with the role for the `Action` bound multiple times. This saves the client programmer the effort of binding the parts that stay the same, such as the bindings to the `Menu` role. Another option is to have smarter tools that aid in binding, which would keep the current single-binding system, but reduce the programmer's effort to express the bindings. A tool could offer the option of cloning most of the bindings for an existing design fragment instance, and then guide the programmer to bind the role for the `Action`.

Unavailable source code. The source code to be annotated with bindings is not always available for editing. This can happen when a framework class plays a role in a design fragment, yet that code is read-only. This problem with annotations has been addressed by the Fluid [26] project where the in-code annotations are augmented by “standoff annotations” that can be placed in a separate file. Such an option allows programmers to choose the appropriate solution for their code but is not yet implemented in the design fragments tools.

9.2.7 Long Tail of Design Fragment Catalog

Some distributions have a long tail [2] where the majority of the area under the curve falls under the tail. The implication for design fragments is that a catalog containing the most popular design fragments may be missing the majority of them. Programmers looking in the catalog may not find what they are looking for.

Our data from the Applet catalog indicates that a long tail is either not present, or does not hurt the utility of design fragments. After examining the twenty demo Applets provided by the framework authors, we found just two additional design fragments after examining thirty-six Applets from the internet, and one of those was deprecated. The new, non-deprecated design fragment was used in just one Applet. So, after creating the catalog from the initial twenty Applets, we were able to describe thirty-five of the thirty-six Applets using our existing design fragments. For the Applet framework, a client program appears likely to be well served by our catalog.

Since we have not created catalogs for many frameworks, it is possible that a long tail distribution exists for them, and it will impair the usefulness of the catalog. If the long tail contains unnecessary diversity, that is, variants of existing design fragments, a programmer may be equally well served by a variant already in the catalog that accomplishes the same goal. Also, a client program with only half of its client-framework interactions bound to design fragments still enjoys the benefits of design fragments, since, as discussed in Section 4.5, the benefits of design fragments accumulate with each one that is bound.

9.2.8 Java Language

Our case studies and tools all work exclusively with the Java language. There is a concern that design fragments have only been shown to work with this one language. However, the most popular object-oriented languages today, such as Java, C++ and C#, have quite consistent features, especially those features relevant to client-framework interactions. This core set of concepts includes classes, fields, methods, inheritance, and interfaces. The design fragment language directly expresses these core concepts, as shown earlier in Figure 4.2.

The C++ language [61] does not have interfaces, but does have *multiple inheritance*, and an abstract class can be used for the same effect as an interface. However, a C++ framework that relied upon multiple inheritance to provide methods would go beyond our core features. C++ has some different ways of providing access, such as *friend* classes. C++ also has the features of C, including *function pointers*, that could be used by a framework.

The C# language [46] is quite close to Java, but like C++ it has a few additions. *Delegates* work similarly to function pointers in C++, and *properties* allow the difference between fields and methods to be blurred. Like Java, C# has *attributes*, which enable declarative metadata to be associated with program elements.

Because of the similarity of concepts between the mainstream object-oriented languages, the design fragment language should already be able to handle many C++ and C# frameworks. With a few additions to its core concepts, would be able to handle all of them. Some Java frameworks, including the Enterprise Java Beans 3.0, now require the use of annotations, so that should be the first addition to the core concepts.

9.3 Discussion

Looking back across our experiences with design fragments, four items are worthy of discussion. The first is that framework authors, using existing object-oriented mechanisms, still have difficulties encapsulating some private design details within the framework. The second is the observation that demo client programs have a virtuous effect in that they reduce the variability of client-framework interactions. The third is our advice to framework authors, who we believe should deliver a large set of demo programs, sketch design fragments to rate the complexity of the client-framework interactions, and, deliver a catalog of design fragments for their framework that is bound to their demo programs. Finally, we have initial evidence that it is possible to refactor frameworks so that client code is more assurable.

9.3.1 Framework Encapsulation

The essence of encapsulation is that components publish details (API's) that other components can depend on; other details are private and may change. Today, framework authors have a limited ability to publish details of what to depend on. Since frameworks are built from object-oriented parts, they can mark classes and methods as private, and they can use interfaces to hide implementation classes. But many framework constraints involve object lifecycles and invocation protocols, and these cannot be expressed except through natural language documentation. When programmers infer these constraints by reading other client programs, they may make mistakes.

Consequently, Applet client programmers have accidentally or deliberately relied on an internal detail of the Applet framework, specifically, that there is no runtime error if you fail to deregister for events. This is a private detail of the framework implementation and should be subject to change. However, it is hard to imagine that the Applet framework authors can change this private detail, since our data indicates that between 90% or more of Applets on the internet (see table 6.2) rely on this detail. Note that similar failures to deregister would, in other frameworks, cause runtime errors.

The demo programs distributed with the Applet framework appear to have failed to communicate this framework constraint, possibly because the client-framework interaction is non-local, and consequently client programmers inferred the requisite client-framework interaction incorrectly. Design fragments can help framework authors to clearly articulate known-good solutions, and ensure that programmers do not miss non-local details. Design fragments help because they make non-local interactions obvious, and because tools can flag non-conformance.

9.3.2 Examples as Seed Crystals

In our examination of large numbers of client programs found on the internet it became clear that the example programs provided by framework authors were acting as seed crystals, causing client programmers to interact with the framework in the same way the example did. The evidence for this is syntactic correspondence between the example and internet programs, including identical field names, identical method names, and identical bugs. With the exception of efficiently propagating bugs, we believe that the seed crystal effect is completely virtuous.

Frameworks generally have large API's, which means many different ways that clients could accomplish their goals. Not all of these ways were anticipated by the framework authors and some interactions may reveal latent bugs in the framework. Sticking to the known-good solution means less chance of revealing these framework bugs, which in turn means more reliable software.

