JASPER: Facilitating Software Maintenance Activities With Explicit Task Representations

Michael J. Coblenz

CMU-CS-06-150
CMU-HCII-06-107

August 2006

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee
Brad A. Myers, Chair
James D. Herbsleb

*Submitted in partial fulfillment of the requirements
for the Degree of Master of Science*

Acknowledgments

Abstract

Recent research has shown that developers spend an average of 35% of their time performing navigations around code. Much of this time is spent on redundant navigations to code that the developer previously found. This is necessary today because existing development environments do not enable users to easily keep relevant information, such as web pages, textual notes, and code, visible during their tasks. Instead, users must constantly switch among and re-navigate to the various relevant artifacts. JASPER is a new system that allows users to collect relevant artifacts into a working set for easy reference. These artifacts are visible in a single view that represents the user's current task and allows users to easily make each artifact visible within its context. Users collect relevant artifacts pertaining to each task, and a distinct working set is maintained for each task. These working sets can be saved to disk and later loaded, so that users may return to tasks later and share task information with colleagues.

We predict that JASPER will significantly reduce time spent on redundant navigations. In addition, JASPER will facilitate multitasking, interruption management, and sharing task information with other developers. JASPER could be integrated with other tools, such as existing source code suggestion tools, to permit users to more easily collect and maintain sets of task-relevant information.

Table of Contents

1. Introduction

1.1. Motivation

Recent research by Ko et al. [9] has shown that software developers spend approximately 35% of their time performing the mechanics of navigations among code fragments in their IDE. Thus, navigations constitute a significant fraction of developer time, and reducing navigation time would be likely to significantly improve developer productivity.

Efforts to reduce this navigation time must consider how developers complete software maintenance tasks. Ko et al. gave evidence [9] for a "search-relate-collect" model, in which developers search for relevant information; relate it to their tasks; and then collect the information. This information consists of *artifacts,* which are relevant pieces of data that form a small, coherent unit. For example, a method implementation, a few contiguous variable declarations, or a particular syntactic element such as a certain *for* loop, may all be artifacts. But existing development environments do not support collecting small segments of code. Instead, they require users to choose artifacts at the *file* or *syntax* granularity rather than the *code* granularity. Users must select first a file to view and then scroll through the file or select an artifact from a long list. Systems also only facilitate showing a small number of artifacts simultaneously. The result is that users typically must navigate away from relevant code, requiring them to subsequently re-locate artifacts that were just visible.

Therefore, the data indicate that IDEs should have three features to facilitate navigation: they should allow users to easily *collect* relevant artifacts; keep those artifacts *visible*; and enable users to easily see the artifacts in *context*. There is significant room for improvement with respect to existing systems, since systems that are currently in use do not support these features. A system that facilitated collecting many different artifacts, keeping them visible at all times, and

navigating to the original context of each artifact, would be likely to greatly reduce the amount of time users spend navigating among artifacts compared to existing systems.

Widely-used development environments do not support these features. In Figure 1, a user is modifying the Paint program in the study described in [9]. There are many relevant artifacts: the initializations of other variables, the external documentation window for `JSlider` (although only a button for it is visible at the bottom of the screen in Figure 1), the code being edited, and other information encoded in the state of the environment, such as open editor tabs. However, in this view, which is typical of existing environments, very few pieces of information are available. Note the large amount of irrelevant code displayed. It is generally too much trouble for users to make all relevant artifacts visible — in many cases, this is impossible. For example, in Microsoft's Visual Studio 2005 and Eclipse 3.2, only tabs and panes are available, not overlapping windows. Previous versions of Visual Studio included support for overlapping windows, but it was removed presumably because people found it too hard to use. Apple's XCode, which is a widely used IDE on Mac OS X, also does not allow users to easily collect and view all the relevant information. XCode is shown in Figure 2 with a similar view to that used by the user in Figure 1.
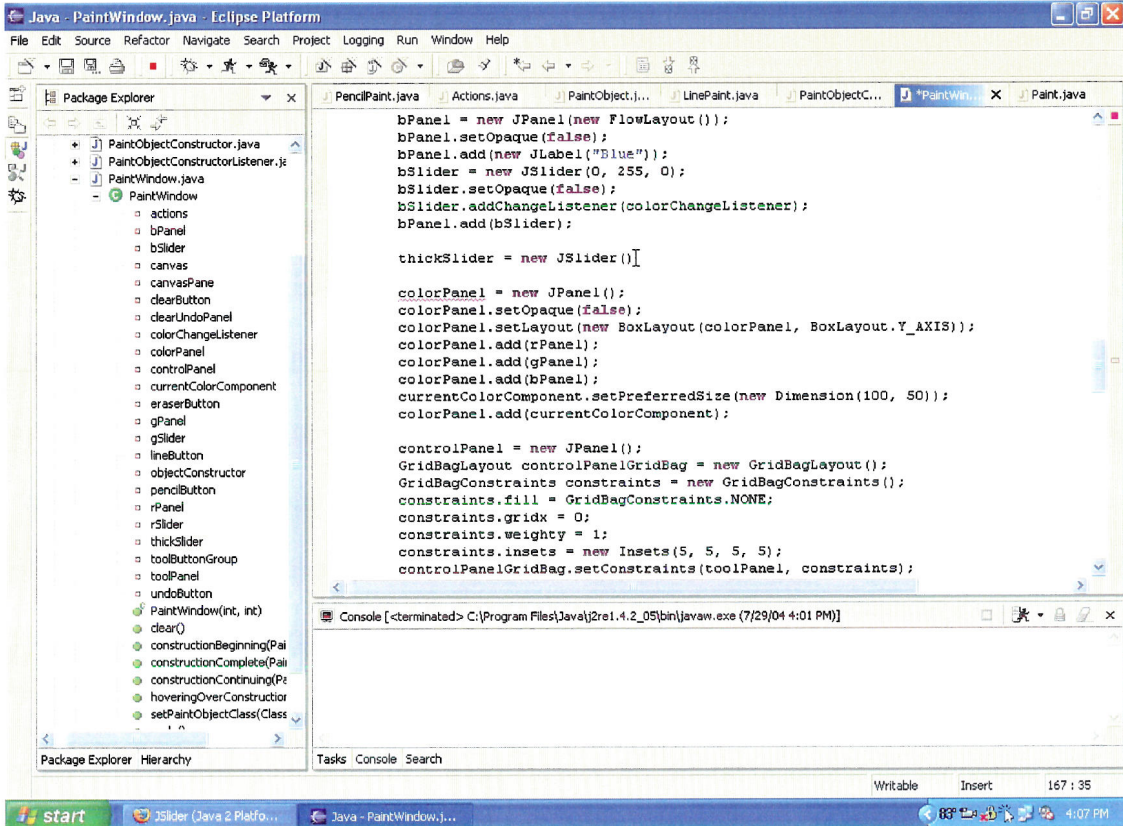
Figure 1: A user using Eclipse 2.1.2 to modify Paint [9], a small drawing program.
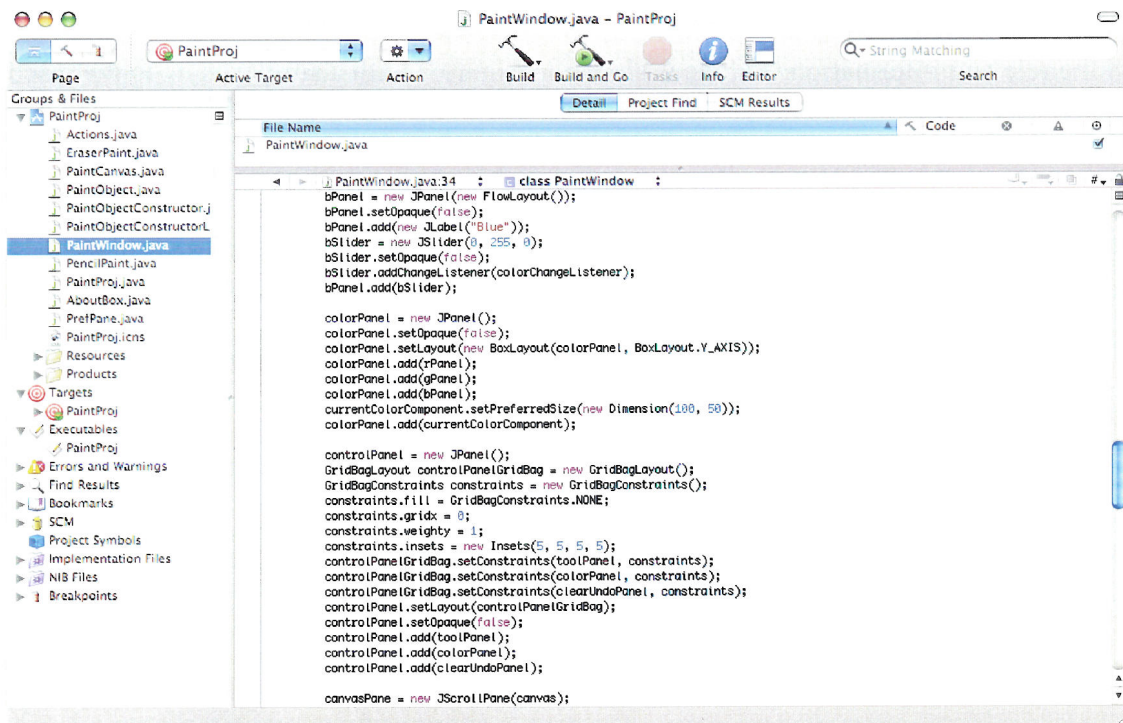

Figure 2: A similar view as in Figure 1 shown in XCode 2.3.

