Automatic Juxtaposition of Source Files

by

Samuel Davis

B.Sc., The University of British Columbia, 2006

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August, 2008

Abstract

Previous research has found that programmers spend a significant fraction of their time navigating between different source code locations and that much of that time is spent returning to previously viewed code. Other work has identified the ability to juxtapose arbitrary pieces of code as cognitively important. However, modern IDEs have inherited a user interface design in which, usually, only one source file is displayed at a time, with the result that users must switch back and forth from one file to another.

Taking advantage of the increasing availability of large displays, we propose a new interaction paradigm in which an IDE presents parts of multiple source files side by side, using the Mylyn degree-of-interest function to dynamically allocate screen space to them on the basis of degree-of-interest to the current development task. We demonstrate the feasibility of this paradigm with a prototype implementation built on the Eclipse IDE and note that it was used by the author over a period of months in the development of the prototype itself. Additionally, we present two case studies which quantify the potential reduction in navigation and demonstrate the simplicity of the approach and its ability to capture complete concerns on screen. These case studies suggest that the approach has the potential to reduce the time that programmers spend navigating by as much as 50%.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor, Gregor Kiczales, for his guidance, patience, and support, and for encouraging me to pursue this degree in the first place. I am in awe of his ability to distill the essence of an idea, and grateful for the way he engages with his work and his students. I would also like to thank Gail Murphy for being my second reader and providing helpful comments on this work. Finally, I thank my family and friends for their valuable advice and support.

Chapter 1

Introduction

1.1 Motivation

Most code comprehension and code modification tasks involve reading parts of and possibly editing multiple source files. Previous work [8] has identified the ability to juxtapose arbitrary pieces of code as cognitively important. However, modern IDEs have inherited a user interface design in which, usually, a single source file is displayed, along with related views such as a code outline. This single file interaction paradigm requires the user to physically select the one source file to be viewed or edited at any given time, replacing the file that was previously visible on the screen. As a result, any non-trivial code comprehension or code modification task involving more than one source file requires the user to switch back and forth from one file to another. In fact, Ko et al. [17] reported that software developers performing maintenance tasks on a small code base spent an average of 35% of their time navigating between different source code locations and that much of that time was spent returning to previously viewed code. Because each of these navigations involves a scene change, that is, the entire contents of the editor area is completely replaced, the user may suffer from a loss of context and become disoriented. Furthermore, when following a chain of cross-references in the code through multiple files, the user may even lose track of her exploration path and spend time trying to remember or rediscover what she was previously looking at.

JQuery [12] addresses the problem of disorientation resulting from scene changes at the level of structure browsing. It allows the user to trace a variety of types of relationships among program elements within a single view, using queries to direct the exploration task. The authors argue that it

reduces disorientation when compared with traditional IDE structure browsing support, which requires the user to switch to a different view whenever she needs to examine a different type of relationship. However, this does not operate at the source code level: when the user needs to actually read or edit the code, she must still switch between files in the traditional way. While the JQuery view records the exploration path and may provide an easier means of performing this navigation, the problem of scene change remains.

IDEs such as Eclipse[1] support code popups, where the target of a cross-reference (e.g. a method call) is displayed in a tooltip on top of the editor. However, these tooltips are transient and non-editable, and because they appear at the source of the cross-reference, they often cover up information crucial to understanding the contents of the tooltip itself. They also do not allow the user to follow cross-references from within the tooltip, so they only provide access to code that is directly referenced from the current file. Fluid source code views [6] allow the user to expand cross-references inline, for example, displaying a method or advice body as though it were defined at the location at which it is invoked. These views are less transient than popups and allow code from multiple files to be displayed in a single editor. However, they do not take advantage of today's wider computer screens and may make the file containing the cross references harder to understand. Like popups, fluid views display code orphaned from the context of its own file. In some cases, this lack of context may impede understanding of the orphaned code. Furthermore, there is concern [6, 13] that this kind of approach could encourage programmers to view a program simply as a sequence of statements and to think of fundamental abstractions such as inheritance, procedures, and advice merely as code structuring techniques. While [6] mentions the possibility of making fluid views editable, one wonders if programmers would sometimes forget that the code is not actually local and make changes with broader consequences than intended.

A study of programmer behaviour during a code comprehension and modification task [19] found that developers who successfully completed the task reinvestigated previously examined methods less frequently than un-

[1] http://www.eclipse.org/

successful developers. However, even the successful developers performed a substantial amount of reinvestigation. Among other things, the authors hypothesize that developers would benefit from tools which help them to remember relevant methods.

Ko et al. [17] proposed a model of code comprehension in which a developer repeatedly goes through a process of choosing a starting point ("searching"), navigating dependencies in the code ("relating"), and "collecting" information and source locations deemed relevant, either in her head or using some external memory aid or tool. One of two factors identified as being key to the effectiveness of the comprehension task is the presence of a reliable way of collecting information.

Mylyn [14, 15], discussed in Section 2.1, is an Eclipse plugin that automatically maintains a list of program elements that the user has considered might be relevant, however, the user is only presented with the names of each element and the containment relationships between them. When that is not enough information, she must still individually select elements from the list in order to view their source code. Thus, while Mylyn does help the user to collect information in the form of a list of program elements, it does little to help her make sense of that list.

1.2 Displaying Multiple Source Files, Side by Side

The single file interaction paradigm makes sense if one assumes that displaying a single source file will require most of the available screen space. However, increasing screen sizes (and falling prices) create new possibilities of which IDEs should take advantage. An IDE which uses a large screen to display the relevant parts of multiple files side by side and in context could reduce the time spent on redundant navigations while allowing the programmer to use her spatial memory to track the locations of relevant code on the screen. Presenting each piece of code in the context of its original file may provide useful contextual information to the user and should be conducive

to a more breadth-first style of code exploration; [21] discusses research suggesting that breadth-first strategies lead to solutions more quickly, both when programming and more generally.

Displaying multiple files at once has the potential to reduce the disorientation associated with jumping between code locations and could support short term waypointing [23] by allowing the user to remember relevant code in terms of its location on screen. It also naturally supports visual comparisons of source code. Allowing the programmer to see related program elements within and across different files at the same time could make it easier to understand how the elements are related and how they interact. This may be especially beneficial in the context of aspect-oriented programming [16], where the program contains elements that directly modify other program elements.

Finally, this approach also creates the opportunity for visual annotations that span multiple source files. For instance, arrows could trace method calls or connect advice to joinpoints, and occurrences of the currently selected identifier could be highlighted in every visible file.

1.3 Thesis Statement

An IDE which simultaneously displays parts of multiple files, allocating screen space to each on the basis of degree-of-interest to the current task, can display all (or much of) the code needed to complete a task at once. This has the potential to reduce time spent on navigation and to ease concern comprehension.

1.4 Contributions

This dissertation proposes a new interaction paradigm in which an IDE presents parts of multiple source files side by side on a large display, dynamically allocating screen space to them on the basis of degree-of-interest to the current development task. We demonstrate the feasibility of this paradigm with a prototype implementation built on Eclipse and note that it was used

by the author over a period of months in the development of the prototype itself. Additionally, we present two case studies which quantify the potential reduction in navigation and demonstrate the simplicity of the approach and its ability to capture complete concerns on screen.

The remainder of this document is organized as follows. The next chapter describes our prototype and Chapter 3 details salient features of its implementation. Case studies are presented in Chapter 4 and the potential benefits and drawbacks of the system, as well as the experience of its sole user (the author) are discussed in Chapter 5. Chapters 6 and 7 discuss related and future work and we conclude in Chapter 8.

Chapter 2

A Prototype in Multiple File Interaction

To test our hypothesis, we constructed a prototype implementation as a plugin for the Eclipse IDE, targeting Java and AspectJ. The primary objective of the implementation is to reduce the time users spend on navigation by keeping multiple source files visible simultaneously. Since this necessarily entails some automatic editor hiding, we adopted the philosophy that when an editor must be hidden, it should be easy for the user to find, and we try to avoid hiding editors that the user would be likely to still want to see. The prototype uses arrows to indicate important relationships between program elements both within and across files, and it aims to take advantage of the user's spatial memory by keeping editors in the same location as much as possible.

Another important objective for the prototype was that it should integrate well with Eclipse. Thus, it uses the standard Eclipse editors and commands and is meant to be a natural extension of the Eclipse user interface, rather than requiring users to learn a radically new interface. As much as possible, it should not require the user to make any extra effort, so as to prevent laziness from being a reason not to use it. Finally, for evaluation purposes, the prototype needed to function well enough that it could be used by the author in its own development.

2.1 Mylyn: Measuring Degree-of-Interest

The prototype is built on Mylyn [14, 15], using its *degree-of-interest* function to allocate screen space to editors. Mylyn is an Eclipse plugin which

enhances Eclipse's built-in views with the ability to focus on the current task. Mylyn keeps track of the user's tasks and maintains a *task context* for each. The task context records which files and program elements are relevant to a task and how interesting they are. The user indicates which task she is currently working on and Mylyn records the history of her interactions while working on that task, in terms of which files and program elements she selects or edits. Each file or program element is assigned a task-specific and time-varying degree-of-interest based on how frequently and how recently she has interacted with it. This allows Mylyn to filter less interesting elements out of the Eclipse Package Explorer and other views, reducing clutter and helping the user to keep track of which elements are important to the task she is working on. It also allows her to switch to another task without losing the current task context. When returning to a task, Mylyn saves her from needing to remember or rediscover its context. Of course, there may be other information about the elements in the task context that she still needs to remember, but having to look only at the small fraction of the system which is relevant to the task is intended to make this much easier.