The seed crystal effect also reduces the diversity of client programs. This is a boon during evolution of programs since a programmer new to this code who knows the framework-provided examples will already know both the intent and implementation of the code just by recognizing the field or method names. Evolution and maintenance should be easier and more accurate.

Design fragments themselves can be seen as a formalization of the seed crystal effect. Not only does the code imitate its seed crystal, this intent is declared in the code and assured through

tools. Catalogs are an even more concentrated source of these seed crystals than example programs, so they may be more effective at reducing diversity.

It is likely that design fragments have been shown to work well partly because both the catalog and the client programs we analyzed derived from a common ancestor: the example programs provided by the framework authors. It would be a mistake to view this as a liability of the technique. It is instead a reflection of real world programming practice in which the design fragments technique must show its value, and no modern framework is delivered without at least some example programs. One such example program, the Java Pet Store [62], has become so famous that it has been used as a benchmark between frameworks [45].

9.3.3 Advice to Framework Authors

Given the virtuous seed crystal effect that example programs can have, we recommend that each new framework be delivered with a set of example programs. These example programs should exercise most of the framework interface, and will reflect the client-framework interactions that are anticipated by the framework authors. Of course we believe that these example programs should also be bound to a catalog of design fragments.

Even if they do not formalize them into a catalog, framework authors can still benefit from sketching their expected client-framework interactions using the ideas of design fragments. Doing this will yield useful insights into the interface they have designed, including how to refactor the interface for simplicity, ease of use, and accuracy.

If they change nothing about their interfaces, framework authors who create a catalog of design fragments will have made it easier to create client programs. Another option is for authors to create more complex interfaces, providing client programmers with more power and flexibility, and use design fragments to help manage the increased complexity.

9.3.4 Framework Refactoring Opportunities

It should be possible to refactor frameworks such that client-framework interactions are more assurable. Framework authors always make trade-offs in the design of their frameworks. Now that design fragments and some analyses exist, framework authors could design their frameworks differently. Not all framework constraints can be analyzed by the existing tools, so framework authors should prefer to impose those constraints that can be checked.

We experienced one example where the framework could be refactored, yielding client code that was more assurable by our analyses. In this example, the client code must override the framework select method, which is invoked when an object is selected:

```
public boolean select (Object o);
```

The framework passes the object that is selected to the client program, and the client program tests the object, returning a boolean value based on the test. If the object is an `EditPart`, the client must extract the model from it before performing any tests. Before the refactoring, client code extracted the model itself:

```
public boolean select (Object o) {  
    if (o instanceof EditPart) o = ((EditPart )o).getModel ();
```

```

        return // some test on the object
    }

```

Our provided set of specifications does not yet express checks for `instanceof`, but we can check that the client code invokes a framework service method. The refactoring introduced a new framework service method called `unwrapToElement` that performed the testing for `EditParts` and the extraction of the model. This yields equivalent client code, but the new client code can be checked through off-the-shelf analyses, since every client can be required to call the framework service method:

```

public boolean select (Object o) {
    o = EclipseHelper.unwrapToElement (o);
    return // test on o
}

```

This refactoring has the benefit that the code is slightly simpler and that tools can easily assure conformance. Aiding framework evolution, it is now possible to add new tests or conditions to `unwrapToElement` without disturbing existing client code.

When a client-framework interaction is not currently assurable, it is possible to either increase the expressiveness of the language or refactor the framework. While it might seem that framework authors would always avoid refactoring, the above example shows that complexity in the client code can be hoisted into the framework and also provide evolution benefits to the framework authors.

9.4 Contributions

Through its case studies, this work has directly demonstrated three primary contributions to software engineering.

First, it provides a new technique to help programmers use frameworks. Design fragments and the tools directly address the problems we have identified with frameworks. Design fragments improve on previous abstractions by describing relevant resources in the framework, connecting object-oriented code with external declarative configuration files, and by enabling new constraints and tools to be added to our predefined set. This helps client programmers who are creating new code and who are evolving existing code.

Second, it provides a systematic way to increase code quality. Design fragments provide a means to communicate known-good designs to programmers, and, unlike simple copying of examples, a means of influencing the uses of that design so that revisions can be propagated. Bindings between design fragments and client programs express design intent, and this enduring expression of intent enables conformance assurance from both existing and future analysis tools. This helps client programmers, and also helps writers of analysis routines because the stated design intent removes the need to infer intent.

Third, it provides an empirically-based understanding of how clients use frameworks, which aids researchers in choosing research directions and aids framework authors in delivery of new frameworks. Specifically, the observed variety of client-framework interactions is very low, despite the opportunity for high variety.

In addition to the contributions demonstrated through our case studies, we expect this work to enable other benefits to software engineering.

Design fragments may significantly improve a programmer’s ability to manage complexity. Design fragments make non-local code interactions evident, and analysis tools help prevent code evolution from breaking code that is bound to design fragments. Both of these improve a programmer’s ability to manage complexity. As a consequence, programmers may be able to build more complex programs, or framework authors may design more complex frameworks, since they know that design fragments make the added complexity manageable.

Design fragments may enable a new technique to audit and improve the design of existing frameworks, yielding reduced complexity and increased assurability of client-framework interactions. Design fragments themselves can be analyzed for complexity through an inventory of the interactions they specify. Framework complexity could be improved by reducing the complexity of the design fragments it employs. Design fragments may form the basis for metrics to quantify the complexity of client-framework interaction.

9.5 Future work

Our work on design fragments suggests that continued work in the area will be fruitful. In the short term, we believe that improvements in the design fragments language, tools, and analysis will reduce its burden and increase its benefits.

For the design fragments language, some minor changes could have large impacts. Programmers currently have to provide bindings even when the binding could be inferred. For example, the `init` method is a template method defined by the `Applet` superclass, and consequently its name cannot be changed. The design fragments language should express and recognize situations like these, and not require programmers to write bindings. In addition, the design fragments language could encode relationships between design fragments. A design fragment can have pre-requisites or co-requisites, and it can also be related to a more abstract pattern, like the `Observer` pattern [23]. The design fragment language could express these relationships, which would provide a richer understanding to users, as well as enabling the pre-requisites and co-requisites to be checked by tools.