Because existing IDEs do not facilitate collection of relevant artifacts, users must perform navigations frequently. Navigations occur when information that the user wants to see is not visible. They also occur while users are searching for information but have not found it yet. One can think of the total navigation time spent by a user in a particular development period as follows:

$$\textit{Navigation time = number of navigations} \times \textit{mean time per navigation}$$

Therefore, one could reduce navigation time by reducing the number of navigations or by reducing the average time per navigation. There are two ways to reduce the number of navigations: reduce the amount of required information; and increase the amount of visible relevant information so that fewer navigations are required. This work focuses on new interaction techniques for reducing the number of navigations required and reducing the time required for many of the common navigations.

1.2. Approach

JASPER ("Java Aid with Sets of Pertinent Elements for Recognition") is a system that is designed to reduce the time spent by users doing navigation while performing programming tasks. By facilitating users' ability to organize relevant artifacts, it may significantly improve programmer productivity.

JASPER allows users to maintain a *working set* for each task. Each working set can contain working set *items*, each of which represents a particular artifact relevant to the task. These items are always visible in the working set so that users can always refer to them while completing the corresponding task. Furthermore, when users are interrupted, they can save working sets for later use. Then, when returning to the task, they can restore the previous context.

1.3. Benefits

In addition to saving programmers' time, this task and artifact structure may aid programmers in several other ways. Users of the system can share task information with other programmers. This has several benefits. First, if a user is interrupted and cannot proceed with a particular task, another user could continue, taking advantage of the information collected by the first user. Second, the working sets can be archived as part of a version-control system. Then, when considering changes related to a particular task, users may examine the previous working set used. This may give context to future tasks, helping to answer questions such as "why did the previous developer not notice this bug?" The answer might be of the form "since a particular line of code was not in the working set, the developer probably did not know about a dependency." In the opposite case, it can reveal dependencies: future users may not understand all the dependencies involved, but inspecting dependencies discovered by other users may help explain the changes that were previously made.

Sharing working sets with other users could also be helpful for informing users — especially new members of a development team — about the project. Working sets could be attached to bug reports in an organization's bug tracking system. Then, when developers later review previous changes, they can better understand the context in which the changes were made. A system such as TeamTracks [3] could be integrated to suggest working sets that are relevant to a particular task. This would be an improvement over TeamTracks alone because working sets represent explicit information about relationships among artifacts and task-relevance information for artifacts, rather than automatically-inferred information. Thus, working sets are likely to be more precise and accurate than automatically-inferred data. Users could discover both related code and related tasks that were previously unknown to them. The difficulty of this task, relating to cross-cutting concerns, has been well documented [8] by the aspect-oriented programming community.

Working sets can be thought of as each representing a particular aspect of the software under development. They may form a concise, convenient way of informally documenting aspects in the context of aspect-oriented programming. Working sets represent cross-cutting concerns, and as such, could be used by the aspect-oriented programming community as documentation.

Interruptions are a well-studied part of programming [5, 10]. Working sets represent an external memory of programming tasks. Some interruptions are predictable, such as lunch, the end of the work day, and the need to reboot one's computer. In these cases, programmers can assemble working sets to specifically represent their current progress to help them continue after the interruption. If users keep working sets up to date while working, then this benefit will result even for unpredictable interruptions. After an interruption, the previous working sets may be initially useful for several days as developers suspend tasks to work on other, higher priority tasks.

JASPER represents a new approach to managing information in a programming environment. Based on the data collected from studies of programmers, it may have a significant impact on programmer productivity. It may also facilitate information sharing in groups of programmers.

The next section describes the system and the rationale for its design. Section 3 discusses the implementation of JASPER. Section 4 continues with discussion of JASPER as a system. Section 5 details other work that also has the goal of helping users manage information while programming and improving programmer productivity. Section 6 gives several possible future directions for this research, and section 7 concludes with a summary of the contributions.

2. The JASPER System

2.1. Working Sets

JASPER is an Eclipse plug-in that maintains a list of *working sets* on which a user is working. Each working set consists of several working set *items*. A working set is intended to correspond to a particular *task*, or goal, that the user intends to accomplish. For example, a user might be working on a small drawing application, and have several tasks: fix the "undo" feature; add a tool for drawing lines; and permit users to change the thickness of drawn lines. The examples given here are identical to those used in [9] and the working sets illustrated are derived from the actual reported user data. In [9], Java programmers were given five modification tasks to complete in 70 minutes on a small drawing program called *Paint*. The tasks were: SCROLL (fix the scroll bars, which did not always appear); YELLOW (fix a bug preventing users from drawing in yellow); UNDO (fix the undo feature, which sometimes did not work); LINE (add a feature so that users could draw lines); and THICKNESS (allow users to control the thickness of their strokes).

In completing each task, the user must refer to various artifacts. For example, the THICKNESS task required adding a slider to control line thickness. It also required implementing an event listener for the slider and calling `setThickness` on the `PaintObjectConstructor`. Users completing this task might need to refer to documentation for `JSlider`, a report giving a description of the task, the segment of code in which the slider would be initialized, a copy of some analogous code that initializes another slider, and several other artifacts. These artifacts, however, are scattered in different places, including several source files and the Internet. JASPER lets users easily collect these artifacts and keep them visible.

In Figure 3, user G in [9] has created a new working set by clicking the New button, and named it according to the task: add a thickness slider to the Paint program. The user has double-clicked the working set in the Working Sets list, so JASPER has opened the working set view. At this point, the user continues to work as usual. In this case, the user has declared a new variable, `thickSlider`, and will soon need to initialize it and use it. Although this user did not have access to JASPER, the screen shots are constructed to represent one way that user G might have used the system. He puts the variable declaration in the working set so that he remembers the name of the variable.
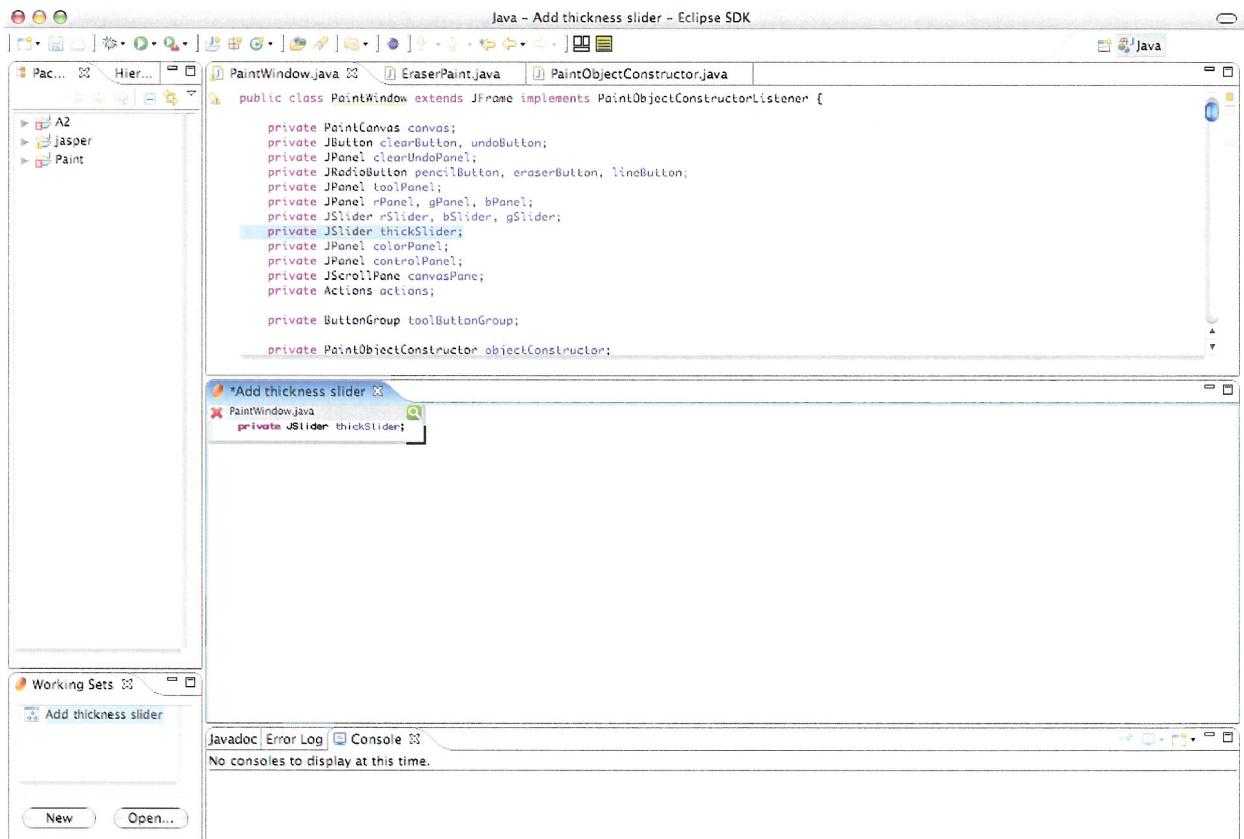

Figure 3: The user has declared the thickSlider variable.