2.1.1 Automatic Code Folding

Mylyn provides a command which elides the bodies of uninteresting methods using Eclipse's built-in code folding support. If the user does not need to see the source code for elements which have not been added to the current task context (for instance, if she feels that it contains enough of the code that is relevant to the task), she can enable automatic folding. This causes any elements which Mylyn deems uninteresting to be collapsed so that only their headers are shown. If desired, the user can then expand an individual element to see its body. If a collapsed element is selected or navigated to using an outline view or a cross-reference it will be automatically expanded.

A potential negative consequence of using this command is that annotations in the bodies of collapsed methods are not shown in the sidebar. For example, when using Eclipse's *Mark Occurrences* feature to highlight occurrences of an identifier within a file, occurrences within collapsed methods

will not be shown, potentially causing the user to draw inaccurate conclusions. While this is really an issue inherent in Eclipse's code folding, it may be more problematic when a tool automatically folds code without explicit action by the user to indicate which code should be folded.

2.2 Features of the Prototype

2.2.1 Multiple Editors

Figure 2.1 shows a screenshot[2]. Unfortunately, to allow them to fit legibly on the page the screenshots in this chapter had to be taken on a much smaller screen than the prototype would actually be used with, and other views such as the Package Explorer are hidden. The prototype arranges editors in three columns (there is also a two column mode suitable for smaller screens). Within each column, space is allocated to each editor based on the degree-of-interest of the file it displays. When more editors are open than can fit on the screen, some are hidden, appearing only as tabs behind other editors. Section 4 contains four series of screenshots that give some sense of the way the display evolves, though the images only contain two columns to allow them to fit on the page.

The set of files that are visible is directed by user actions. When only one file is open, it fills the editor area, but as the user opens more files, whether by selecting them from a list or by following cross-references, the display becomes divided into columns, and the columns are eventually divided into rows. Each editor is guaranteed to receive at least 20% of the height of the Eclipse editor area, so there can be at most five files displayed in a single column. Typically, the number of files displayed is smaller than this maximum.

The Mylyn degree-of-interest is used to allocate screen space to editors and, when required, to choose which editors are hidden. As files become more or less interesting over time, their editors are resized accordingly, but

[2] All screenshots in this chapter use the AJHotDraw code base, available at http://swerl.tudelft.nl/bin/view/AMR/AJHotDraw

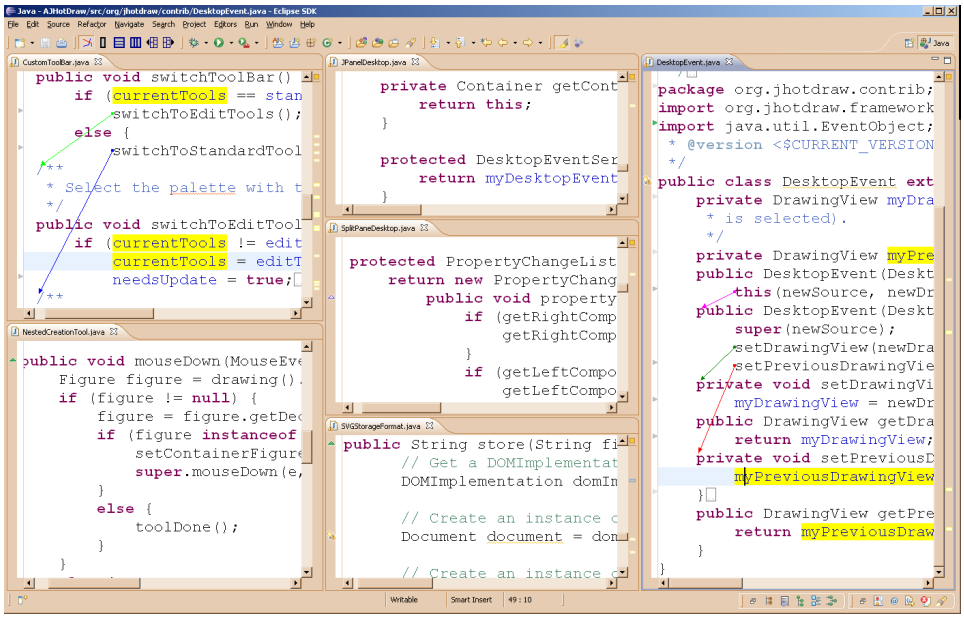


Figure 2.1: A screenshot of the prototype.

only when the user opens files or activates hidden editors. This resizing only affects the column containing the newly opened or activated file. Thus, the display is updated to reflect changes in the relative interests of files, but only at predictable times, that is, when the user has initiated an action which already causes the display to change. Furthermore, the updates consist of relatively minor adjustments to one third of the display, in contrast to the complete scene change that occurs when opening a file in a traditional IDE.

Because a newly opened file has a relatively low degree-of-interest, its degree-of-interest is not used either when it is first laid out on screen or the next time the display is updated. This allows the file's degree-of-interest time to become more meaningful as a result of the user's interactions. If the user is interested in the file, she will interact with it, causing its degree-of-interest to rise and ensuring that it remains on the screen. On the other hand, if the newly opened file turns out not to be interesting, it's degree-of-interest will become lower in relation to the files with which she does interact

and thus it will be shrunk or hidden when its display space is needed for another file.

When a file is opened, it is always placed at the bottom of a column to ensure that it is easy to find on the screen. Other editors in the column have their sizes adjusted to make room. The newly opened editor is added to the column whose visible files have the lowest total degree-of-interest, and allocated $1/n$ of the height of that column, where n is the number of visible files in the column (including the newly opened file). The editors already in that column are resized so that they each receive space in proportion to their degree-of-interest. If any editor would occupy less than 20% of the column as a result of resizing, it is instead hidden behind the editor that takes its place in the layout, so that its tab remains in the same location on the screen.

Visible editors will be shifted upwards when a file above them is selected for hiding, but otherwise, they always remain in the same relative location on the screen. When the user activates a hidden editor, if the editor whose place it would take is at least 15%[3] more interesting than the least interesting visible editor in that column, the layout is adjusted so that the least interesting editor is hidden and the editor that would have been hidden is shifted above or below the activated editor so that it remains visible. The sizes of the editors in the column are then adjusted to reflect their relative degree-of-interests.

For the most part, the user is not expected to exert direct control over the layout. However, there is a button which causes the active editor to expand to fill its column. This can be useful if the user wants to read a large part of the file in depth. There is also a previous layout button which reverses this action.

2.2.2 Progressive Elision

As an alternative to Mylyn's automatic code folding, we use a form of elision that should make better use of screen space by increasing the density of

[3]These values were arrived at through experimentation and are not claimed to be optimal.

useful information. Progressive elision computes interest at the line level rather than only at the declaration level and allows the user to adjust the amount of elision on a per-file basis and using a continuous scale, in contrast to Mylyn's global on/off control. Each line of code is assigned an interest level based on the degree-of-interest values of its parent declaration and of any elements it references. For example, the interest level of a call to a method `m1` in the body of `m2` is the sum of the degree-of-interests of `m1` and `m2`. Each file has a threshold, and lines with an interest level below the current threshold are hidden. The user can control the level of elision for a given file using a popup slider to change the threshold. Comments, closing delimiters, and blank lines are assigned progressively lower interests than code or declarations, so within a given element, they are generally the first things to be hidden.

Unlike automatic code folding, where "interesting" elements are shown in full and others have only their headers visible, progressive elision allows elements to be partially visible or completely hidden. This means that when the level of elision is set high enough, space is not wasted showing the headers of uninteresting elements, which are still readily accessible using Eclipse's popup Quick Outline view (the Quick Outline view is not filtered by Mylyn). On the other hand, if a method is not part of the task context but accesses methods or fields which are, the parts of the method that access those elements may be shown. In order to place these fragments of a method in context, the header of a method is always shown when any part of its body is visible. Finally, methods which are part of the task context may have uninteresting pieces hidden. Figure 2.2 presents an example showing partially elided methods.

When a line or multiple consecutive lines are elided, this fact is indicated with a small coloured triangle in the left vertical ruler which points at the location where text has been elided, as well as with the standard Eclipse elision indicator at the end of the line preceding the elided text, as shown in Figure 2.2. Moving the mouse over the triangle displays a popup containing the hidden text (compare Figure 2.2 and Figure 2.3, and clicking on the triangle unfolds the text in the editor. The colour of the triangle indicates

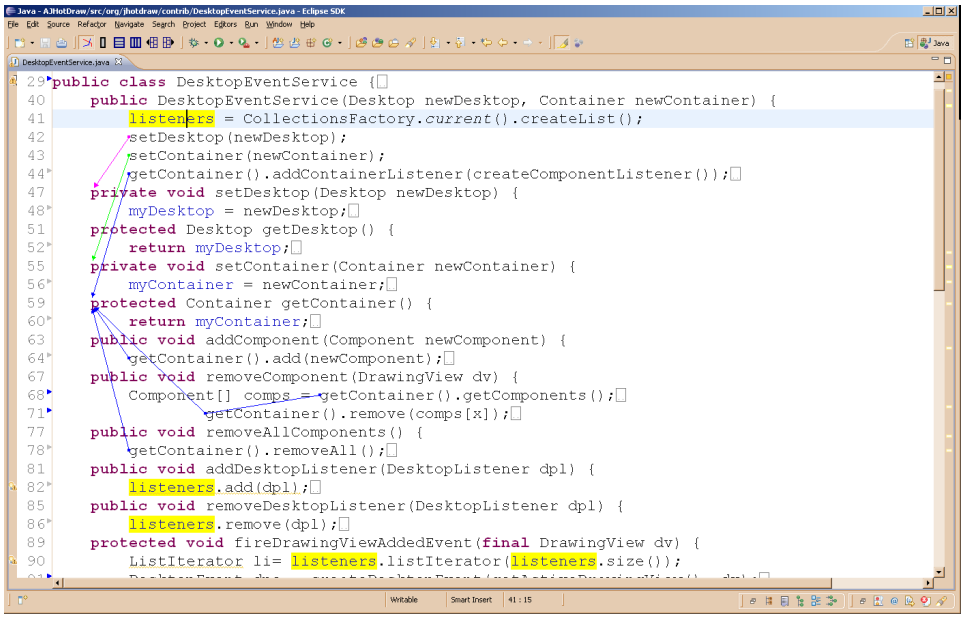


Figure 2.2: Progressive elision.

the kind of text that is hidden. A hidden block of text containing any statements or declarations is indicated with a blue triangle. If the hidden text is simply delimiters closing a code block or a statement, for instance, if it is a single line containing only a closing brace, the corresponding triangle will be dark grey. In this case, if the code is well-formatted, the fact that the block or statement has been closed will also be obvious from the indentation level of the next visible line. When a hidden block of text consists only of whitespace, it is indicated with a just visible, light grey triangle. If the hidden text is a comment, the corresponding triangle will be green (by default, Eclipse's Java editor colours comments green). Thus, the prominence of the triangle corresponds to the significance of the hidden text.