Other tool improvements would reduce programmer effort to use design fragments. Providing tools to facilitate binding a design fragment to source code, to mine source code for unbound uses of design fragments (see section 8.2.8), and to author design fragments would reduce user effort, and might consequently speed adoption. Once a design fragment mining tool exists, it becomes reasonable to flag anti-patterns, and refer the programmer to a replacement design fragment.

A clear opportunity exists to integrate existing code analysis tools with the design fragments tools, for example, the `Fluid` [26] and `ESC/Java` [9] tools. Some tools require their own annotations in the Java code. It should be possible to extend the design fragments language to refer to these annotations directly, enabling a design fragment to insist that particular annotations exist for the design fragment to be satisfied. For example, a design fragment representing a thread-safe producer-consumer pattern could insist that `Fluid` annotations exist, so that the `Fluid` tool could check the safety of the code. Additionally, a small improvement to the existing design fragments required-call analysis would yield a large benefit. It currently insists that the call exists within

the bound method, so refactoring the call into a subroutine will cause a constraint violation. It could be revised to look within subroutines to make it more resilient.

While it was not our intention to discover bugs in existing code, our case studies did reveal systematic errors in client programs. If, in the future, design fragments are used extensively, there is an opportunity to focus programmers' attention on code that deviates from the standard practice as defined by the catalog(s) of design fragments. Non-conforming code may well be correct, but our experience shows that it is a worthwhile place to apply attention in code reviews and testing.

While these improvements can be implemented rather quickly, and would significantly increase the adoptability of design fragments, their contribution to the field of software engineering would be modest. However, our research on design fragments also suggests three areas of longer term research with potential to significantly impact software engineering.

First, as discussed in section 9.3.4, it should be possible to refactor frameworks such that client-framework interactions are more assurable. It may be possible to provide a standard set of refactorings for framework interfaces, or a set of guidelines, that would help framework authors transform existing interfaces or build new ones. Changes such as these could significantly increase code quality.

Second, the design fragment language expresses how clients can interact with a framework, specifying the object-oriented interactions that are necessary. It is currently not possible to discuss client-framework interaction, or framework constraints, without referencing object-oriented mechanics, yet different frameworks implement the same concepts with different mechanics. For example, events can be communicated to a client program with a method call on a distinguished method, a method call with a distinguished parameter, or via a distributed event bus. It is also the case that different concepts have identical mechanics. For example, events from a framework can be synchronous or asynchronous, yet the method call on the client program would look the same. What is needed is a language that expresses the concepts of framework constraints and client-framework interaction without referencing object-oriented mechanics.

Once these framework concepts are in place, it should be possible to design a next generation object-oriented language where framework concepts have first class representation. A similar path was taken with software architecture, where the concepts were first developed [57], then merged into Java with the ArchJava [1] language. A similar integration would be fruitful between architecture description languages [24] and framework concepts.

Third, from an implementation perspective, we take for granted that a client program written for one framework will not work on another framework. We once had the same perspective about client programs for operating systems, but it is now common to write one program that is targeted at multiple operating systems. While the points of variability and the strategies for re-targeting are now well understood for operating systems, we do not yet have a similar understanding for frameworks. Solutions range from heavyweight, like fully emulating the hardware, to lightweight, like using different math libraries. The Eclipse Rich Client Platform [42] allows client programs to work across the operating systems supported by Eclipse, but does not address how to re-target an application for a different framework. Sometimes incompatibilities will be insurmountable, but considering the similarity of desktop environments, and the similarities of application servers, it seems possible that we can remove the accidental incompatibilities and re-target many client programs.

9.6 Summary

Frameworks are a valuable way to share designs and implementations on a large scale. Client programmers, however, have difficulty using frameworks. They find it difficult to understand non-local client-framework interactions, design solutions when they do not own the architectural skeleton, gain confidence that they have engaged with the framework correctly, represent their successful engagement with the framework in a way that can be shared with others, ensure their design intent is expressed in their source code, and connect with external files.

A design fragment is a specification of how a client program can use framework resources to accomplish a goal. From the framework, it identifies the minimal set of classes, interfaces, and methods that should be employed. For the client program, it specifies the client-framework interactions that must be implemented. The structure of the client program is specified as roles, where the roles can be filled by an actual client program's classes, fields, and methods. A design fragment exists separately from client programs, and can be bound to the client program via annotations in their source code. These annotations express design intent; specifically that it is the intention of the client programs to interact with the framework as specified by the design fragment.

The thesis of this dissertation is: *We can provide pragmatic help for programmers to use frameworks by providing a form of specification, called a design fragment, to describe how a client program can correctly employ a framework and by providing tools to assure conformance between the client program and the design fragments.*

To demonstrate that the help provided by design fragments and assurance tools is pragmatic, we collected evidence to support three subordinate hypotheses. The first was that design fragments can be used with existing large commercial frameworks, real programming languages, and off-the-shelf code. The second was that the variety of design fragments to accomplish a given goal is limited, so a small catalog of design fragments can have good coverage of the code seen in practice. And the third was that analysis can provide programmers with assurance that their code conforms to the constraints of the framework as expressed in design fragments. We built tools into an IDE to demonstrate how design fragments could alleviate the difficulties experienced by client programmers, and used these tools to collect empirical evidence on how client programs interact with frameworks.