In Figure 4, the `thickSlider` variable needs to be initialized by calling a constructor of the `JSlider` class. But the user does not remember the arguments for the constructor, so he

adds this partially-entered line of code to his working set. In Figure 4, this line is highlighted in blue because the cursor is currently on that line.


```
 *PaintWindow.java ⊠      EraserPaint.java      PaintObjectConstructor.java

        bPanel.setOpaque(false);
        bPanel.add(new JLabel("Blue"));
        bSlider = new JSlider(0, 255, 0);
        bSlider.setOpaque(false);
        bSlider.addChangeListener(colorChangeListener);
        bPanel.add(bSlider);

        thickSlider = new JSlider()

        colorPanel = new JPanel();
        colorPanel.setOpaque(false);
        colorPanel.setLayout(new BoxLayout(colorPanel, BoxLayout.Y_AXIS));
        colorPanel.add(rPanel);
        colorPanel.add(gPanel);
        colorPanel.add(bPanel);
        currentColorComponent.setPreferredSize(new Dimension(100, 50));
        colorPanel.add(currentColorComponent);

        controlPanel = new JPanel();
        GridBagLayout controlPanelGridBag = new GridBagLayout();
        GridBagConstraints constraints = new GridBagConstraints();
        constraints.fill = GridBagConstraints.NONE;
        constraints.gridx = 0;
        constraints.weighty = 1;


 *Add thickness slider ⊠
✖ PaintWindow.java                    🔍
      private JSlider thickSlider;
✖ PaintWindow.java#PaintWindow        🔍
        thickSlider = new JSlider()
```
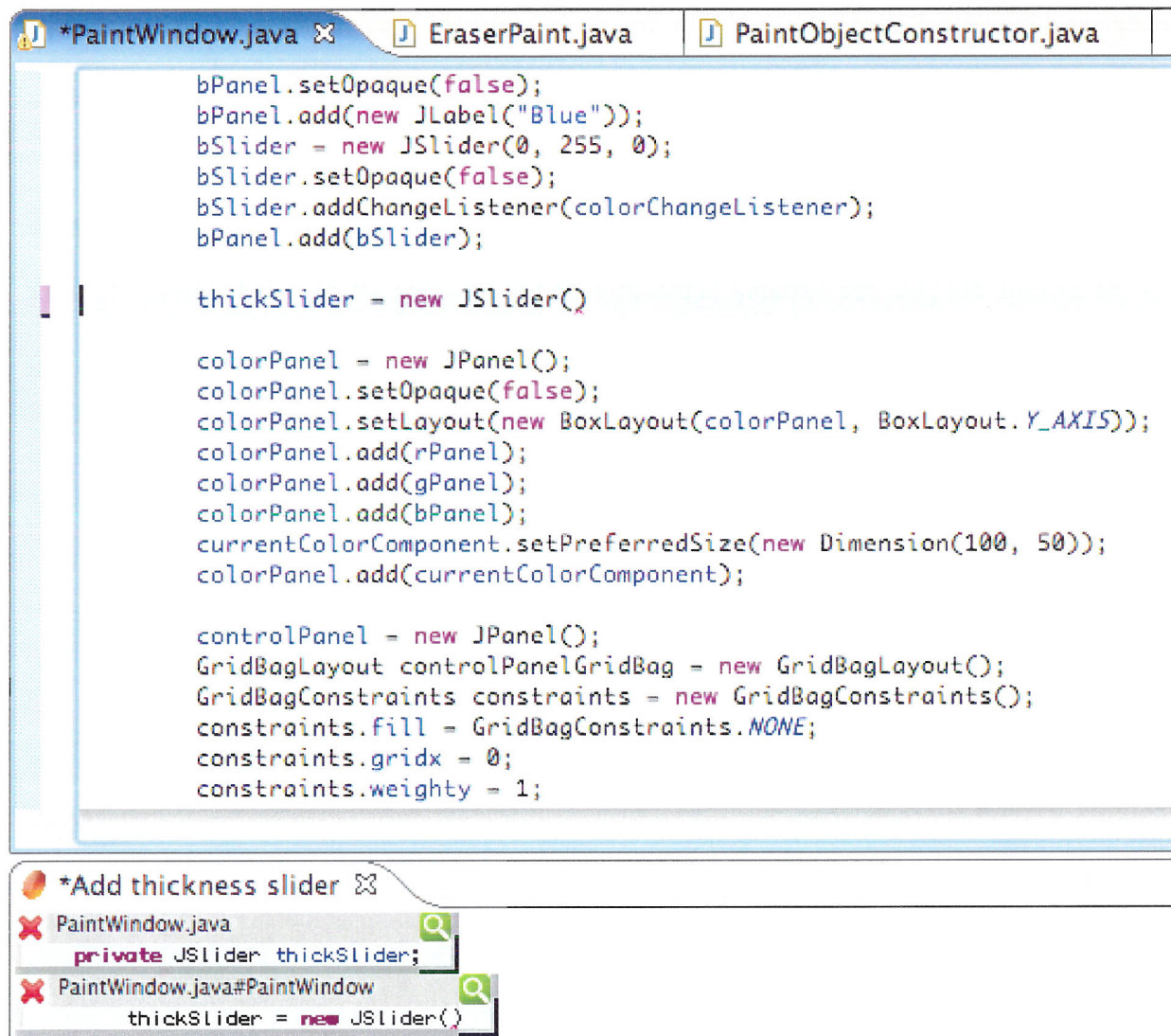
Figure 4: The user has added a call to a constructor of JSlider.

In Figure 5, the user used an external Web browser to find the `JSlider` documentation, and added the constructor documentation to the working set for easy reference.
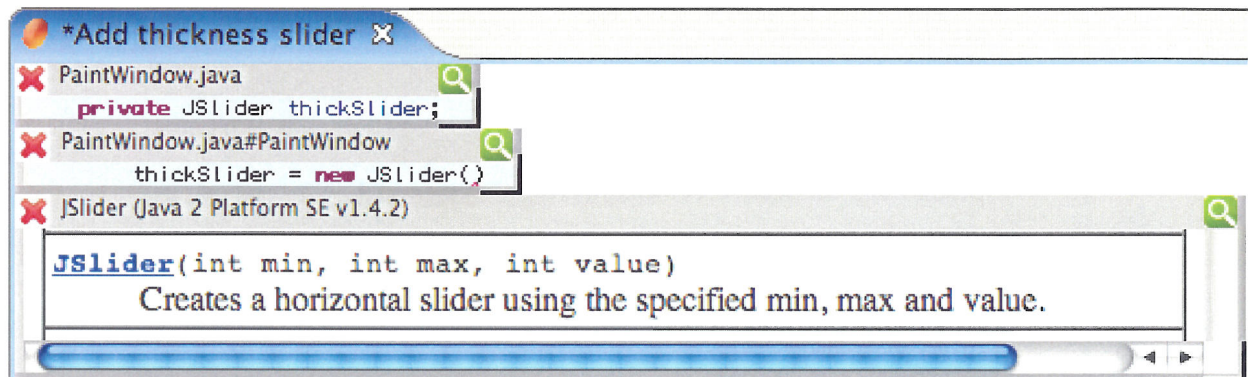

Figure 5: The user has added documentation for a `JSlider` constructor.

In the previous study, this user switched to Eclipse and tried to write the arguments for the `JSlider` constructor, but then realized he needed information that had been visible in the browser. He went back to the browser *twice* while writing this line of code. With JASPER, the relevant information is already visible.

After adding two lines of code in the top middle pane of Figure 6, the working set item, titled "PaintWindow.java#PaintWindow," is now too small. JASPER never resizes working set items once they have been created, as discussed in section 2.3. The user resized the changed working set item by dragging the resize widget in the lower right corner and clicked the "auto-layout" button so that the resized item does not overlap with the other items. Because the user added the new lines at the end of an existing working set item, JASPER included the two new lines of code in the item as he added them.
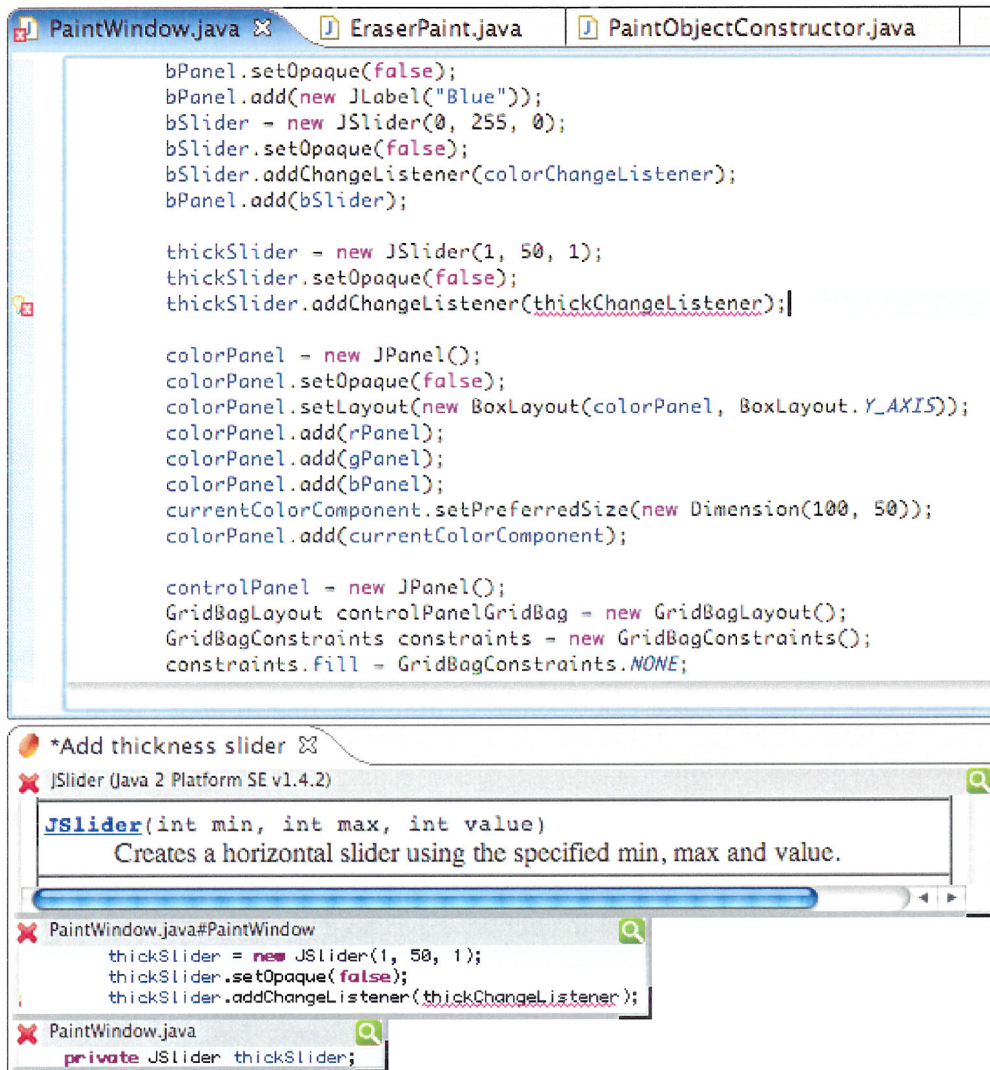
Figure 6: The user has added two lines of code below an existing working set item, so JASPER automatically added it to the item.