Progressive elision is intended to fully hide elements which are completely uninteresting while showing the interesting parts of relatively uninteresting elements to provide more context. This makes it possible for two interesting elements in the same file to be visible at once, even when they are textually

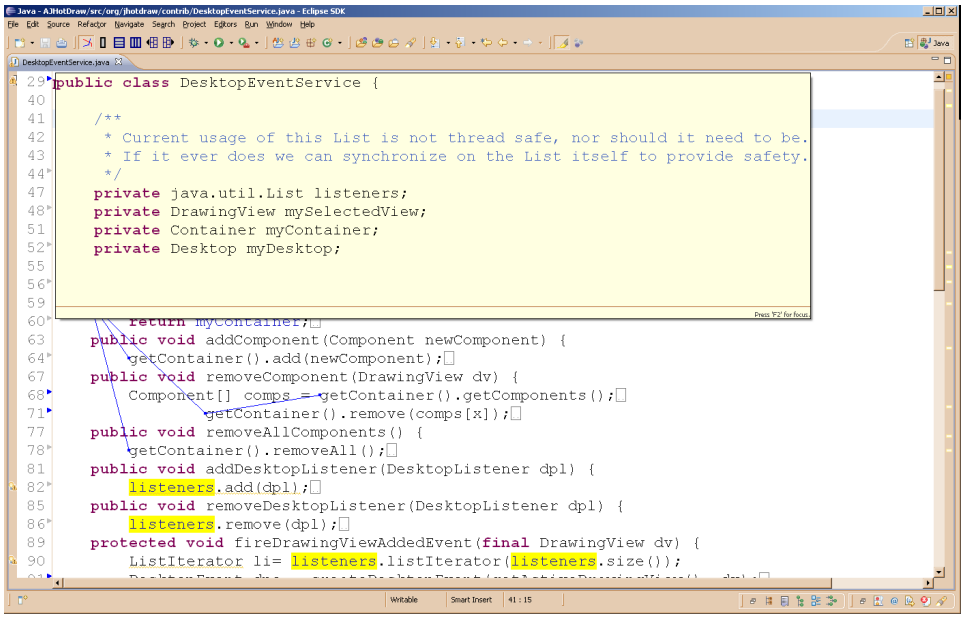


Figure 2.3: Progressive elision with a popup.

separated by numerous declarations. While it is subject to the same issue as any form of code folding, namely, that hiding information may lead the user to draw false conclusions, it is hoped that it will increase the reliability of the user's conclusions by hiding less interesting information as compared to Mylyn's automatic code folding. While displaying multiple files in a column reduces the vertical space allotted to each, progressive elision tries to use that space more effectively.

2.2.3 Graphical Annotations

By laying out editors in a two-dimensional space, rather than making them accessible only through a one-dimensional list, we are adding a dimension to the user's interaction. We are also encouraging the user to work and think in this two-dimensional space, rather than thinking of one editor at a time and viewing the one-dimensional editor list as separate from normal interaction. This creates both an opportunity and a potential problem. Presenting mul-

tiple files side by side could allow the IDE to display additional information about the relationships between files, but it also creates a more complex display – especially when one considers that each Eclipse editor contains two vertical rulers potentially showing more than a dozen different kinds of annotations – and this obviously has the potential to cause confusion. In order to help the developer make sense of this two-dimensional space, we graphically indicate important relationships between program elements both within and across files, using arrows. A toolbar button allows the user to toggle this feature on and off.

There are three relationships which are shown graphically: method calls, advice applying to a method,[4] and advice applying to an expression. For method calls, an arrow is drawn from the calling expression to the method declaration based on the static type of the receiver. For simplicity, the declarations of overriding methods which could be invoked depending on the dynamic type of the receiver are not indicated (but see Section 2.2.4 for an exception). Advice applying to a method is indicated by an arrow from the declaration of the advice to the method declaration. Advice applying to an expression, such as a method call, is indicated with an arrow from the advice declaration to the expression. Other relationships, such as references to pointcuts and fields, could be indicated, but would risk overly cluttering the display.

In keeping with AJDT's gutter annotations, arrows indicating advice application are coloured orange. For other arrows, the target of the arrow is assigned a colour from a fixed set, chosen to be easily distinguishable, using a scheme which tries to avoid using the same colour twice. Every arrow pointing to a given location is drawn in that location's colour. As a consequence, when a location is scrolled offscreen and later returns to view, it may be assigned a different colour if its previous colour has been assigned to a new location. Given the number of relationships present in the code, the fact that they change as it is edited, and the fact that the subset

[4] In this paragraph, we use phrases such as "advice applying to a method" as a shorthand for "advice with a pointcut which statically matches a method, possibly with a dynamic test."

which are visible on screen (and their relative position) changes frequently, permanently assigning colours would certainly lead to a less readily understandable display, where multiple arrows of the same colour cross. This design choice implies that the user should not try to remember relationships by their colour. Rather, the colours serve to make the arrows easy to distinguish. Whether this will work in practice needs to be validated: users may automatically associate relationships with colours and become confused when they change, or they may become accustomed to this use of colour.

Arrows are drawn only when both the source and target of the relationship being indicated are visible on the screen. A possible refinement would be to also draw arrows to offscreen targets if they have a very high degree-of-interest. If an arrow is drawn between the left and right columns, it passes "underneath" the central column so as to prevent editors in the middle of the screen from being buried under crisscrossing lines. Figure 2.4 contains an example of this. Arrows typically point to the left edge of the first line of a declaration, since this is the lexical start of the declaration and is near the conceptual location that the declaration assumes control upon being invoked. However, when the arrow is coming from a location more than 25 pixels to the right of the last character in the declaration's header, it will point to a location just to the right of this character (depending on how the code is formatted, this is usually the location of the opening brace which starts the body of the declaration). This not only avoids drawing unnecessarily long arrows but also gives the arrow the appearance of pointing at the declaration rather than past it, making it easier for the user to follow.

Given locations `X`, `Y`, and `Z` on the screen, when drawing an arrow from `X` to `Z`, if there is already an arrow from `Y` to `Z`, the new arrow will instead point to `Y`, the tail of the first arrow, whenever either the resulting line segment `XY` is less than half the length of `XZ` or the angle between `XY` and `YZ` is less than 10°. This reduces clutter and in the second case also avoids drawing nearly overlapping arrows. A small square is drawn wherever the line segments join to indicate that that location also contains a reference to the target of the arrow.

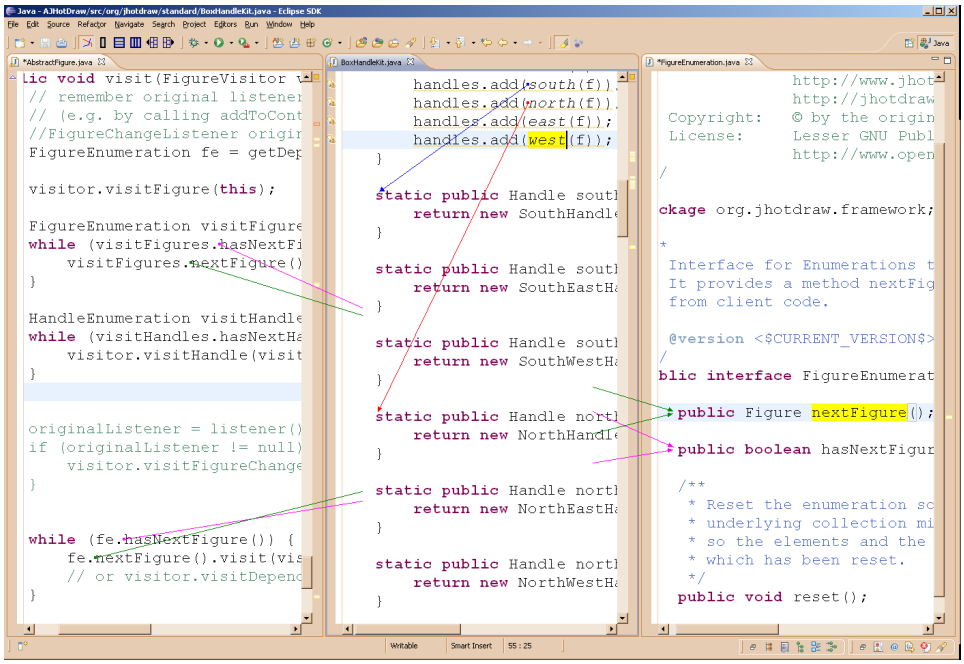


Figure 2.4: Arrows passing “underneath” the central column.