We performed two case studies on commercial Java frameworks, using demo client programs from the framework authors, and client programs we found on the internet. The first case study, on the Applet framework, yielded a complete catalog of twelve design fragments based on our analysis of fifty-six Applets. There was evidence that Applets we collected from the internet had been influenced by demo Applets provided by the framework authors, and of incorrectly inferred framework constraints. The second case study, on the larger Eclipse framework, yielded a partial catalog of fourteen design fragments based on our analysis of more than fifty client programs. There was evidence on this framework also that client programs from the internet were influenced by the demo client programs. There was also evidence that client programs were interacting with the framework following consistent patterns, which we were able to encode as design fragments.

This work provides three primary contributions to software engineering. First, it provides a new technique to help programmers use frameworks. Design fragments improve on previous abstractions by describing relevant resources in the framework, connecting object-oriented code

with external declarative configuration files, and by enabling new constraints and tools to be added to our predefined set.

Second, it provides a systematic way to increase code quality. Design fragments provide a means to communicate known-good designs to programmers, and, unlike simple copying of examples, a means of influencing the uses of that design so that revisions can be propagated. Bindings between design fragments and client programs express the programmer's design intent, and this enduring expression of intent enables conformance assurance from both existing and future analysis tools.

Third, it provides an empirically-based understanding of how clients use frameworks, which aids researchers in choosing research directions and aids framework authors in delivery of new frameworks. Specifically, the observed variety of client-framework interactions is very low, despite the opportunity for high variety.

Appendix A

Adding a New Specification

The design fragments language was designed to be extensible by users. The easiest extension is simply to write the specification in natural language within a *freeformspec*, but such an extension cannot be checked by tools. Users can add structured, checkable specifications to those already in the design fragments language. Indeed, all of the specifications already in the language are implemented using the same technique we will describe here.

The following sections walk through adding a new specification and an analysis tool to check that specification. Our example comes from the Applet framework. When an Applet is embedded in a web page, the author of that web page can pass parameters to the Applet. These parameters can be anything, but are often configuration values like the height and width of the applet, or preferred colors. It is possible to invoke the `getParameterInfo()` method on the Applet to ask it which parameters it needs. This callback method on the Applet must be implemented by each Applet, and should return an array containing every parameter that it needs, and a description of what that parameter is used for.

The Applet itself can ask the framework what parameters have been passed in from the web page. This is done by calling the framework service method `getParameter(parameterName)` for each parameter. It makes sense that the array of parameters returned by `getParameterInfo()` should match the parameters that the Applet checks by calls to `getParameter(parameterName)`. It is easy to imagine a programming error where an additional parameter is checked yet not reported. We will start off by writing our specification in natural language, convert it over to a structured specification, then show how an analysis tool could check this specification.

A.1 New Specification

We will create a new specification in the design fragment language called *parametersCheckedAndReportedSpec*. The current Parameterized Applet design fragment is shown in Figure A.1.

As you can see in bold, our design fragment currently contains *freeformspec* that tells the programmer about the constraint, but since it is written in natural language, it cannot be parsed by a tool and enforced. We will change that *freeformspec* to *parametersCheckedAndReportedSpec* specification as shown in Figure A.2. This specification has been placed on the `getParam-`

```

<class>
  <name>RoleApplet</name>
  <superclass>java.applet.Applet</superclass>
  <method>
    <name>init</name>
    <returnvalue>void</returnvalue>
    <requiredcallspec
      targetobject="this"
      targetmethod="getParameter"
      arguments="the name of the parameter"/>
    <freeformspec text="See if web page that contains this applet
      has provided this parameter" />
  </method>
  <method>
    <name>getParameterInfo</name>
    <returnvalue>String [][]</returnvalue>
    <b><freeformspec text="Should check that every parameter checked
      in this program via getParameter(), usually in init(),
      is in the returned array" /></b>
  </method>
</class>

```

Figure A.1: Current Parameterized Applet design fragment

eterInfo method but could also be placed on the `init` method or even on the `RoleApplet` class itself. In order for the design fragment parser to understand this new specification, we must declare it in the design fragment XML Schema Definition (XSD) [27].

The design fragment XSD encodes the syntax of the XML representation of the design fragment language. Our new specification, shown in Figure A.3, is expressed in XSD. The first bolded section declares that our new specification can be added to a method specification, and the second bolded section defines our new specification.

At this point we can now write design fragments containing the *parametersCheckedAndReportedSpec* and the parser will understand them. In some cases it would be appropriate to stop here with a just a specification, for example if tooling was currently unable to check such

```

<class>
  ...
  <method>
    <name>getParameterInfo</name>
    <returnvalue>String [][]</returnvalue>
    <b><parametersCheckedAndReportedSpec /></b>
  </method>
</class>

```

Figure A.2: Improved Parameterized Applet design fragment

```

<xs:complexType name="method-type">
  <xs:sequence maxOccurs="unbounded" minOccurs="0">
    <xs:choice>
      <xs:element name="name" type="xs:string" ... />
      <xs:element name="abstract" type="empty-type" ... />
      <xs:element name="returnvalue" type="xs:string" ... />
      <xs:element name="code" type="xs:string" ... />
      <xs:element name="argument" type="method-argument-type" ... />
      <xs:element name="requirednewinstance"
        type="required-new-instance-type" ... />
      <xs:element name="requiredcallspec"
        type="required-call-spec-type" ... />
      <xs:element name="freeformspec" type="freeform-spec-type" ... />
      <xs:element name="parametersCheckedAndReportedSpec"
        type="parameterscheckedandreported-spec-type"
        minOccurs="0" maxOccurs="unbounded" />
        <xs:element name="comment" type="comment-type" ... />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
  ...
  <xs:complexType name="parameterscheckedandreported-spec-type">
    <xs:attribute name="comment" type="xs:string" use="optional" />
  </xs:complexType>

```

Figure A.3: Design fragment XSD for parameter checking

```

<extension point="DesignFragments.specification">
  <specificationtype
    parent="method"
    name="parametersCheckedAndReported"
    description="Checked parameters = reported parameters" >
    <specattribute
      key="comment"
      valuetype="string"
      required="false" />
  </specificationtype>
  <presenter class="my.package.ParameterSpecPlugin" />
  <checker class="my.package.ParameterSpecPlugin" />
</extension>

```

Figure A.4: Declaration of presenter and checker in `plugin.xml`

a specification. We will continue on and show how the checking could be implemented for our new specification.