In the study, the user scrolled up to find the declarations section of the class. But with JASPER, the user could have double-clicked the item, shown in Figure 7, containing the `thickSlider` declaration. Notice that the user added the new variable declaration (for `thickPanel`) immediately below the `thickSlider` declaration. JASPER automatically includes new code added immediately above or below existing working set items, so no new working set item was necessary in this case. The user resized the item to show the new code.
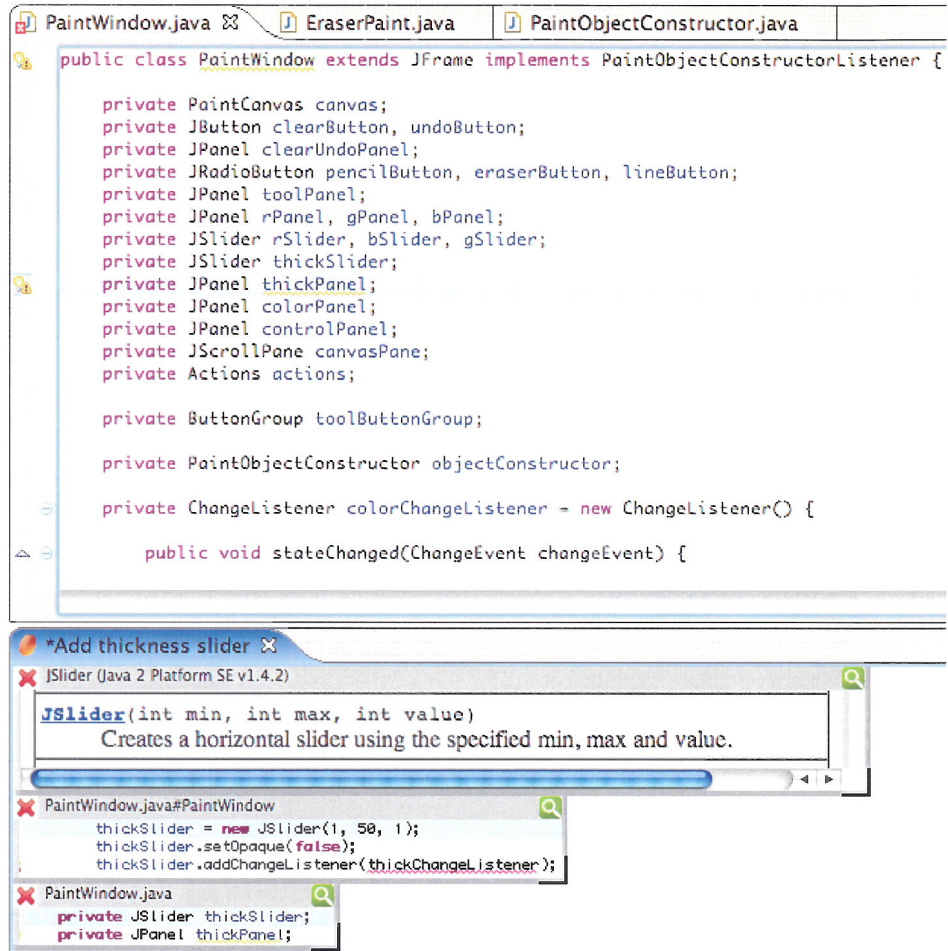
Figure 7: The user has added a new variable declaration, shown in the bottom-most item.

Although the above steps represent only the first segment of the THICKNESS task, there has already been significant opportunity for JASPER to save the user time and help the user recover from interruptions.

Figure 8 shows how JASPER could show the entire working set at the end of the THICKNESS task. JASPER has automatically arranged the items in Figure 8.
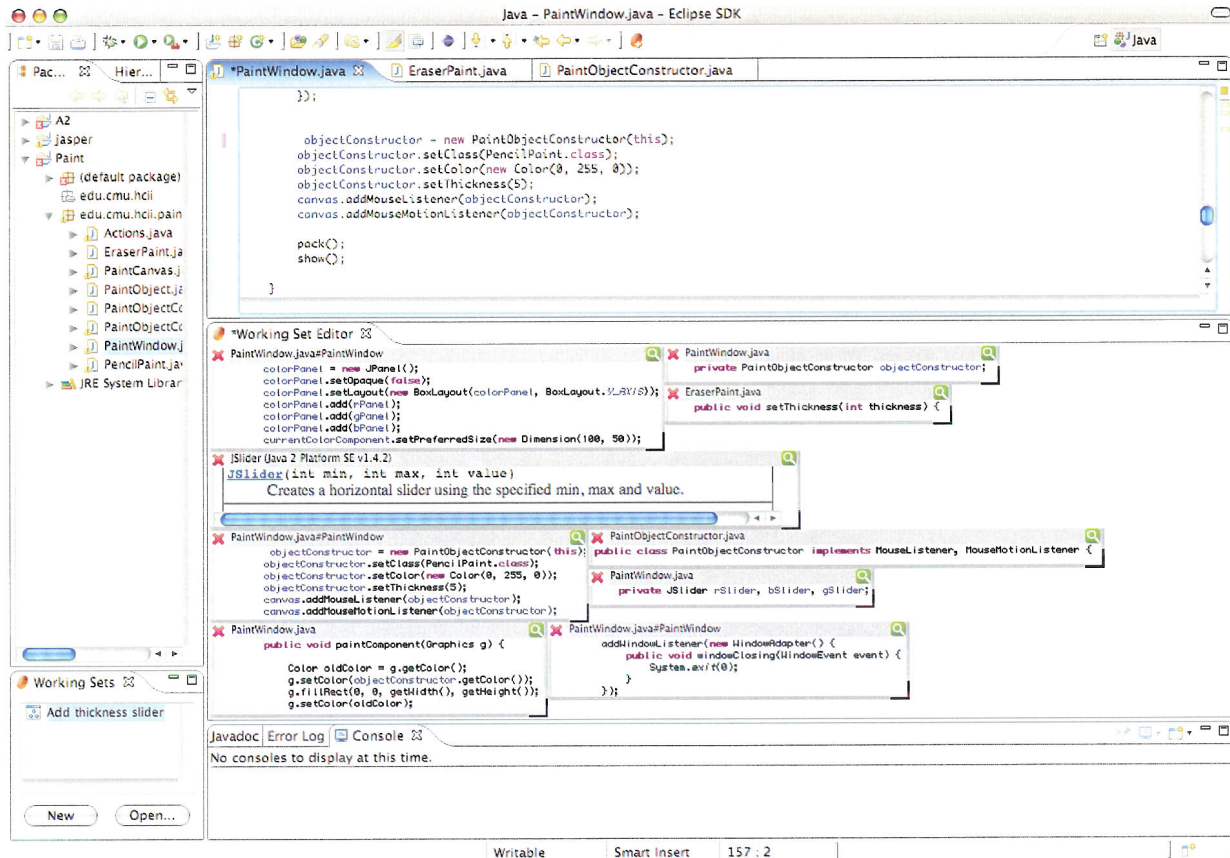
Figure 8: JASPER shows the entire working set for a user in [9] performing the THICKNESS task.

The pane at the bottom left in the above figures contains the list of working sets (currently only one is shown). The working sets list and working set views are Eclipse panes, so users can position them anywhere panes can go. Informal observations of programmers suggest that users need to see a lot of context, so they are likely to arrange the panes so that there is a large region for code and a smaller region for the working set, but JASPER allows users to arrange the panes as desired.

JASPER supports three kinds of working set items:

- *Java code* consists of a contiguous sequence of lines of source code from a file. If the code itself is edited, the text shown in the working set item is updated immediately as is described in

detail below in section 3.2. If the original file is edited external to Eclipse, the working set item is updated so that it displays the same code as before. The code is formatted exactly as it was in the original view, including syntax coloring and indentation, so it will look familiar to the user. However, the code might be shown using a smaller font, so more code will fit in the working set view at the same time.

- *URL items* uniquely identify a particular web page. The URL item displays the web page with scroll bars so that the user can choose which portion of the page to view. URLs are commonly used for referring to bugs (e.g. Bugzilla bug reports) and documentation. Javadoc is typically viewed in an external web browser, which would normally require frequent switching between the IDE and the browser. But JASPER allows users to keep frequently used items, such as documentation, visible in the IDE.