2.2.4 Displaying Arbitrary Concerns

Section 2.2.1 described how the history of the user's interactions determines which files are visible. The set of visible files essentially functions as a view of the Mylyn task context, providing access to a set of program elements which are implicitly related by their relevance to the current task, as determined by the degree-of-interest function. However, the multiple file interaction paradigm also lends itself naturally to displaying concerns specified in any other manner. As a simple example, the user can select any subset of the results from an Eclipse search and display them simultaneously. She can then compare the results without having to manually iterate back and forth through them. In some cases, such as when searching for the readers and writers of a field, the results of a search might constitute a well-defined concern, in which case the user can easily display the components of that concern together.

There is a tighter integration with the Eclipse call hierarchy view. This view can display either a caller or callee hierarchy as a tree. The prototype adds an *Open Children* command which displays the children of the currently selected node in the tree. This command hides the currently open editors and opens editors containing the selected node and its immediate children, with the selected node in the central column and its children on either side. The Eclipse call*er* hierarchy on a method `m1` includes calls to methods which `m1` overrides, that is, calls which may or may not actually invoke `m1`, depending on the dynamic type of the receiver. Normally, we do not graphically indicate such calls, but when *Open Children*, is invoked, all relationships shown in the call hierarchy are indicated with arrows, as shown in Figure 2.5. Thus, this command provides an alternate view of the call hierarchy which shows the source code for one level of the hierarchy at a time, with the call relationships graphically displayed. Sometimes a node has too many children to fit simultaneously on the screen. In this case, we partition the children into groups and augment the call hierarchy view with next and previous buttons which allow the user to iterate through the groups (the previous layout button in the main Eclipse toolbar allows the user to

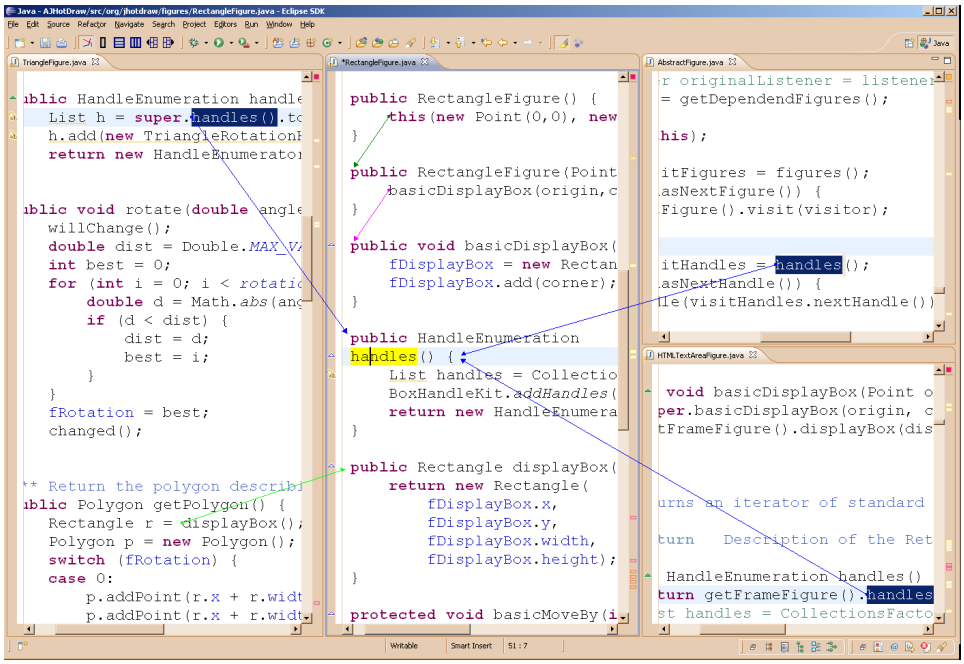


Figure 2.5: A caller hierarchy on `RectangleFigure.handles()` (center), with arrows indicating potential calls. In particular, notice that the call to the abstract `handles()` method of `AbstractFigure` (top right) is shown as calling `RectangleFigure.handles()`.

return the display to the state it was in prior to invoking the call hierarchy). Viewing a level of the caller hierarchy in this way allows the user to make comparisons of the callers of a method in terms of what parameters they pass and how they use its results, without having to navigate back and forth between them. Displaying the children of a method in the callee hierarchy could allow the user to better understand how they interact to make up the functionality of that method.

To facilitate the understanding of crosscutting concerns, we have implemented a command that shows how advice crosscuts the system. When invoked on an advice, the advice is displayed alongside each shadow which is a static match for its pointcut. This makes it easy to see how the advice interacts with the code it advises. A complimentary command to display

all the advice which applies to or within a method could also be useful, although this has not been implemented in the prototype.

Finally, it should be fairly straightforward to integrate this approach with a concern description tool such as FEAT [20] and with JQuery [12]. This would allow the user to simultaneously view the source code for the different parts of a concern as captured by these tools.

Chapter 3

Implementation

The prototype is implemented as a plugin for Eclipse version 3.3.1.1. It works with any Eclipse editor, although progressive elision does depend on Eclipse's internal `JavaEditor` class,[5] using it to retrieve its associated `ProjectionViewer`, and thus is only available for subclasses of `JavaEditor`, including the AJDT editor for AspectJ. It was necessary to make some minor changes to two Eclipse classes; these are described in the following sections. The plugin also relies on some internal packages which are not part of the public API of Eclipse and might therefore change in future versions.

3.1 Multiple Editors

Eclipse already supports displaying multiple editors, but only when the user drags them into position with the Mouse. There is no API that allows them to be programmatically arranged or resized. To get around this limitation, we simulated drag and drop events using an internal Eclipse class.[6] Whenever a file is opened, Eclipse arbitrarily chooses another open file and places the new file on top of it. Then, we simulate drag events to move it into the desired position and resize it appropriately. This creates a slight delay, barely noticeable on a fast computer, when opening a file. However, when opening multiple files at once, for example, when selecting several results of a search, or when activating a Mylyn task context, the delay can last several seconds, and the movement of editors on the screen is visible. This is because using drag and drop means that the display is updated after each editor is positioned. The problem could be eliminated if Eclipse

[5] `org.eclipse.jdt.internal.ui.javaeditor.JavaEditor`
[6] `org.eclipse.ui.internal.dnd.DragUtil`

added a public API for manipulating the editor area that allowed multiple operations to be performed before updating the display.

We use our own representation of the Eclipse editor area, dividing it into columns which are in turn divided into editor stacks, each of which contains an ordered list of editors and a pointer to the currently active editor. This is much more convenient than Eclipse's internal representation which is essentially a binary tree that splits the editor area into successively smaller regions. This allows us to save previous editor layouts and to separate code that manipulates our representation from the code that converts these manipulations into the drag events needed to manipulate Eclipse's representation. We do not update our representation if the user manually drags an editor to a new location, because Eclipse provides no notification of such actions (although it would probably be possible, though non-trivial, to determine the current layout and update our representation before attempting to manipulate it). Therefore, users should not manually rearrange editors.

We modified the `JavaEditor` class to provide access to the currently selected element so that we can ensure it remains visible when the editor is resized. This is a minor, non-critical feature.

3.2 Progressive Elision

Progressive elision is implemented using Eclipse's support for projection, also used to implement code folding. For each open `JavaEditor`, we maintain a model of its lines which maps each line to the declaration it belongs to (if any) as well as to its lexical position within the file, keeps a list of the references to other elements that appear on each line and computes the interest level of each line. When updating the level of elision, the set of text regions that are to be elided is computed and compared with the set of regions that are already elided. Any overlapping or adjacent regions are combined into a single region and, for each resulting region, a subclass of `ProjectionAnnotation` is added to the `ProjectionAnnotationModel` that Eclipse associates with each `JavaEditor`. The process for revealing text is

similar: the set of regions to be expanded is computed and compared with the set of elided regions, resulting in elided regions having their annotations removed or replaced with one or more annotations which span fewer lines. The annotation controls the drawing of coloured triangles in the margin.

The elision level is changed using a slider whose position is interpreted internally as a number between 0 and 1. In the initial implementation, this number was mapped to an interest level by equating the current least and most interesting lines with 0 and 1, respectively. The result was then taken as a threshold, and any less interesting lines were hidden. However, the interest level space is typically both sparsely populated and heavily clustered. The result was that, sometimes, moving the slider a short distance caused dramatic changes, while other times, moving it a long distance appeared to have no effect. We refined the implementation by sorting the lines by interest level (using line numbers to break ties) and interpreting the slider position as the percentage of lines that should be hidden. The result is a much smoother control.

In order to allow multiple annotations to be collapsed or expanded without updating the display after each one, we had to add the appropriate methods to `ProjectionAnnotationModel`.[7] We also had to give it a public `fireModelChanged()` method. Without these changes, smoothly adjusting the level of elision would not have been possible.

3.3 Graphical Annotations

Graphical annotations are implemented by drawing on an SWT canvas which we overlay with the editor area. Because SWT does not support transparent canvases, the canvas is configured to use the editor area as its background and must be repainted every time the editor area changes, for example, whenever the user types, scrolls, brings up or dismisses a tooltip, or selects text. This can cause graphical annotations to flicker, and it sometimes causes a small delay in changes to the editor area becoming visible. This is because many of the changes which must cause the canvas to be repainted are not

[7] `org.eclipse.jface.text.source.projection.ProjectionAnnotationModel`

events that Eclipse allows one to listen for, but rather happen shortly after mouse or keyboard events. For example, when the mark occurrences feature is enabled, the set of marked occurrences is updated a short time after the user selects an identifier. In order for the canvas to reflect these changes, we use a timer to update it after a small delay. Whenever an editor opens, we must attach a listener to each of its scroll bars so that we can update the canvas when scrolling. This is because mouse events on scroll bars are handled by SWT at a high level and not passed down to the Eclipse UI toolkit, JFace.

Chapter 4

Case Studies

Ultimately, the measure of the utility of a system such as this is the extent to which it improves programmer productivity. This appears to depend on the answers to two questions: (1) Does the system reduce the time spent on navigation? and (2) What effect does it have on the time spent on activities other than navigation?