A.2 Tool Extension

When the design fragment tool see a specification in a design fragment, it consults the list of specification checkers to find one that is registered to handle this kind of specification. If it finds a matching checker, it hands the specification and some context information to the checker. So when writing a new specification checker, we need to ensure that our new checker appears on the list of checkers. This is done by declaring the checker as an Eclipse plugin, which is done in the `plugin.xml` file for the design fragment tool. Figure A.4 shows the declaration of a checker for our new specification.

The third line, `parent="method"`, declares that this specification applies to a method instead of a class or field. The second and third lines from the end declare which Java classes provide the implementation for this checker. The implementation is required to provide both a checker, which decides if the specification conforms to the source code, and a presenter, which provides the text and images necessary for this specification to be put into the user interface.

The final step is to write the Java code that implements the checker and presenter classes. The checker and presenter must implement the `IDFMethodSpecAnalyzer` and `IDFSpecPresenter` interfaces, shown in Figure A.5. When the design fragment tool invokes the checker, it passes in a reference to the source code Abstract Syntax Tree node (it is inside the `DFUserDeclaration`), which enables the checker to perform analysis.

A.3 Summary

All of the provided specifications have been specified in the manner we have described here. Users can also add their own class, field, method, or XML specification by following the same

```

public interface IDFMethodSpecAnalyzer {
    public boolean isEnabled();
    public void setEnabled(boolean value);
    public String getName();
    public void setName(String aName);
    public String getDescription();
    public void setDescription(String aDescription);
    public boolean checkMethodSpec(
        DFMethodInstance methodInst,
        DFSpecInstance methodSpecInst,
        DFUserDeclaration declaration);
}
...
public interface IDFSpecPresenter {
    public String getLabel( DFSpec aSpec );
    public Image getImage();
    public int category();
}

```

Figure A.5: Interfaces for Analyzer and Presenter

process. When documenting a constraint, the easiest option is to document it in natural language using the existing *freeformspec* specification, but since they are in natural language, they cannot be analyzed by tools. The next step is to create a new structured specification as we showed above. These specifications can have their own parameters, and they capture design intent more precisely. These specifications are preferred over natural language specifications even when no analysis for them yet exists, because their structured nature holds the promise that future tools could analyze the specification, and because they help human readers and authors. Finally, analysis tools can be defined using a combination of XML and Java code that work with interfaces provided by the design fragment tool.

Appendix B

Design Fragment Language

B.1 Abstract Syntax

This is the abstract syntax of the design fragment language, described using Extended Backus-Naur Form (EBNF) [20]. The first part defines the object-oriented structure. It cannot be changed by users. Parts in bold are where the specifications fit in. Note that unstructured text is italicized, as in the *name-text* in the first line.

```
design-fragment = name-text , goal-text , [ is-deprecated ] ,  
                  framework-provided , programmer-required ;  
is-deprecated = "yes" | "no" ;  
framework-provided = { type } ;  
programmer-required = { type } , [ { xml-file } ] ;  
type = class | interface ;  
class = name , [ superclass ] , [ implements-interface ] ,  
       [ { field } ] , [ { method } ] , [ { class-spec } ] ;  
interface = name , [ superclass ] , [ { field } ] , [ { method } ] ;  
superclass = name ;  
implements-interface = name ;  
field = name , field-type-text , [ { field-spec } ] ;  
method = name , return-type-text , [ { argument } ] ,  
        [ is-abstract ] , [ source-code ] , [ { method-spec } ] ;  
argument = name , argument-type-text ;  
is-abstract = "yes" | "no" ;
```

The second part defines the structure for configuration files.

```
xml-file = name , xml-node ;  
xml-node = name , [ { xml-node } ] , [ xml-role-name-text ] ,  
           [ { xml-spec } ] ;
```

The third part defines the specifications. Additional specifications can be added by users.

```
class-spec = class-referenced-in-xml-spec ;  
class-referenced-in-xml-spec = xml-role-name ,  
                           attribute-name-text ;
```

```

field-spec = ; (* no field specs pre-defined *)
method-spec = freeform-spec | required-call-spec |
    required-new-instance-spec ;
freeform-spec = specification-text ;
required-call-spec = target-class-name ,
    target-method-name , arguments , purpose ;
required-new-instance-spec = target-class-name ,
    arguments , purpose ;
xml-spec = string-match-spec | attribute-exists-spec ;
string-match-spec = specification-purpose ,
    xpath-predicate-1 , xpath-predicate-2 ;
attribute-exists-spec = attribute-name ;

```

B.2 XML Schema Definition

This is the XML Schema Definition [27] for the concrete syntax of the design fragment language.

```

<?xml version="1.0" encoding="UTF-8"?>
<xss:schema
  xmlns:xss="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://designfragments.org/df1.0"
  xmlns="http://designfragments.org/df1.0">
  <!-- Definition of complex elements -->
  <xss:element name="designfragment">
    <xss:complexType>
      <xss:sequence maxOccurs="unbounded" minOccurs="0">
        <xss:choice>
          <xss:element name="name" type="xs:string" />
          <xss:element name="deprecated" type="empty-type"
            minOccurs="0" maxOccurs="1"/>
          <xss:element name="goal" type="xs:string" />
          <xss:element name="framework-provided"
            type="framework-provided-type" />
          <xss:element name="programmer-created"
            type="programmer-created-type" />
          <xss:element name="comment" type="comment-type"
            minOccurs="0" maxOccurs="unbounded" />
        </xss:choice>
      </xss:sequence>
    </xss:complexType>
  </xss:element>
  <!-- Definition of complex types -->
  <xss:complexType name="empty-type">
  </xss:complexType>
  <xss:complexType name="framework-provided-type">
    <xss:sequence maxOccurs="unbounded" minOccurs="0">
      <xss:choice>
        <xss:element name="interface" type="class-type"
          minOccurs="0" maxOccurs="unbounded"/>