- *Text* items allow users to record notes and information about the task. These notes are editable within JASPER. Text items may be copied from text files or entered directly by the user using JASPER.

2.2. Creating New Items

There are three ways to create a new item:

- Dragging a URL into the working set creates a new URL item; dragging text in creates a new text item whose contents are initialized to that text.

- When a text editor or Java editor has the focus, an "add" button, , appears in the toolbar. When a user clicks this button, JASPER adds the selected text or Java code to the working set. The user can later retrieve the context of the item by double-clicking it or clicking a magnifying glass icon in its title bar.

- When a working set view has the focus, a "new note" button, , appears in the toolbar. When a user clicks the button, JASPER creates a new, empty text item.

Java items refer to code on a *line* granularity. When a user creates a new Java item, the item contains all of the lines that are in the current selection. This makes the code in the item visually clear: if the item could contain only parts of certain lines, then the text would have to be drawn in such a way that it was obvious that there was omitted text. Furthermore, this choice allows users to quickly highlight an *approximation* of the region they are interested in, making it more convenient and faster to define items.

2.3. Manipulating Existing Items

The interaction with existing working set items is modeled after the standard GUI windowing paradigm. In JASPER, working set items behave similarly to windows. This offers an interface that is familiar to the user and consistent with traditional interaction techniques. The working set view provides a workspace in which users may manipulate working set items; items may not leave the view. All items have a title bar, which shows information about Java items and URL items. Text items have title bars but do not currently have titles.

When users create Java items, JASPER automatically chooses a size that fits the contents of the item. However, JASPER does not have enough information to choose a correct size for empty text items (it cannot be known *a priori* how much text the user will type) and for web pages (the page is likely to be large, with only a small relevant portion). Furthermore, even though JASPER can size Java items correctly initially, further edits may change the appropriate size. Since the user may have manually positioned items, it would be inappropriate for the system to automatically resize items after changes. Furthermore, even if resizing were acceptable, it would be likely to obscure an adjacent item, requiring items to move automatically. But this would interfere with the user's spatial memory of the locations of items. Therefore, each item has a resize widget in the lower right corner.

Figure 9 shows an example of a Java working set item. In Figure 10, the user has recorded a note in a text item. The user has already added the thickness slider, and it works for the pencil tool. But it does not work for the eraser tool, so the note could be used as a reminder in preparation for an interruption.

Figure 9: A working set item representing part of PaintWindow.java.
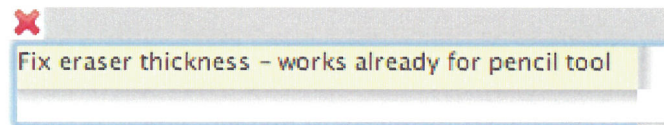
Figure 10: A working set text item.

Java items and URL items have a magnifying glass icon in the upper right corner (see Figure 9). Users can click on this icon to cause the item to be displayed in full in the original context. Text items do not have this icon because the text in text items does not refer to text stored elsewhere; the text item stores only a copy of the text.

Mouse actions on URL items behave as in the native browser, for example navigating links. However, mouse actions in Java and text items work as follows:

- *Click and drag* means "move," and causes the item to be moved with the cursor.

- *Double-click* means "Edit/reveal context." For text items, this enables editing. For Java items, this shows the original file and therefore has the same effect as clicking the magnifying glass icon.

The items behave as follows:

- *Java items* display a contiguous sequence of lines of code from a Java file. The magnifying glass opens a standard Eclipse Java editor for the file and scrolls it so that the item is visible. If an editor for that file is already open, JASPER re-uses it. Java items are not directly editable in

the item view. This is because, in informal observations of developers [9], they almost always preferred to see a significant amount of context when editing. Therefore, developers will almost never want to edit a Java item directly in the tiny working set view, so the space that would be required for all the controls necessary to control the view, such as scroll bars, is better used to display other items. When clicked, then, instead of displaying an insertion point, Java items can be dragged. This makes the drag region significantly bigger. Since it is expected that dragging will be a relatively common operation, there is a significant time savings as shown by Fitts' law [4].

- *Text items* are editable because they are intended to be used for taking notes and recording information. Their contents are owned by the working set. Thus, there is no magnifying glass icon, since the item is itself the original. Since text items are editable, they include scroll bars. This enables maximum flexibility: users can have many or only a few text items, and they can choose to have only a portion of certain items visible — some portions may be irrelevant. The scroll bars then serve as a visual indicator that some information is not currently visible.

- *URL items* behave as the platform-default web browser does. However, to reduce space requirements, there are no controls on the items, such as back and forward buttons.

2.4. Automatic Layout

Automatic layout is a crucial feature of JASPER. Manually positioning each item could be a time-intensive operation, especially since upon adding an item users might need to rearrange several other items to make room. To save time, then, JASPER automatically chooses a size and location for each new item. The size is chosen to be just large enough in each dimension such that the text of the item fits inside it. The position is chosen so that the new item does not overlap with any other item currently being displayed. Because choosing the position optimally is NP-

hard, JASPER uses an approximation of a solution to the bin packing problem. Details of how automatic layout works are in section 3.3.

The scale of the items is selected automatically when items are added or removed, and is constrained so that the text never gets too large. If an added item does not fit (i.e. the heuristic fails to find a space for it), then the scale of the entire working set view is decreased so that there will be enough space to fit all of the items. The font size is proportionally decreased, in addition to making the items smaller, so that all the text of the items remains visible. If an item is removed and the current scale is smaller than the maximum, the view is rescaled so that the items are as large as possible while still fitting in the view. Figure 11 shows how JASPER might display the working set from Figure 8 in a smaller pane. Each item is scaled down so that the entire working set is visible. Scaling also occurs when the working set pane is resized.
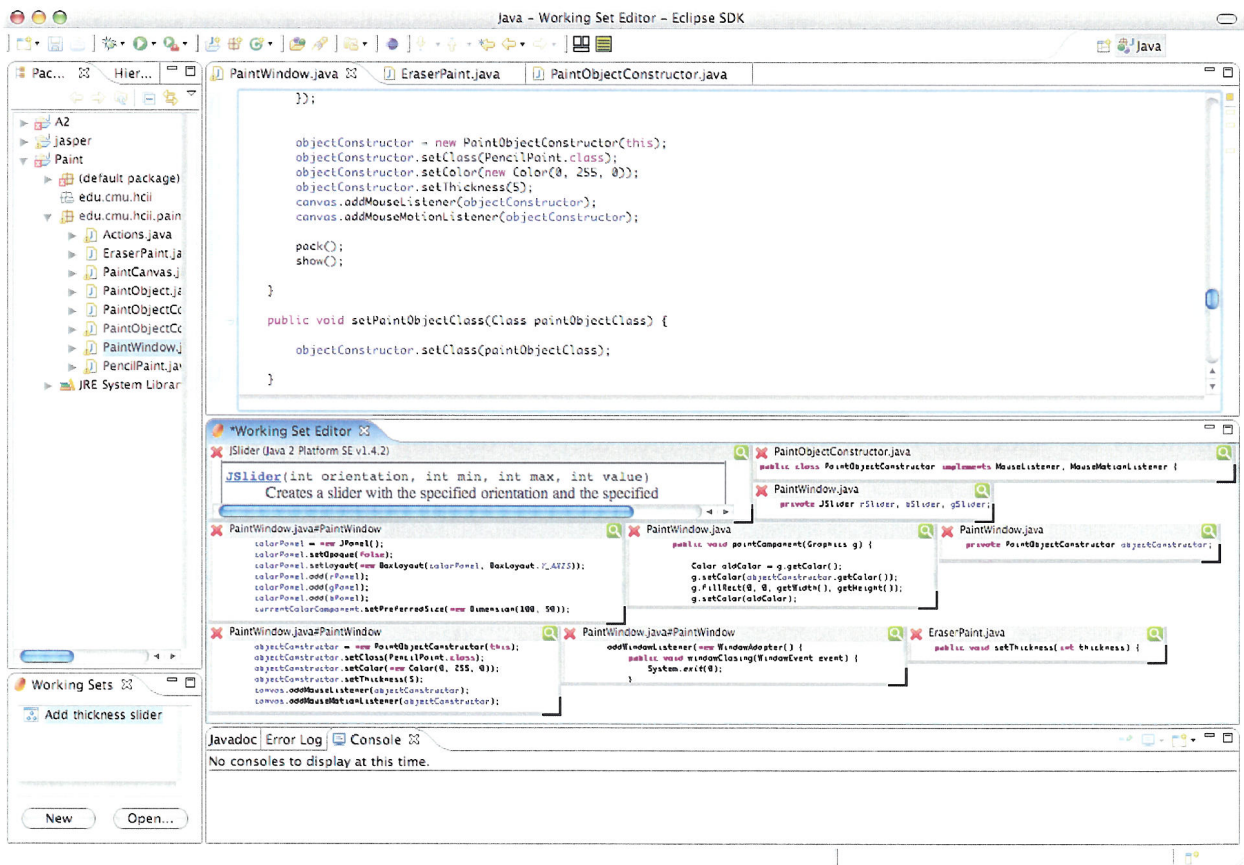

Figure 11: Same as Figure 8, but using a smaller working set pane.

When scaling working set items, the title bar text remains the same size — only the item content is scaled. This allows users to still distinguish items using the text in the title bar. If the title bar text were scaled, then items could become completely indistinguishable except for location and shape. In the extreme, it could become very difficult to click on the items. By not scaling the title bar text, JASPER ensures a minimum height for all items. However, if contents of items are not shown and the user continues to add items, then the titles continue to become narrower to accommodate new items. This would be unusual, however, since this is only necessary when there are several dozen items, which is significantly larger than the working sets seen in [9]. JASPER does not scale URL item content because of implementation difficulties: the SWT `Browser` widget does not support scaling.