The primary quantifiable benefit of the system is expected to be a reduction in the time spent navigating. Therefore, if it does not in fact reduce navigation time, the second question becomes moot in this context. For that reason, answering the first question is the first step in a practical evaluation of the system; if it is answered in the affirmative, future work should explore the second question through user studies. Section 7.4 describes some of the questions that such studies could investigate.

This chapter presents two case studies involving programming tasks taken from previous user studies into programmers' behaviour when investigating unfamiliar code [17, 19]. The first case study consists of maintenance tasks on a simple drawing application; the second consists of a more involved change to a popular text editor. These case studies are a first step toward answering question (1) and also serve to demonstrate our approach in action and show that it can capture much of the code relevant to a task in a single view.

4.1 Methodology

For each programming task, we analyzed the knowledge required to perform the task, including structural relationships (e.g. which methods make calls to a particular method, which class in a hierarchy implements a method),

concern-to-code mappings (i.e. which program elements participate in the implementation of a concern), design rules (e.g. operations of a certain sort should be handled by a particular object), method behaviours (e.g. that a method uses certain state in computing its result, that a method updates certain data structures), and the locations of relevant code (e.g. code that can serve as a template for a part of the solution, helpful comments).

This analysis resulted in a set of facts a developer ought to know in order to perform the task and a plausible sequence of code locations to be read (to discover these facts) or edited (to complete the task). Note that there may be room for debate about exactly how much information is needed to properly complete a task. Because most systems are too large to permit a developer to read every line of code that might be relevant, there is probably always a certain amount of assumption involved in determining how a system works, and how best to implement an enhancement or bug fix. So, the amount of code a developer actually needs to read in some sense depends on a combination of luck and the keenness of her intuition. Also, there may be many different reasonable exploration paths that would uncover the needed information. We tried to make realistic choices.

By counting the number of navigations a developer would have to perform to visit this sequence of locations, both when using our prototype and when using Mylyn and Eclipse alone, we get a rough measure of the potential impact of the system on navigation time. Because the analysis is based only on the information which a developer should know in order to successfully complete the task, our navigation counts approximate a best case; real developers will likely take wrong turns and discover facts that turn out to be irrelevant to the task. We expect that such unnecessary navigations may incur a similar benefit when using our prototype and we claim that our prototype will not significantly penalize developers for them by filling the screen with irrelevant files. This claim is based on the success of Mylyn, which would be of little benefit if task contexts were prone to becoming heavily polluted with extraneous elements. Our analysis makes the generous assumption that the developer remembers all the facts they have discovered; in practice we expect that real developers will need to revisit previous lo-

cations more than we have modeled, so our analysis may underestimate the savings that result from our approach. Also, because our approach may help developers find previously visited locations of interest more easily, real developers may perform extra unnecessary navigations when not using our prototype. Our analysis does not account for these possibilities.

Our assessment of the potential impact of the system on navigation time uses navigation counts for a developer who is both lucky and intuitive as a proxy for the time spent navigating in the best case. We think that this allows us to make a reasonable estimate of the potential effect of the system on navigation time in the expected case, ignoring the cognitive impact of the system which should be measured through user studies. Given that developers spend a significant fraction of their time performing navigations [17], improvements in navigation time should translate into higher productivity.

The following sections walk through each task, describing the sequence of actions taken by an imaginary developer without reference to whether she is using the prototype or not. A series of screenshots illustrate how the prototype evolves to display the whole task context. Section 4.4 presents the results in terms of the number of navigations performed when using and not using our prototype.

In order to allow the screenshots to fit legibly on the page, it was necessary to increase the font size and greatly reduce the width of the Eclipse window. As a result, the screenshots show editors that are not nearly wide enough for practical use. We also set the prototype to use its two column mode. However, even a 24" widescreen display, which is much smaller than the largest displays currently on the market, can accommodate 3 columns of source files, each 80 characters wide (in 10 pt. font). When looking at the screenshots, the reader should keep in mind that in real use, there would be three columns of editors, each probably at least twice as wide (in characters) as the editors shown here and two to three times as high (in lines).

4.2 Case Study 1: Paint

As a starting point, we analyzed the user study performed in [17], in which programmers were given 70 minutes[8] to complete five maintenance tasks on a small drawing application called Paint, implemented in Java using Swing.[9] The number of actions performed by each programmer on each task was recorded, where examples of actions include reading code, editing code, and performing a navigation. An analysis of their behaviour concluded, among other things, that they could benefit from a workspace that allowed them to keep relevant information on the screen. Three of the tasks involve fixing real (i.e. not artificial) bugs, and the other two are enhancement tasks. While the codebase used was very small, it was described as "reasonably complex" and the study had the advantage that, before our implementation was completed, we were able to determine relatively quickly whether the system could produce the measurable improvement that we expected.

Because the tasks used in this study are small and the sets of facts a developer must know to implement them have some overlap, we treat them independently, imagining that each task is being performed by a different developer. One of the debugging tasks, Yellow, and one of the enhancement tasks, Thickness, require so little information to complete that they do not provide an opportunity for our prototype to have much impact in this kind of study, and thus we do not discuss them. However, it is quite possible that user studies would show that our approach is beneficial even for these smallest tasks because real users may perform much more navigation than is strictly necessary to complete them.

The Paint application is implemented in one package whose classes are listed in Figure 4.1. The application's interface is shown in Figure 4.2.

[8] Programmers were interrupted with math problems every 2.5 to 3.5 minutes, and spent 15 minutes on average handling these interruptions.

[9] The source code of Paint is available at http://www.cs.cmu.edu/~marmalade/studies.html

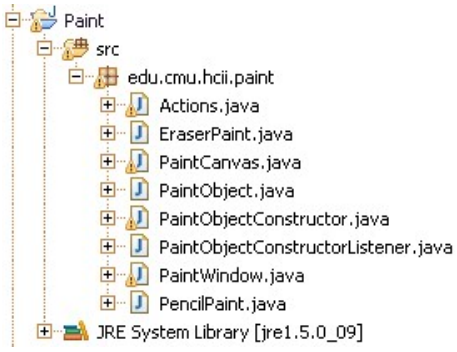


Figure 4.1: The classes of the Paint application.

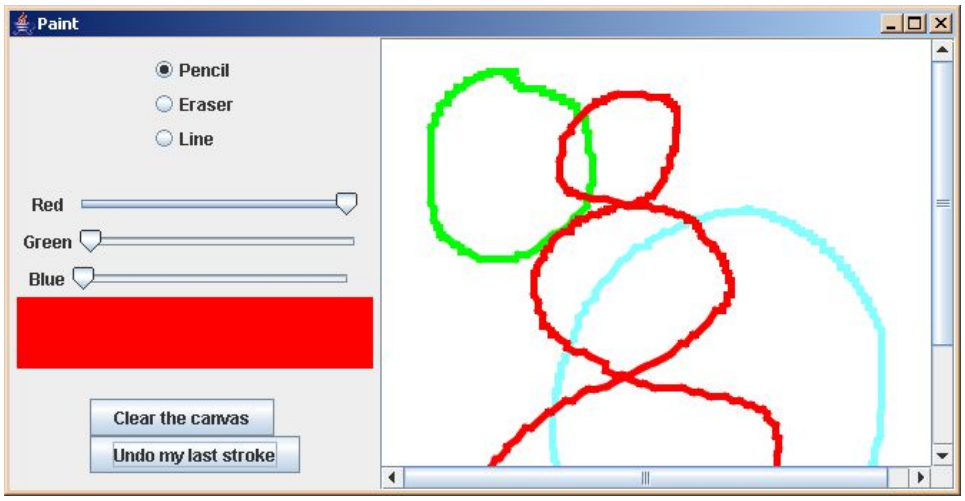


Figure 4.2: The Paint application.

4.2.1 Task 1: Scroll

This task involves fixing a bug in which scroll bars do not always appear after painting outside the canvas, and when they do, it becomes distorted. In [17], on average, study participants spent 17 minutes on the task and performed 64.5 actions. However, only one of ten subjects completed the task successfully.

In search of the code that controls when painting happens, our imaginary developer begins by opening the `Actions` class and seeing that it declares the public fields `clearAction`, `undoAction`, `pencilAction`, and `eraserAction` (Figure 4.3). Guessing that `pencilAction` is responsible for painting, she searches for references to it and arrives at the constructor of the `PaintWindow` class. Seeing that `pencilAction` is attached to a `JRadioButton` (4.4), she realizes that `pencilAction` selects the pencil tool but does not actually perform any drawing. Returning to the `Actions` class, she sees that `pencilAction` selects the pencil with the call

```
paintWindow.setPaintObjectClass(PencilPaint.class)
```

and follows the cross-reference back to `PaintWindow` (4.5). She then follows another method call which leads her to the `setClass()` method of the `PaintObjectConstructor` class and she notices that this class contains methods that handle mouse events (4.6). Looking for the code that is invoked when drawing on the canvas, she looks at the `mouseReleased()` method and then follows a method call to the `constructionComplete()` method of `PaintObjectConstructorListener` (4.7). Seeing that this is an abstract method, she uses the Quick Type Hierarchy to navigate to its lone implementor, taking her back to the `PaintWindow` class (4.8). From there, she follows the call to the `addPaintObject()` method in `PaintCanvas` and sees that this in turn calls `repaint()` on itself (4.9). Scrolling up to the class declaration, she sees that `PaintCanvas` extends from the Swing class `JPanel`. Returning to `PaintWindow.constructionComplete()`, she uses the Mark Occurrences command on the `canvas` field and scrolls up to find that the `PaintWindow` constructor embeds it in a `JScrollPane`(4.10). Based on her knowledge of Swing, she realizes that the scroll bars will not