```

```

<xs:element name="class" type="class-type"
    minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="comment" type="comment-type"
    minOccurs="0" maxOccurs="unbounded" />
</xs:choice>
</xs:sequence>
</xs:complexType>

<xs:complexType name="programmer-created-type">
<xs:sequence maxOccurs="unbounded" minOccurs="0">
    <xs:choice>
        <xs:element name="interface" type="class-type"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="class" type="class-type"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="xmlfile" type="xmlfile-type"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="comment" type="comment-type"
            minOccurs="0" maxOccurs="unbounded" />
    </xs:choice>
</xs:sequence>
</xs:complexType>
<xs:complexType name="class-type">
<xs:sequence maxOccurs="unbounded" minOccurs="0">
    <xs:choice>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="abstract" type="empty-type"
            minOccurs="0" maxOccurs="1"/>
        <xs:element name="superclass" type="xs:string"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="implementsinterface" type="xs:string"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="field" type="field-type"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="method" type="method-type"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="classreferencedineclipseextensionpoint"
            type="class-referenced-in-eclipse-extension-point-type"
            minOccurs="0" maxOccurs="unbounded" />
        <xs:element name="comment" type="comment-type"
            minOccurs="0" maxOccurs="unbounded" />
    </xs:choice>
</xs:sequence>
</xs:complexType>
<xs:complexType name="interface-type">
<xs:sequence maxOccurs="unbounded" minOccurs="0">
    <xs:choice>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="superclass" type="xs:string"
            minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="implementsinterface" type="xs:string"
            minOccurs="0" maxOccurs="unbounded"/>

```

```

<xs:element name="field" type="field-type"
    minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="method" type="method-type"
    minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="comment" type="comment-type"
    minOccurs="0" maxOccurs="unbounded" />
</xs:choice>
</xs:sequence>
</xs:complexType>
<xs:complexType name="field-type">
    <xs:sequence maxOccurs="unbounded" minOccurs="0">
        <xs:choice>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="type" type="xs:string"/>
            <xs:element name="freeformspec" type="xs:string"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="comment" type="comment-type"
                minOccurs="0" maxOccurs="unbounded" />
        </xs:choice>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="method-type">
    <xs:sequence maxOccurs="unbounded" minOccurs="0">
        <xs:choice>
            <xs:element name="name" type="xs:string"
                minOccurs="1" maxOccurs="1"/>
            <xs:element name="abstract" type="empty-type"
                minOccurs="0" maxOccurs="1"/>
            <xs:element name="returnvalue" type="xs:string"
                minOccurs="1" maxOccurs="1"/>
            <xs:element name="code" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
            <xs:element name="argument" type="method-argument-type"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="requirednewinstance"
                type="required-new-instance-type"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="requiredcallspec"
                type="required-call-spec-type"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="freeformspec" type="freeform-spec-type"
                minOccurs="0" maxOccurs="unbounded" />
            <xs:element name="comment" type="comment-type"
                minOccurs="0" maxOccurs="unbounded" />
        </xs:choice>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="method-argument-type">
    <xs:sequence maxOccurs="unbounded" minOccurs="0">
        <xs:choice>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="type" type="xs:string"/>

```

```

<xs:element name="comment" type="comment-type"
    minOccurs="0" maxOccurs="unbounded" />
</xs:choice>
</xs:sequence>
</xs:complexType>
<xs:complexType name="xmlfile-type">
    <xs:sequence maxOccurs="unbounded" minOccurs="0">
        <xs:choice>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="xmlfilespecs" type="xmlfilespecs-type"
                minOccurs="0" maxOccurs="1"/>
            <xs:element name="contents" type="contents-type"
                minOccurs="0" maxOccurs="1"/>
            <xs:element name="comment" type="comment-type"
                minOccurs="0" maxOccurs="unbounded" />
        </xs:choice>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="xmlfilespecs-type">
    <xs:sequence>
        <xs:element name="stringmatchspec"
            type="string-match-spec-type"
            minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="contents-type">
    <xs:sequence>
        <xs:any minOccurs="0" maxOccurs="unbounded"
            processContents="skip" />
    </xs:sequence>
</xs:complexType>
<xs:simpleType name="comment-type">
    <xs:restriction base="xs:string" />
</xs:simpleType>
<!-- The following specs should be defined in another XSD --&gt;
<!-- For convenience, they are defined here --&gt;
&lt;xs:complexType name="freeform-spec-type"&gt;
    &lt;xs:attribute name="text" type="xs:string"/&gt;
    &lt;xs:attribute name="comment" type="xs:string" /&gt;
&lt;/xs:complexType&gt;
&lt;xs:complexType name="required-call-spec-type"&gt;
    &lt;xs:attribute name="targetobject" type="xs:string"/&gt;
    &lt;xs:attribute name="targetmethod" type="xs:string"/&gt;
    &lt;xs:attribute name="arguments" type="xs:string" /&gt;
    &lt;xs:attribute name="purpose" type="xs:string" use="optional" /&gt;
    &lt;xs:attribute name="text" type="xs:string" use="optional"/&gt;
&lt;/xs:complexType&gt;
&lt;xs:complexType name="required-new-instance-type"&gt;
    &lt;xs:attribute name="target" type="xs:string"/&gt;
    &lt;xs:attribute name="arguments" type="xs:string" /&gt;
    &lt;xs:attribute name="purpose" type="xs:string" use="optional" /&gt;
</pre>