This automatic re-scaling ensures that all working set items are always visible — at least for reasonable sizes of working sets. This was one of the primary goals of the system: make all information that the user has deemed relevant visible. Any automatic system that tries to infer which items to hide is likely to be wrong some of the time, potentially annoying and confusing the user. Since the user has explicitly indicated that each of the items is relevant, it would be surprising to the user to hide some of them.

If the user explicitly repositions an item or changes its size, the item is not moved or resized when items are added or removed. This enables users to develop a spatial memory of items. However, this behavior comes at a cost. The automatic layout algorithm uses a heuristic to partition the area into empty and nonempty rectangles. This heuristic may construct empty rectangles less efficiently when the items have been manually positioned, resulting in poorer auto-layout space utilization.

The incremental layout can be improved on: if the system could arrange all of the items at the same time, the heuristic is likely to produce a better arrangement. However, doing this automatically would result in an unexpected, counterintuitive change, since users are likely to

develop a spatial memory for the positions of the items. For this reason, JASPER only moves all items on user command. JASPER has an "auto-layout" button, which rearranges all of the working set items according to its heuristic. The button, ⊞, is visible in Figure 11 at the top.

Automatic layout does not arrange the items in a way that is particularly meaningful to users. Another approach that was considered was a one-dimensional layout instead of a two-dimensional layout. In a one-dimensional layout, all the items would be displayed in a list rather than in windows that are arranged in a two-dimensional space. This might make it easier for users to find items — especially new ones. However, items vary widely in width, so this layout would be likely to waste significant horizontal space.

Another approach to two-dimensional layout would be to display items in a table rather than in movable windows. If the table consisted of a grid, the same problem as with one-dimensional layout would occur: varying widths of items would cause wasted space. If the different rows in the table could have a variable number of working set items, this would be limiting because users could not easily position items near each other for comparison or for semantic reasons (e.g. users might prefer to position related items near each other). Even if the user could move items in the table, history has shown that users prefer window systems with overlapping windows rather than tiled windows. However, this table-based approach might be faster because it would allow users to quickly drag items to a location near their desired location instead of requiring users to position them exactly — items could snap to the grid when they were dropped. Many studies have been done on this problem (such as that described in [1]), so this warrants further investigation.

The existing automatic layout algorithm might be extended with an improved heuristic to make the layout more predictable, as discussed in section 6.

3. Implementation

3.1. Overview

JASPER was designed to work within Eclipse, which is primarily designed for Java. Eclipse was chosen as a target platform due to its maturity and extensibility. Eclipse has a plugin architecture that makes it relatively easy to implement and integrate additional tools, such as JASPER. Most of the components of Eclipse are implemented as plugins and are accessible to plugin developers. For example, this allowed the implementation of JASPER to use the same class for displaying code as Eclipse uses in its native code views, ensuring that code displayed in working set items is rendered the same as code in the standard view.

JASPER is a plug-in for Eclipse 3.2 and is implemented using SWT ("Standard Widget Toolkit"), which provides platform-native widgets on each of the various platforms that Eclipse supports. JASPER is implemented in approximately 3100 lines of Java code in 22 classes. The list of working sets (shown on the bottom left of Figure 9) is implemented as a `ViewPart` subclass. Each working set is displayed by an instance of an `EditorPart` subclass, which facilitates treatment of working sets as documents with respect to the save/load behavior. The working set items are subclasses of `Composite`.

In order to render the source code in Java items exactly as Eclipse does in its standard views, JASPER uses an instance of `CompilationUnitEditor`, which is the editor Eclipse uses for Java code. However, this is fragile, since `CompilationUnitEditor` is officially an internal class of Eclipse and not for use outside the Eclipse implementation. It was necessary, however, because there is no public class that provides the same functionality.

3.2. References to Code

Representing Java working set items presents a challenge, since the files containing the items may change frequently, especially on large development teams — evidenced by the ubiquitous nature of version control systems such as CVS. For example, if changes are imported from a version control system to a file that the working set refers to, the location of the item may change. In fact, the item itself may change: lines may be inserted or removed, and the item may disappear entirely. JASPER must detect these external changes in a robust fashion when the file is loaded and, where possible, repair references to working set items. To facilitate reference repair, references to Java code consist of the project name, a path within the project to the file containing the code, and the line numbers of the beginning and end of the item. In addition, JASPER stores as a string the text of the item as it is currently known each time the item is archived.

JASPER must be robust to changes, but alert the user if it was unable to find the referenced item. There must be some tolerance for imperfect matches so that the item will be maintained even if it has been edited. Therefore, JASPER performs a search in the new file for the code in the item as it was last saved. Every line in the new file is scored: +1 if it matches some line of the item, and -1 otherwise. Then, JASPER finds the region that defines the maximum contiguous subsequence sum of scores. Ties are broken by biasing the results toward the location of the original item. This allows for lines that do not match as long as enough of the surrounding lines do. If more than half of the found region was not in the original item, the user is alerted by displaying the found region with a bright red background. In Figure 12, the lines directly above the shown line were part of the working set, but the entire working set item contents were deleted from the file outside Eclipse.

If changes are made *inside* Eclipse, JASPER detects these changes and updates the working set item's reference to the changed code. JASPER adds itself as a listener to the document that

represents the file. Then, when a change is made to the file anywhere in Eclipse, JASPER is notified. The reference to the item is updated so that it still points to the same text. However, if the edit modifies the item itself, the reference is updated to include the union of the old item text and the new item text. This ensures that any lines the user marked as relevant are kept, and also that their *replacement code* is kept as part of the working set item. If the stricter approach of including only lines that existed before the change were taken, then as the user made changes, the working set items would slowly grow smaller and smaller, and would eventually not include the relevant code.
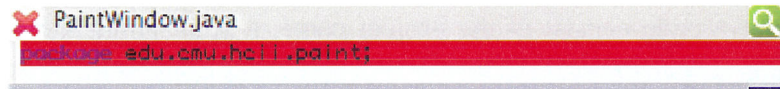

Figure 12: The working set item could not be found, so nearby text is shown with a red background.

3.3. Automatic Layout

Automatic layout determines locations for working set items so that the items can be positioned without overlapping and without users specifying locations manually. There are two versions of automatic layout in Eclipse. *Incremental* automatic layout determines the dimensions of and a location for each newly added working set item. *Complete* automatic layout moves all of the working set items so that none overlap, and is executed only upon user request.

Automatic layout is equivalent to the problem of bin packing, in which objects of different sizes must be assigned to a finite number of bins of capacity V so that the minimum number of bins is used. But bin packing is reducible from the subset sum problem, which is NP-hard, so automatic layout is NP-hard.

Because optimal automatic layout is NP-hard, automatic layout uses a heuristic. JASPER divides the available space into rectangles, and sorts these in order of increasing area. For each item that needs a space assigned, JASPER chooses the first in the sorted list that fits. The item is

placed in the upper left corner of this empty rectangle, and then the list of free rectangles is updated.

Sizes are chosen based on the content of the item. Each item has a "preferred size": the size at which the content of the item can be displayed completely at the normal font size. The incremental automatic layout algorithm scales each item to the current view scale and finds a space according to the heuristic described above. But if no space is found, JASPER enumerates all of the spaces at the edges and chooses the one into which the item fits most deeply, taking into account the dimensions of the item and the location of the space. If there are no spaces at the edge (an extremely rare occurrence), JASPER puts the item in the bottom left corner. Then, JASPER scales the entire view so that all items fit.

Complete layout uses an additional heuristic for choosing spaces: it chooses spaces for larger items first. The intuition is that the smaller items may fit between the larger items; of course, this only happens sometimes in practice. A side effect is that the items are completely re-ordered.

3.4. Extensibility of Eclipse

Eclipse is an immensely extensible environment for software development. Extension points allow plug-ins to add functionality to many different features of Eclipse. This permitted development of JASPER, which would not be possible in most other existing IDEs. However, development was greatly hindered by the poor documentation of Eclipse and SWT. The class used to display Java code (`CompilationUnitEditor`) is an internal Eclipse class, not for public use, but it was needed to use it in order to make Java items look the same as the original code. Documentation is frequently only available in the form of JavaDoc for individual classes and some tutorials; information about the architecture of the system is difficult to find. This makes answering questions like "which class should I use for X?" and "I have an instance of class X; how do I get an instance of class Y?" very difficult.

Plug-ins indicate to Eclipse how they integrate with the platform via an XML file. Unfortunately, the tags used are poorly documented also. The documentation for some tags have a "not yet implemented" warning, but no reference to the appropriate tag to use for that functionality. Errors in this XML file are nearly impossible to debug because Eclipse gives no feedback when errors occur. For example, adding a button to the toolbar was a difficult, arduous task because it is necessary to fill in several attributes of an XML tag, and when the wrong values are given, the only feedback is that the button does not appear.

The SWT drawing system lacks basic features like transparency and a strikethrough text style, so some approaches to showing working set items that we considered were impossible. A two-week-long attempt to port the system to Draw2D and GEF (the Graphical Editor Framework, an Eclipse plug-in for building graphical editors in Eclipse) ended in failure because of poor documentation, complex model requirements and complex interactions among classes, and great difficulty in customizing default functionality of GEF. Hopefully in the future, developers of Eclipse will focus more on creating high-quality documentation.