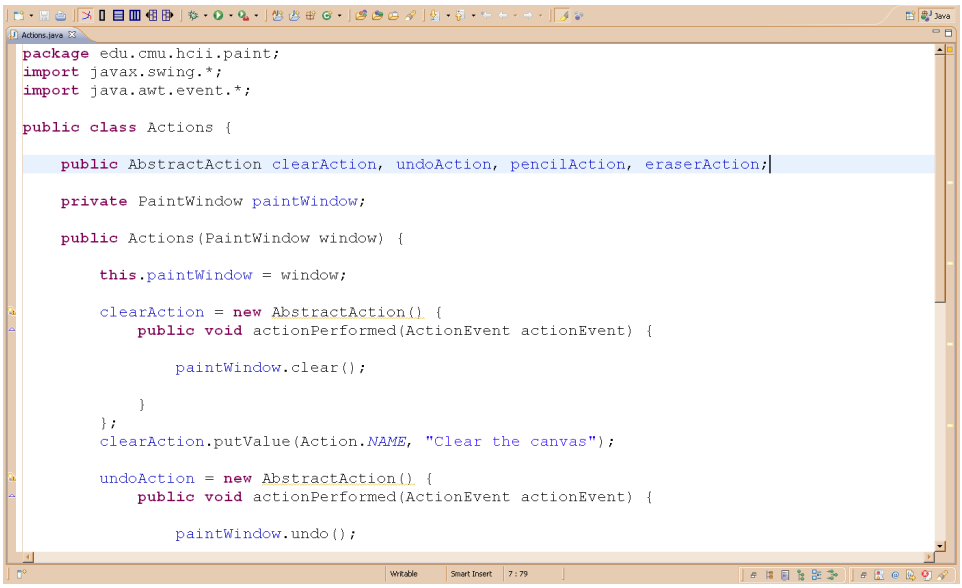


Figure 4.3: The Scroll task after opening the Actions class.

appear unless the preferred size of the canvas is updated. Returning to `PaintCanvas.addPaintObject()`, she adds the necessary code (4.11). Running the program reveals that the scroll bars now appear, but that the canvas is not drawn correctly when it is scrolled. Selecting the `paintComponent()` method of `PaintCanvas` from the Quick Outline, she notices an extra call to `clipBounds.getX()` which should be `clipBounds.getY()` (4.12). Correcting this error completes the task.

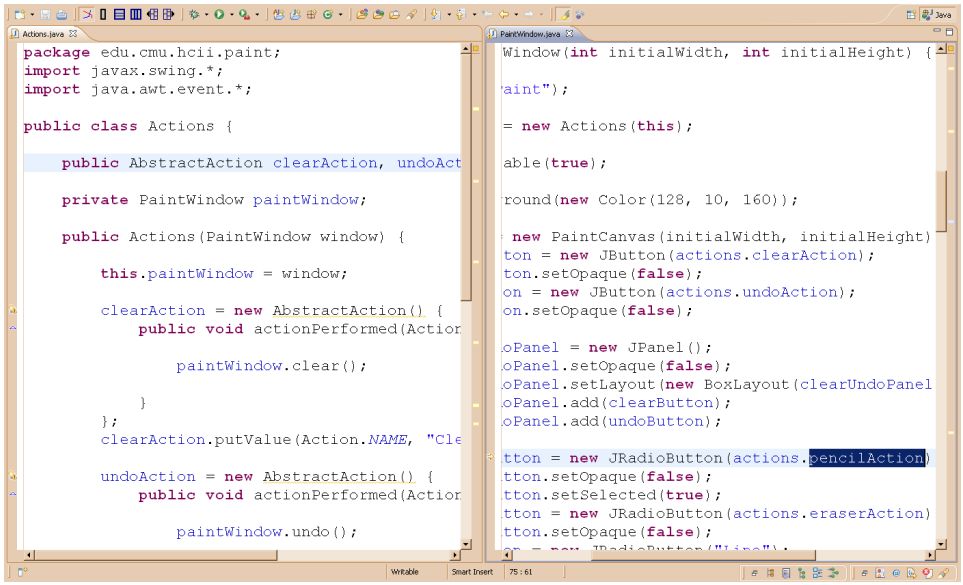


Figure 4.4: The Scroll task after navigating to the PaintWindow constructor.

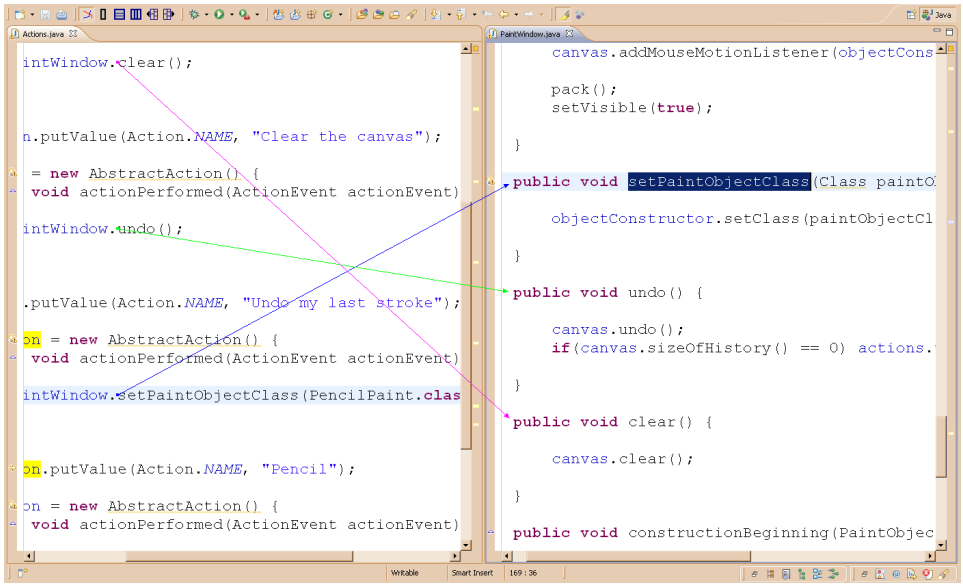


Figure 4.5: The Scroll task after navigating to setPaintObjectClass() in PaintWindow.

Figure 4.6: The Scroll task after navigating to setClass() in PaintObject-Constructor.

Figure 4.7: The Scroll task after navigating to constructionComplete() in PaintObiectConstructorListener.

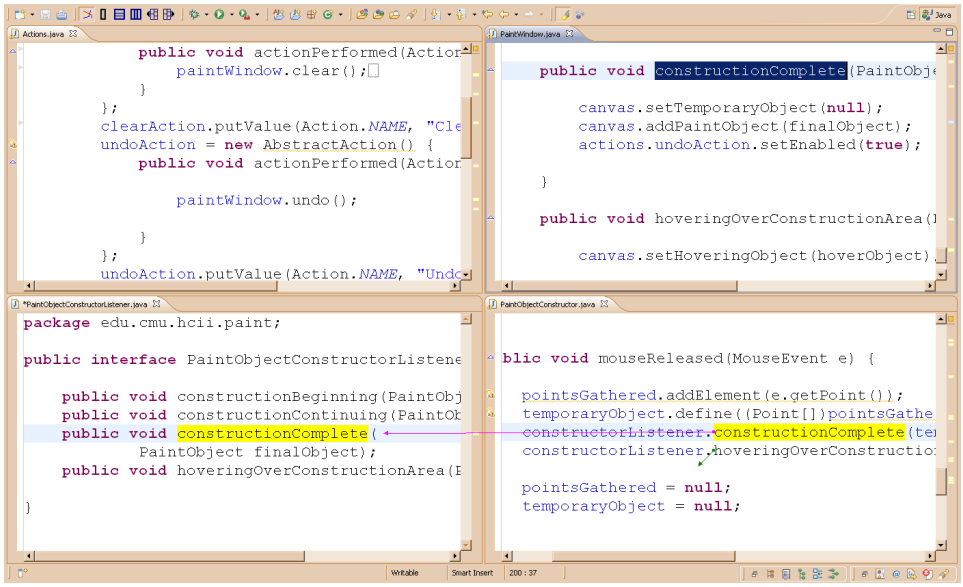


Figure 4.8: The Scroll task after navigating to constructionComplete() in PaintWindow.

Figure 4.9: The Scroll task after navigating to addPaintObject() in PaintCanvas.

Figure 4.10: The Scroll task after scrolling to the PaintWindow constructor.

Figure 4.11: The Scroll task after editing addPaintObject() in PaintCanvas.

Figure 4.12: The Scroll task after finding a bug in the paintComponent() method of PaintCanvas.

4.2.2 Task 2: Undo

In this task, developers were asked to fix a bug in which the "Undo my last stroke" button does not always work. The button is supposed to undo the last painting operation or clearing of the canvas. In [17], on average, study participants spent 6 minutes on the task and performed 30 actions, with nine of ten subjects completing the task successfully.

Our developer begins by opening the `Actions` class and marking occurrences of the `undoAction` field, discovering that it invokes the `undo()` method of the `PaintWindow` class (Figure 4.13). She navigates to this method (4.14) and, from there, she follows a call to `PaintCanvas.undo()` and observes that nearby methods of the `PaintCanvas` class such as `clear()` and `addPaintObject()` end with a call to `repaint()`, whereas `undo()` does not (4.15). She then adds a call to `repaint()` to the `undo()` method.

While [17] considers this to be a complete fix to the bug, our developer notices that the undo button is still sometimes disabled when it should not be. Returning to `PaintWindow.undo()`, she sees that this method disables the undo button whenever the method `sizeOfHistory()` in the `PaintCanvas`class returns zero. Judging this logic to be correct, she returns to `PaintCanvas.undo()`, which removes the last element from the `history` field. By looking at this method and the `addPaintObject()` method, she learns that the current set of `PaintObject`s are stored in a `Vector`, and that the history is a `Vector` of `Vector`s, where each `Vector` stores the complete state of the canvas at some point in time. By using the debugger, she realizes that the call

```
history.removeElement(history.lastElement())
```

actually removes the first element which is `equal` to the last element, which is not necessarily the last element in the `Vector` (this occurs when the "Clear the canvas" button is pressed, causing the undo history to contain multiple `equal` blank canvases). She fixes this bug by replacing the above call with `history.removeElementAt(history.size() - 1)` (4.16).