```

```
<xs:attribute name="text" type="xs:string" use="optional"/>
</xs:complexType>
<xs:complexType
  name="class-referenced-in-eclipse-extension-point-type">
  <xs:attribute name="xmlrolename" type="xs:string" />
  <xs:attribute name="attributename" type="xs:string"/>
</xs:complexType>
<xs:complexType name="string-match-spec-type">
  <xs:attribute name="reason" type="xs:string" />
  <xs:attribute name="p1" type="xs:string"/>
  <xs:attribute name="p2" type="xs:string"/>
</xs:complexType>
</xs:schema>
```

Bibliography

- [1] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-472-X. doi: <http://doi.acm.org/10.1145/581339.581365>. 9.5
- [2] Chris Anderson. *The Long Tail: Why the Future of Business Is Selling Less of More*. Hyperion, 2006. ISBN 1401302378. 9.2.7
- [3] M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. *MoDELS 2006 - Model Driven Engineering Languages and Systems, 9th International Conference*, 0:692–706, 2006. 8.2.5
- [4] Jagdish Bansya. Automating design-pattern identification. *Dr. Dobb's Journal*, 0, 1998. 8.2.8
- [5] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall PTR, 1996. ISBN 013476904X. 1.2
- [6] Jon Bentley. Programming pearls: little languages. *Commun. ACM*, 29(8):711–721, 1986. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/6424.315691>. 9.2.2
- [7] Ian Blackburn. Are you mort, elvis, or einstein?, January 2004. URL <http://www.bbits.co.uk/blog/archive/2004/01/15/167.aspx>. 9.2.1
- [8] Steven Clarke. Measuring API usability. *Dr. Dobb's Journal*, 0:S6–S9, 2004. 9.2.1
- [9] D.R. Cok and J.R. Kiniry. ESC/Java2: Uniting ESC/Java and JML . Technical Report NIII-R0413, Radboud University Nijmegen, May 2004. 9.2.3, 9.5
- [10] Brad Cox. *Object-Oriented Programming, An Evolutionary Approach*. Addison Wesley, New York, 1987. 8.2.1
- [11] Danny Dig and Ralph Johnson. The role of refactorings in API evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2368-4. 1.3, 2.1, 7.1
- [12] Eclipse Foundation. Eclipse framework web page, 2007. URL <http://eclipse.org/eclipse>. 2.1, 5
- [13] Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall PTR, 2005. ISBN 0131858580. 2.2
- [14] George Fairbanks, David Garlan, and William Scherlis. Design fragments make us-

- ing 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 Press. ISBN 1-59593-348-4. doi: <http://doi.acm.org/10.1145/1167473.1167480>. 6.3
- [15] D. Fay. An architecture for distributed applications on the internet: Overview of microsoft's .NET platform. *IEEE International Parallel and Distributed Processing Symposium*, 00: 90a, April 2003. ISSN 1530-2075. doi: <http://doi.ieeecomputersociety.org/10.1109/IPDPS.2003.1213196>. 1, 2.1
 - [16] Rudolf Ferenc, Arpad Beszedes, Lajos Fulop, and Janos Lele. Design pattern mining enhanced by machine learning. *icsm*, 00:295–304, 2005. ISSN 1063-6773. doi: <http://doi.ieeecomputersociety.org/10.1109/ICSM.2005.40>. 8.2.8
 - [17] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *Lecture Notes in Computer Science*, pages 472–495, Jyvaskyla, Finland, June 1997. 8.2.2
 - [18] Francesca Arcelli Fontana, Claudia Raibulet, and Francesco Tisato. Design pattern recognition. In *IASSE*, pages 290–295. ISCA, 2004. 8.2.8
 - [19] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The UML Profile for Framework Architectures*. Addison-Wesley Professional, 2001. ISBN 0201675188. 8.2.4
 - [20] ISO/IEC Information Technology Task Force. ISO/IEC 14977:1996. Technical report, International Organization for Standardization, 1996. URL [http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996\(E\).zip](http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip). 4.2.1, B.1
 - [21] Gary Froehlich, H. James Hoover, Ling Liu, and Paul 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. ISBN 0-89791-914-9. doi: <http://doi.acm.org/10.1145/253228.253432>. 1.5, 8.2.3
 - [22] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley Professional, 2003. ISBN 0321205758. 1.3, 2.5.1
 - [23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-WesleyProfessional Computing Series)*. Addison-Wesley Professional, 1995. ISBN 0201633612. 1.1, 2.3.5, 8.2.4, 8.2.8, 9.2.3, 9.5
 - [24] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 3, pages 47–67. Cambridge University Press, 2000. ISBN 0-521-77164-1. 4.7, 9.5
 - [25] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, The (3rd Edition) (The Java Series)*. Prentice Hall PTR, 2005. ISBN 0321246780. 4.3
 - [26] Aaron Greenhouse, T.J. Halloran, and William L. Scherlis. Observations on the assured

evolution of concurrent Java programs. *Science of Computer Programming*, 58:384–411, March 2005. 4.7, 9.2.3, 9.2.6, 9.5