4. Discussion

The features of JASPER are implemented for Java in Eclipse, but are not unique to either Eclipse or Java. JASPER is implemented with SWT, which, although providing significant cross-platform capabilities, does nothing crucial for JASPER that any other widget toolkit would not. In fact, more extensible GUI toolkits may be much easier to use; for example, instantiating a Java text viewer in the working set items required using an internal, undocumented Eclipse class that is not normally available for public use. JASPER's user interface does not require any particular features of Java, aside from the fact that it is a text-based language that consists of

lines of code (although displaying titles for Java items is implemented using the parse tree of the Java code).

The field of aspect-oriented programming has demonstrated that the problem of cross-cutting concerns is not restricted to Java; it exists in any programming language that people use to create large or small software systems. Each architectural choice promotes modifiability of some aspects and inhibits modifiability for some others; this promises to remain the case for the foreseeable future.

JASPER is more useful for some tasks than others. Some modification tasks are done by a programmer who already understands all of the relevant code and needs only make the necessary edits. JASPER is not likely to be helpful with these kinds of tasks because users do not need to refer to any code. However, these tasks are extremely rare, especially with large projects. Any project with more than one programmer almost certainly has code that is unfamiliar to some of the developers who will need to interact with it. Furthermore, programmers frequently forget the details of code that they wrote even in the recent past, so JASPER can be useful even on single-person projects.

Further study is necessary to determine whether JASPER can accommodate the working sets for complex tasks in very large projects. One limitation of the study in [9] is that the project was only 508 lines long. Also, the program used in that study was relatively well organized, according to some of the participants. It might be the case that some tasks in large projects have working sets too big for all the relevant items to be simultaneously visible at a readable font size in JASPER.

The utility of JASPER depends on the existence of tasks. However, in most large software development groups, there is an organized change tracking system. Even using very informal development processes, such as in some very small groups, users can divide their work into coherent units. It is possible, however, that in very informal contexts, users may not divide their

work into tasks; JASPER may be less useful for these users. Of course, they may still construct working sets in preparation for interruptions or in order to share information with colleagues.

Related to the question of the existence of tasks is the question of whether users will delete items from working sets when they are no longer relevant. Probably this question will need to be answered with experimental data. However, if the system were to automatically infer irrelevance, it would be likely to be wrong in some cases. If the system automatically removed items, users would probably find that items they were using were suddenly vanishing. There are more subtle approaches, such as recommending items for removal, or gradually shrinking or moving to the side potentially irrelevant items. These approaches could be useful, but great care would need to be taken to avoid disrupting the user's spatial memory by moving items or changing their appearance.

A working set is a kind of a document: working sets can be edited and are persistent. In most graphical systems, undo is traditionally implemented because users sometimes make mistakes. Undo is useful for reducing the burden of correcting these mistakes. Most windowing systems do not have undo features for manipulating windows, however, and JASPER does not have an undo feature either. In JASPER, there are four operations users can do: add an item; remove an item; move/resize an item; and relayout all items. Adding an item is already easily undone by closing it. Moving and resizing are not particularly difficult to undo manually either, although automatic undo might be useful. However, sometimes users may mistakenly remove a relevant item. This is a case where undo would be useful. Another way to accomplish this would be to store a history of all working set items; then, users could look in the history for the lost item. Although this would be less convenient than undo, it would be more flexible because items that were discarded in the distant past could be recovered easily. It might also be useful to be able to undo automatic layout, but this has only limited utility because the consequences of an undesired automatic

layout are not severe: one could rearrange the items manually, and only spatial information is lost.

5. Related Work

This work was inspired by the study described in [9]. In that study, the experimenters observed users performing maintenance tasks on a small drawing program written in Java. They found that programmers spend a significant fraction of their time (35%) performing routine navigations between sections of source code. JASPER is designed to address this shortfall in existing programming environments.

There is other research that helps programmers collect and view frequently used artifacts. Mylar [7] is the most closely related work. Like JASPER, Mylar displays task-relevant data to reduce the time that programmers spend on navigation. However, Mylar displays data as a list of potentially relevant elements of the Java model, such as classes, methods, variables, and other syntactic elements. Mylar uses a degree-of-interest measure to automatically determine which elements are likely to be relevant to the current task.

One significant difference between JASPER and Mylar is that Mylar is restricted to syntactic Java elements, but JASPER can represent any contiguous region of code as well as text and URLs. Mylar cannot represent, for example, the line containing the test of a `for` loop and the following three lines of code. Another difference is that JASPER keeps the contents of all working set items visible, but Mylar hides all but the primary syntactic element until the user selects it. JASPER's approach allows users to more easily recognize and find relevant items, since they can form a spatial memory of the locations and appearances of the items. The automatic inference may be a benefit to Mylar, but it may also be a hindrance. Users will not be able to predict whether Mylar will infer the relevance of a given element, so when trying to

navigate to an item, they must first check Mylar's list, and if it is not there, then use the main list in the package explorer. This unpredictability may reduce Mylar's effectiveness. JASPER is predictable because it does not add or remove items automatically.

FEAT [12] is another tool that helps users navigate to and among relevant artifacts. In FEAT, users create concern graphs, which represent *concerns*, similar to the notion of working sets. Like Mylar, concerns consist of syntactic elements. Users must learn to use FEAT's interface to navigate various kinds of dependencies among elements to locate relevant elements. The result is a graph of elements where the edges in the graph are relationships between elements. However, these elements are even more restricted than in Mylar, since they can only be classes, methods, and fields. But the study in [9] showed that relevant artifacts are not limited to those kinds of items; sometimes only a line or two of a large method implementation is relevant. Furthermore, FEAT does not facilitate viewing the contents of many elements in a concern simultaneously, as JASPER does. An extension to FEAT, described in [14], allows the system to automatically infer concerns from a user's interactions with the IDE. FEAT, in addition to helping navigation, helps users find relevant artifacts. JASPER does not have this goal, since many of the dependencies expressible in FEAT are already navigable in Eclipse.

A query language for code was considered for JASPER but rejected. This would have had several significant benefits. First, queries could be made to be much more robust to certain kinds of code edits. Queries could reflect the syntax of the code in addition to the lexical information that JASPER uses. This would make the items robust to many kinds of structural and lexical changes, such as variable renaming, method extraction, etc. Second, queries could enable working set items to reflect semantic data. For example, one could have a working set item that contains all of the calls to a particular method. This would be a powerful feature. There is already evidence that many people would use it: in [9], the authors showed that programmers frequently navigate several types of direct dependencies, such as "caller-of" and "declares." Furthermore,

support for these searches already exists in Eclipse; users can easily retrieve the list of all callers of a particular method using an Eclipse command, and it might be appropriate to display the resulting list as an item of the working set. If useful, this might be implemented in a future version.

Another benefit of queries is that users could potentially edit the queries themselves, resulting in accurate, meaningful data. One could implement a graphical interface for constructing and editing queries to reduce the necessity of learning a textual query language, as in [13].

Queries, however, have several significant disadvantages. Either users must create the queries manually or the system must derive them automatically. Manual creation would be much more burdensome than selecting a region and clicking one button. Automatic derivation may be error-prone, generalizing inappropriately, or reliable but no better than the simpler representation chosen for JASPER because they would never generate general queries. Furthermore, the data in [9] suggests that a simple representation of lines of code suffices. Therefore, it was decided to have JASPER use a simple representation, and leave integration of a tool such as FEAT to the future.

Another tool related to FEAT, called JQuery, is described in [6]. JQuery is a query language for Java code to help users visualize the structure of a search through software. However, JQuery does not help manage tasks or working sets. Instead, it helps users record the history of their searches through the source code. The resulting representation does not directly represent the working set; it instead represents the search.

Eclipse has, as of version 3.1, a feature already called "Working Sets," but it is very limited. It essentially acts as a filter on the existing Package Explorer, limiting the view to files or projects chosen for a particular working set. Although this may help, its functionality is extremely restricted.

Desert [11] uses a relational database to store artifacts, and displays relevant artifacts in a window derived from FrameMaker, a word processing tool. However, users must edit text to specify references artifacts, so it is cumbersome to add and remove them. Furthermore, items are displayed in a purely textual form — layout is in one dimension rather than two. Desert was created before the Internet was popular, so it has no support for URL items.

WinCuts [16] lets users capture small sections of windows and manipulate them mostly independently of the original windows. Users of WinCuts can then easily compare distant regions, even on large screens. But WinCuts provides no facility for easily arranging and manipulating these views, and the views are graphical references rather than semantic references. For example, if a WinCut was made of some source code, and the containing file were edited so that the original item were no longer in the same place, the WinCut would show whatever code the window now was rendering at the original graphical location.

Much work has been done to facilitate *finding* relevant code. Hipikat [2] and TeamTracks [3] help users find relevant artifacts using data from colleagues. Relo [15] helps users explore large codebases and find relevant artifacts within. Any system, such as JASPER, that helps users view and navigate code requires them to first collect it, so tools that help users in the search and collection process would work nicely with JASPER.

6. Future Work

An important step in this work should be a formal user study. There are many forms this could take, depending on the particular objectives. To determine whether users are more productive with JASPER than without, one could repeat the experiment in [9], but give the users JASPER and measure whether more tasks were completed. However, JASPER is most useful with very large projects, where it is very difficult to remember enough to complete each task.

This kind of study would also evaluate a very limited subset of JASPER's goals, since the study in [9] included only developers working alone.

Another question that should be investigated is whether people will choose to use JASPER, which might be a prerequisite for JASPER's utility in software development. This may not be correlated with whether it improves their individual productivity, but instead may depend on broader perceived benefits. One way to evaluate this would be via an Internet release. A version of JASPER could be released that recorded anonymous usage statistics which, after receiving the user's explicit permission, would be sent to the experimenters for analysis. One could evaluate by this method whether users find JASPER to be a useful tool. In addition, this would be a valuable way to gather direct feedback from users. Users could report on their experiences with the tool. In addition to gathering data about how to improve it and how useful it is, as with other tools, there are likely to be uses of JASPER that have not yet been foreseen.