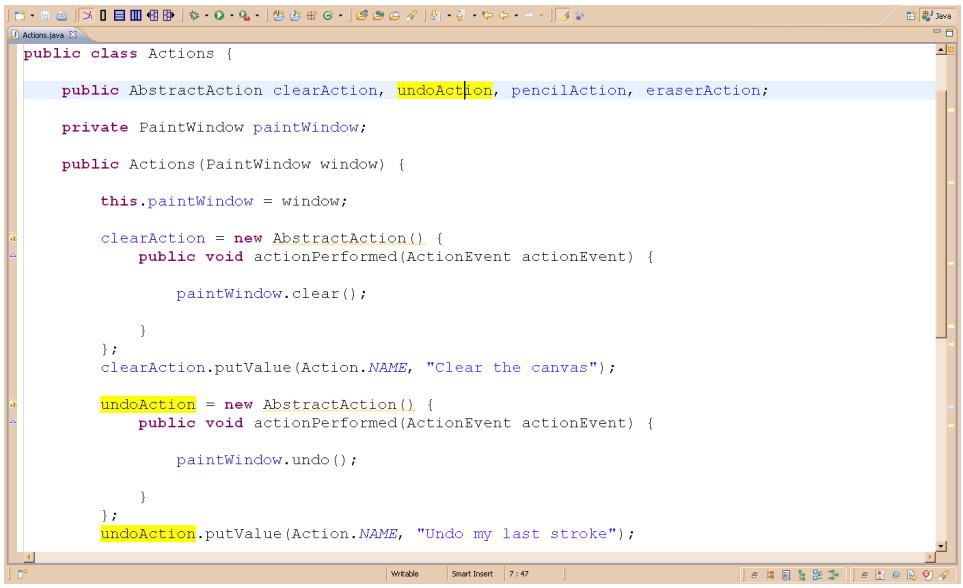


Figure 4.13: The Undo task after opening the Actions class.

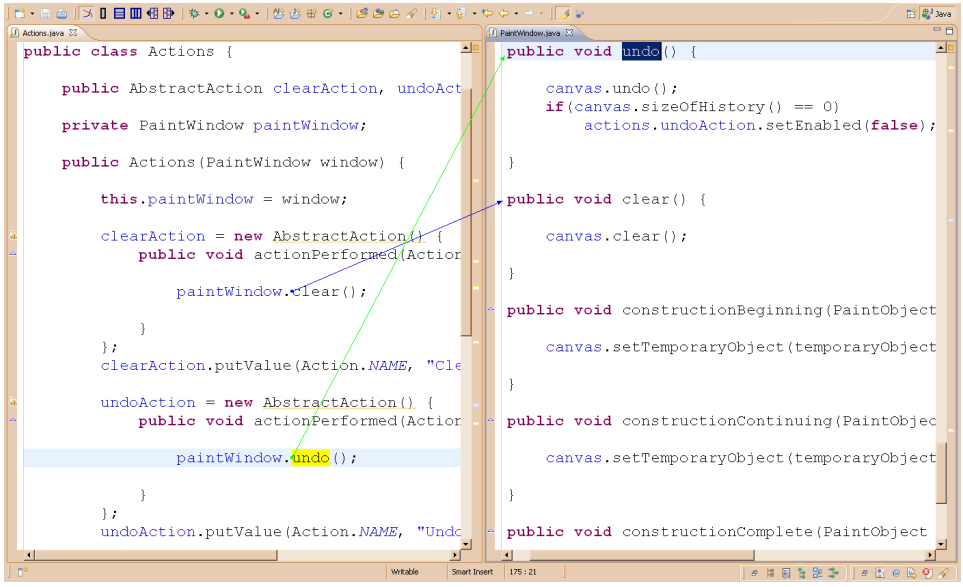


Figure 4.14: The Undo task after navigating to undo() in PaintWindow.

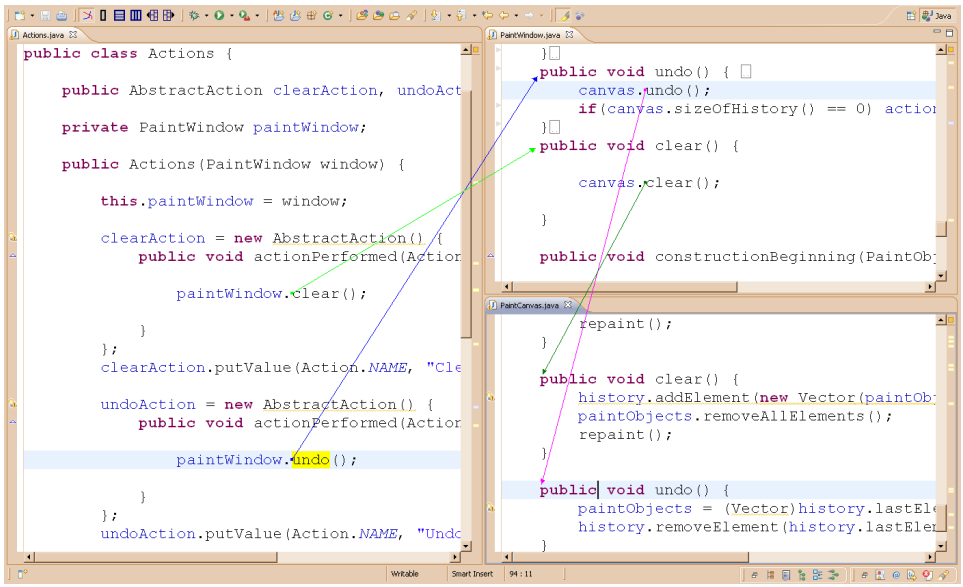


Figure 4.15: The Undo task after navigating to undo() in PaintCanvas.

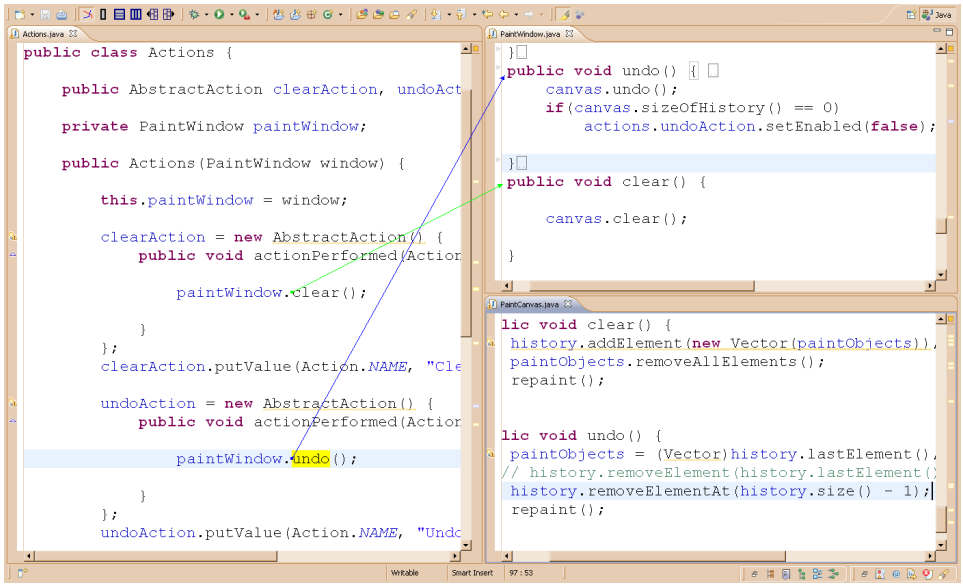


Figure 4.16: The Undo task after fixing the bugs in the undo() method of PaintCanvas.

4.2.3 Task 3: Line

In this task, developers were asked to add a tool for drawing straight lines which could be activated with the provided radio button. A line should be visible while the user is drawing it. In [17], on average, study participants spent 22 minutes on this task and performed 67 actions, with six of eight subjects who attempted the task completing it successfully.

The developer begins by opening the `PaintWindow` class and seeing that it declares the fields `pencilButton`, `eraserButton`, and `lineButton` (Figure 4.17). Marking occurrences of `lineButton` takes her to the `PaintWindow` constructor where she sees that `pencilButton` and `eraserButton` are initialized with `AbstractActions` defined in the `Actions` class, but lineButton is initialized only with the string "Line" (4.18). Following the cross-reference to the `pencilAction` field of `Actions`, she sees that the effect of the "Pencil" button is to pass the `PencilPaint` class to the `setPaintObjectClass()` method of `PaintWindow` (4.19). She then follows the cross-reference to the `PencilPaint` class which reveals that it draws lines between the points stored in a package visible array, `points` (4.20), and that it extends the `PaintObject` class. A Quick Type Hierarchy on `PaintObject`shows that it is an abstract class, that `PencilPaint` is its only direct descendant, and that `EraserPaint` extends from `PencilPaint`. The developer realizes she will need to create a `LinePaint` class. Following the reference to `PaintObject`, she sees that it implements only `setColor()` and `setThickness()` (4.21), so she decides that it would be best for `LinePaint` to extend from `PencilPaint` (this is the solution advocated in [17]).

The developer creates a `LinePaint` class (4.22), returns to `PencilPaint` to copy the its `paint()` method (4.23), and then pastes it into `LinePaint`, modifying it to draw a straight line between the first and last point in the array (4.24). Returning to the `Actions` class, she creates a `lineAction` field which activates the `LinePaint` class (4.25), then returns to the `PaintWindow` constructor to intialize the `lineButton` field with this action (4.26).