- [27] Shruti Gupti and Sonal Mukhi. *XML Schema Definition (XSD)*. BPB Publications, 2002. ISBN 8176566640. 4.2.1, A.1, B.2
- [28] Markku Hakala, Juha Hautamäki, Kai Koskimies, Jukka Paakki and Antti Viljamaa, and Jukka Viljamaa. Annotating reusable software architectures with specialization patterns. In *WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, page 171, Washington, DC, USA, 2001. ISBN 0-7695-1360-3. doi: <http://dx.doi.org/10.1109/WICSA.2001.948426>. 1.5, 8.2.4
- [29] Imed Hammouda and Kai Koskimies. A pattern-based J2EE application development environment. *Nordic Journal of Computing*, 9(3):248–260, 2002. 8.2.4, 9.2.4
- [30] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ISBN 1-58113-471-1. doi: <http://doi.acm.org/10.1145/582419.582436>. 8.2.4
- [31] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 169–180, New York, NY, USA, 1990. ACM Press. ISBN 0-201-52430-X. doi: <http://doi.acm.org/10.1145/97945.97967>. 1.5, 8.2.5
- [32] James Holmes. *Struts: The Complete Reference, 2nd Edition (Complete Reference Series)*. McGraw-Hill Osborne Media, 2006. ISBN 0072263865. 1.2
- [33] Daqing Hou and H. James Hoover. Towards specifying constraints for object-oriented frameworks. In *CASCON '01: Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative research*, page 5. IBM Press, 2001. 1.5, 8.2.5
- [34] Daqing Hou, Kenny Wong, and H. James Hoover. What can programmer questions tell us about frameworks? In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 87–96, Washington, DC, USA, 2005. ISBN 0-7695-2254-8. doi: <http://dx.doi.org/10.1109/WPC.2005.47>. 4.1
- [35] IBM Corporation and others. Eclipse project, 2003. URL <http://eclipse.org>. 7.1
- [36] Ralph E. Johnson. Documenting frameworks using patterns. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 63–76, New York, NY, USA, 1992. ISBN 0-201-53372-3. doi: <http://doi.acm.org/10.1145/141936.141943>. 8, 8.2.4
- [37] Ralph E. Johnson. Components, frameworks, patterns. *SIGSOFT Softw. Eng. Notes*, 22(3): 10–17, 1997. ISSN 0163-5948. doi: <http://doi.acm.org/10.1145/258368.258378>. 8.2.4
- [38] Ralph E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, 1997. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/262793.262799>. 2.1, 2.3.1, 8.2.1
- [39] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented*

Programming, 1:22–35, June/July 1988. 2.2

- [40] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. ISBN 3-540-42206-4. 5.1, 8.2.6
- [41] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988. ISSN 0896-8438. 1.5, 8.2.3
- [42] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley Professional, 2005. ISBN 0321334612. 9.5
- [43] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Expert Syst. Appl.*, 23(4):405–413, 2002. 8.2.8
- [44] Microsoft. .NET framework developer's guide - control execution lifecycle, 2007. URL <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconControlExecutionLifecycle.asp>. 1.3
- [45] Microsoft. Implementing sun microsystems java pet store J2EE blueprint application using microsoft .NET, November 2001. 9.3.2
- [46] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, Redmond, WA, USA, 2001. ISBN 0735614482. 9.2.8
- [47] Richard Monson-Haefel. *Enterprise JavaBeans (3rd Edition)*. O'Reilly, 2001. ISBN 0596002262. 1, 2.1
- [48] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proc. 7th IBM Conf. Object-Oriented Technology*, July 1994. 8.2.6
- [49] Dewayne E. Perry. The inscape environment. In *ICSE '89: Proceedings of the 11th International Conference on Software Engineering*, pages 2–11, New York, NY, USA, 1989. ISBN 0-8186-1941-4. doi: <http://doi.acm.org/10.1145/74587.74588>. 8.2.7
- [50] G. Polya. *How to Solve It: A New Aspect of Mathematical Method (Princeton Science Library)*. Princeton University Press, 2004. ISBN 069111966X. 1.3
- [51] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley Longman, 1994. ISBN 0201422948. 8.2.4
- [52] Dave Reed. Truly understanding ViewState. Blog posting, August 2006. URL <http://weblogs.asp.net/infinitiesloop/archive/2006/08/03/Truly-Understanding-Viewstate.aspx>. 1.3
- [53] Trygve Reenskaug, P. Wold, O. A. Lehne, and Manning. *Working With Objects: The Ooram Software Engineering Method*. Manning Pubns Co, 1995. ISBN 1884777104. 8.2.1
- [54] Charles Rich and Richard. C. Waters. The programmer's apprentice: A research overview.

- In D. Partridge, editor, *Artificial Intelligence & Software Engineering*, pages 155–182. ACM Press, Norwood, NJ, 1991. 8.2.7
- [55] Dirk Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2000. 1.5, 2.1, 8.2.1
 - [56] Roger Schank and Robert Abelson. *Scripts, Plans, Goals, and Understanding: An Inquiry Into Human Knowledge Structures (Artificial Intelligence)*. Lawrence Erlbaum, 1977. ISBN 0898591384. 1.3
 - [57] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996. ISBN 0-13-182957-2. 9.5
 - [58] Jason McC. Smith and David Stotts. SPQR: Flexible automated design pattern extraction from source code. *ase*, 00:215, 2003. ISSN 1527-1366. doi: <http://doi.ieeecomputersociety.org/10.1109/ASE.2003.1240309>. 8.2.8, 9.2.4
 - [59] Software in the Public Interest. The Debian linux distribution, 2007. URL <http://www.debian.org>. 5.4
 - [60] Detlef Streitferdt, Christian Heller, and Ilka Philippow. Searching design patterns in source code. In *COMPSAC (2)*, pages 33–34. IEEE Computer Society, 2005. ISBN 0-7695-2413-3. 8.2.8
 - [61] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, third edition, June 1997. ISBN 0201889544. 9.2.8
 - [62] Sun Microsystems. Java pet store, 2007. URL <https://blueprints.dev.java.net/petstore/>. 9.3.2
 - [63] Sun Microsystems. Java applets web page, 2007. URL <http://java.sun.com/applets/>. 1, 6
 - [64] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999. 4.3, 8.2.6
 - [65] Tom Tourwé. *Automated Support for Framework-Based Software Evolution*. PhD thesis, Vrije Universiteit Brussel, 2002. 8.2.4
 - [66] Tom Tourwé and Tom Mens. Automated support for framework-based software evolution. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 148, Washington, DC, USA, 2003. ISBN 0-7695-1905-9. 8.2.4, 8.3
 - [67] Ian Watson and Farhi Marir. Case-based reasoning: A review. *The Knowledge Engineering Review*, 9(4):355–381, 1994. 1.3
 - [68] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6. 5.1