A general release would require some additional implementation work. In addition to improving stability of the implementation and writing documentation, it would be necessary to implement a logging system so that experimenters can gather data on usage. Currently, JASPER requires Eclipse 3.2. Modifying it so that it could run in Eclipse 3.1 might significantly enlarge the community of potential users, since Eclipse 3.2 was only just released at the end of June 2006. Most of the development of JASPER occurred on Eclipse 3.2 pre-release builds because Eclipse 3.2 is the first release of Eclipse that has native support for Intel processors on Mac OS X.

It might be useful to add other ways to add items to working sets. Mouse gestures could be used to select items; for example, circling a block of text could add the circled text to the working set. There should be a keyboard shortcut for creating working set items. A choice should be added to the contextual menu for code, text, and web pages to add the selected item to the

working set. URLs can already be dragged and dropped in the working set view, but it should be possible to do so with text and Java code as well.

In addition to these manual methods of adding items to working sets, there may be automatic approaches that are not excessively obtrusive. A view could always display several automatically-derived items not in the current working set, and a button would let users easily add the inferred items if they are relevant. The system could also take advantage of navigations that users already perform — for example, automatically adding any items that are edited. Machine learning approaches may be able to predict with sufficient precision which items are relevant, and add them automatically.

JASPER could also use the sequence of navigations to help users add items in retrospect. Some developers may forget to add items to the working set until after they navigate away from them; when they want to return to these locations, JASPER could help them easily add the forgotten items.

JASPER should be integrated with other research systems that complement it. FEAT [12] and JQuery [6] could allow much more versatile and robust working set items. JASPER could allow working set items to contain arbitrary FEAT concerns. The references to code could also be improved by enabling references to be FEAT concerns. In addition, JASPER could be integrated with an artifact recommender, such as Mylar or the recommender for FEAT. A special kind of working set item could always show the system's current recommendation of relevant code, and a single click could add the suggested item to the working set.

Tools that improve collaboration features of working sets could complement JASPER. For example, users might like to know how changes in the codebase affected their working sets. Perhaps a bug was introduced because someone changed a working set item but did not notice a dependency on another working set item. Working sets could be used to suggest related code to

existing working set items. This could be more specific than TeamTracks, as discussed in section 1.3.

JASPER could be generalized so that other tools could use its artifact reference system. Other tools could read and write working sets. There may be interesting and useful visualizations of working sets, especially across a large number of developers or across time. These kinds of analyses would be supported by integration with a version control system such as CVS or Subversion.

In the future, organizations may collect large amounts of data on working sets over time for many developers. This would enable additional studies of working sets and the nature of programming. Developers sometimes use very different strategies for making changes. Analyzing working sets could help reveal which strategies are most effective, since some strategies could be correlated with working set features. For example, it may be found that although insufficient dependency analysis is a common cause of bugs, automatic recommendation systems can recommend relevant artifacts. Likewise, one could use data from JASPER to evaluate recommendation systems using a method analogous to leave-one-out cross-validation: for each artifact in a working set, give the recommender the working set without that artifact, and see whether the recommender recommends the missing artifact.

Data on working sets could also be used for various comparisons. Perhaps data could be used to infer information about which programming techniques were more efficient or which languages or programming styles required programmers to know about more relevant data. It might be useful to analyze architectural patterns with respect to the working sets they induce.

Aggregate working set data could provide design guidance for future code navigation tools. It could give statistics about the sizes of working sets and of working set items, and also the frequency with which users return to items.

Working set items could be enhanced to comprise more types of items, such as emails, instant message windows, chat sessions, and graphic assets used for the project, such as icons. The URL items could be improved so that they can reference a particular region on the rendered page, using a tool such as WinCuts [16]. Text items could be extended to allow references to text in addition to copies of text. This would require some care, however, since if such referential text items looked like text notes, the difference might be confusing to users, who would be surprised when the original text changed unexpectedly. One way to do this would be to instead extend *Java* items to be able to display non-Java code. A user of a preliminary version of JASPER wanted to create items for SQL code. This would be a useful extension of the system, especially if the extension displayed many different kinds of code appropriately. Since JASPER does not have support for SQL items, the user used a text item instead.

JASPER could visually indicate relationships among items. For example, some items may call code in other items, document APIs used in other items, define symbols used in other items, etc. Lines or arrows could be drawn between items to indicate these relationships, and the automatic layout algorithm could use these relationships as a hint for positioning them. Some relationships could be automatically inferred, especially since the system could take advantage of the path the user took to find each item. For example, if a user originally found an item by looking at the list of callers of a particular method, and the method was in the working set, this caller/callee relationship could be shown.

Existing automatic layout in JASPER positions items according to a heuristic that tries to maximize space efficiency. However, this may be confusing for users, who might have difficulty finding items because the items are not organized in an intuitive fashion. Automatic layout could be extended to position items in a more predictable order, such as top to bottom or left to right. An improved heuristic might maximize the space efficiency under these conditions. Likewise, a visual indicator might be helpful for showing users where new items have been placed. Items

could also be annotated with information, such as time and date created or modified, that might help users identify them more quickly.

When a working set is loaded, the user is warned if the match score of the found item is less than half the number of lines in the original item. This criterion of half should be replaced with a value that is based in theory or experimental evidence; half is a heuristic that may work well in many cases, but there may be better choices. Also, because matches occur on a line-by-line basis, the matching algorithm is easily defeated by small changes such as adding tabs to each line. The matching algorithm could be extended to consider changes in content of the lines to make it more robust to this type of change, which is common when refactoring. For example, moving a section of code into a new method may change its indentation level. Also, the algorithm could be made robust to large insertions in the middle of items — currently, JASPER is likely to only find the larger end of the item in this case. Finally, the warnings could be extended to provide more useful information. For example, users could be informed of the size of the change or the certainty with which it was found. Currently, the background is colored red to warn users; the shade of red could depend on this certainty. It may be helpful to also show the previous working set item's text, with a warning, even if it cannot be located.

7. Conclusions

JASPER represents a new approach to navigating among and collecting task-relevant data when programming. It promises to significantly reduce time spent navigating to previously-seen artifacts as well as reduce the detrimental impact of interruptions while programming. JASPER is likely to facilitate group work, which is ubiquitous among programmers, and may in the future help reveal information about the nature of programming and software engineering. Basing the design of JASPER on previous user studies of actual programmers [9] has led to additional

insights into its design and features. In the future, user studies should continue to influence the design of tools for programmers.

8. References

[1] Bly, S. A. and Rosenberg, J. K. 1986. A comparison of tiled and overlapping windows. SIGCHI Bulletin. 17, 4 (Apr. 1986), 101-106.

[2] Čubranić, D. and Murphy, G. C. 2003. Hipikat: recommending pertinent software development artifacts. In Proceedings of the 25th international Conference on Software Engineering (Portland, Oregon, May 03 - 10, 2003). International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 408-418.

[3] Deline, R., Czerwinski, M. & Robertson, G.G. (2005). Easing Program Comprehension by Sharing Navigation Data. In Proceedings of VL/HCC 2005, 241--248.

[4] Fitts, P. M. (1954). The information capacity of the human motor system in controlling the amplitude of movement. Journal of Experimental Psychology, volume 47, number 6, June 1954, pp. 381-391.

[5] Gonzalez, V. M. and Mark, G., "Constant, Constant, Multi-Tasking Craziness": Managing Multiple Working Spheres, CHI 2004, Vienna, Austria, 113-120, 2004

[6] Janzen, D. and De Volder, K. Navigating and querying code without getting lost. In Proceedings of Aspect Oriented Software Development, Boston, 2003, 178-187.

[7] Kersten, M. and Murphy, G. C. 2005. Mylar: a degree-of-interest model for IDEs. In Proceedings of the 4th international Conference on Aspect-Oriented Software Development (Chicago, Illinois, March 14 - 18, 2005). AOSD '05. ACM Press, New York, NY, 159-168.

[8] Kiczales, G., et al. Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming. Springer-Verlag, Finland, 1997, 220-242.

[9] Ko, A. J., Myers, B. A., Coblenz, M. J., Aung, H. H. (2006). An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. IEEE Transactions on Software Engineering, to appear.

[10] Perlow, L., The Time Famine: Toward a Sociology of Work Time, Administrative Science Quarterly, 44, 57-81, 1999.

[11] Reiss, S. P. The Design of the Desert Software Development Environment, International Conference on Software Engineering 1996, Berlin, Germany, 398-407.

[12] Robillard, M. P. Representing Concerns in Source Code. Ph.D. Thesis. Department of Computer Science, University of British Columbia. November 2003

[13] Robillard, M. P. and Murphy, G. C. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. ICSE 2002.

[14] Robillard, M. P. and Murphy, G. C. Automatically Inferring Concern Code from Program Investigation Activities. In Proceedings of the 18th International Conference on Automated Software Engineering, 2003, 225-234.

[15] Sinha, V., Karger, D., Miller, R. Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. Visual Languages and Human-Centric Computing (VL/HCC 2006). Sep. 4-8, 2006, Brighton, United Kingdom, to appear.

[16] Tan, D. S., Meyers, B. & Czerwinski, M. (2004). WinCuts: Manipulating arbitrary window regions for more effective use of screen space. In Extended Abstracts of Proceedings of ACM Human Factors in Computing Systems CHI 2004, p. 1525-1528