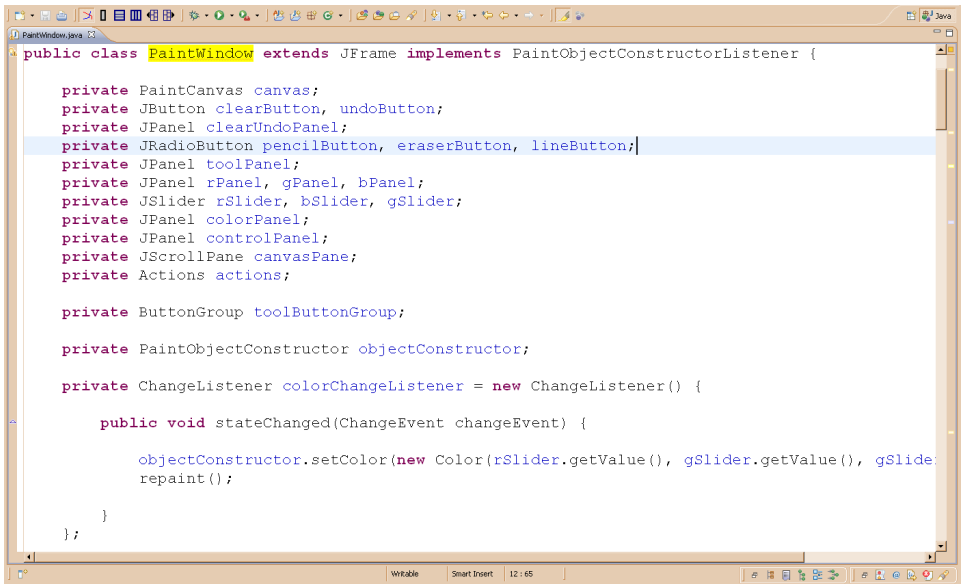


Figure 4.17: The Line task after opening the PaintWindow class.

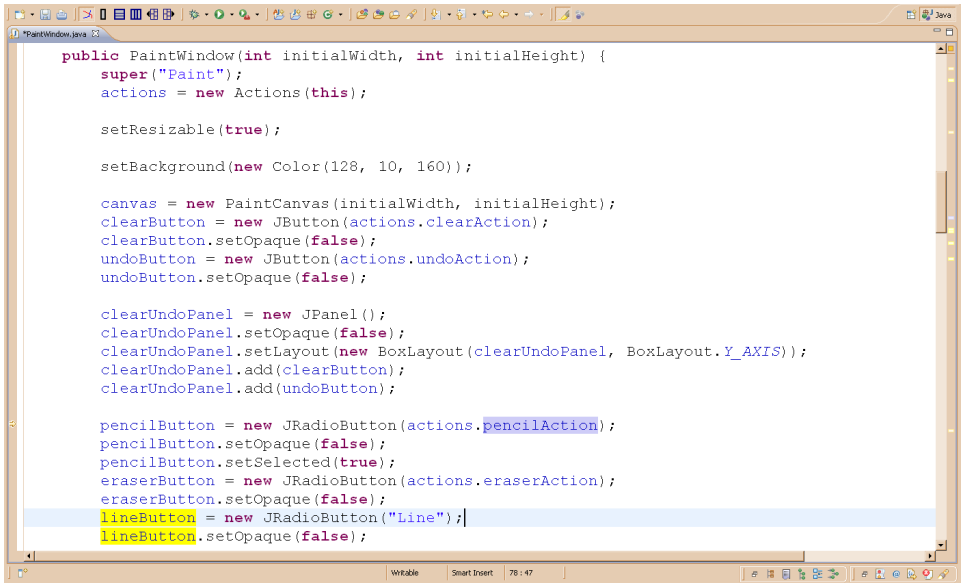


Figure 4.18: The Line task after scrolling to the PaintWindow constructor.

Figure 4.19: The Line task after navigating to the pencilAction field of Actions.

Figure 4.20: The Line task after navigating to PencilPaint.

Figure 4.21: The Line task after navigating to PaintObject.

Figure 4.22: The Line task after creating a LinePaint class.

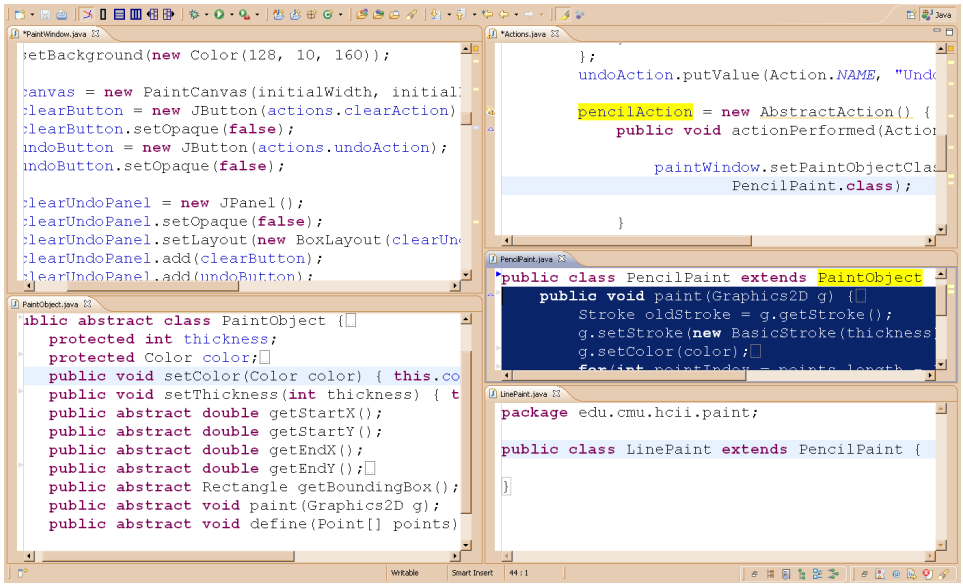


Figure 4.23: The Line task after copying the paint() method of PencilPaint.

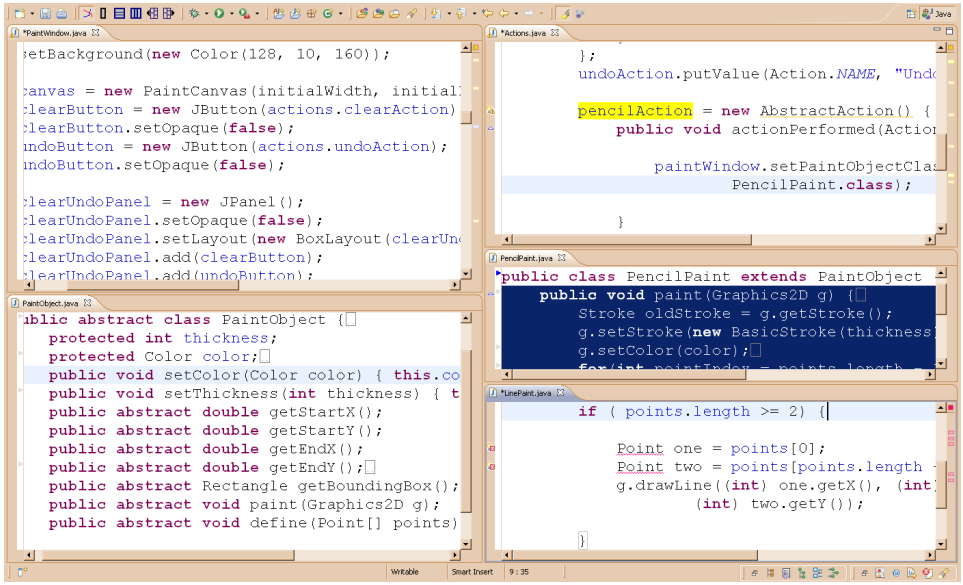


Figure 4.24: The Line task after creating a paint() method in LinePaint.

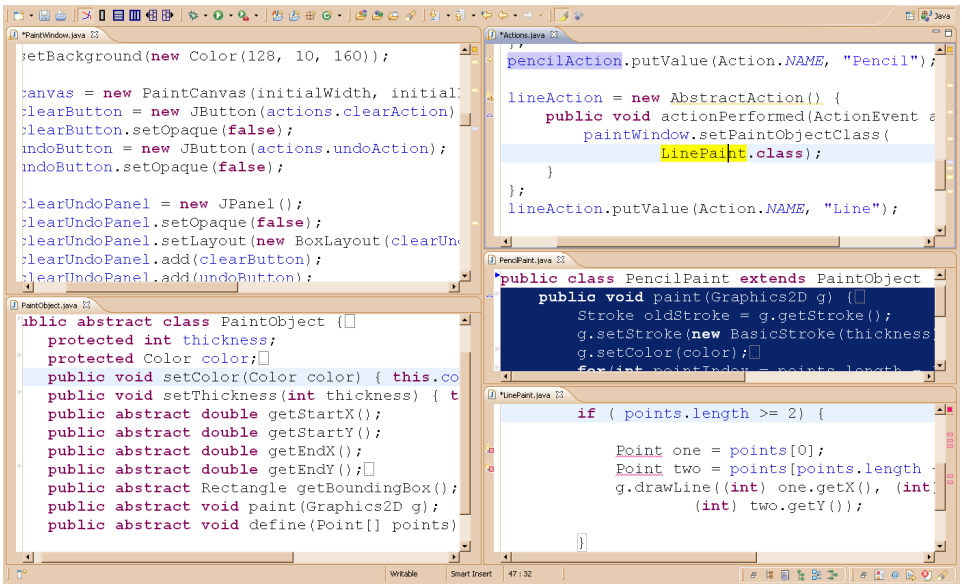


Figure 4.25: The Line task after adding a lineAction field to Actions.

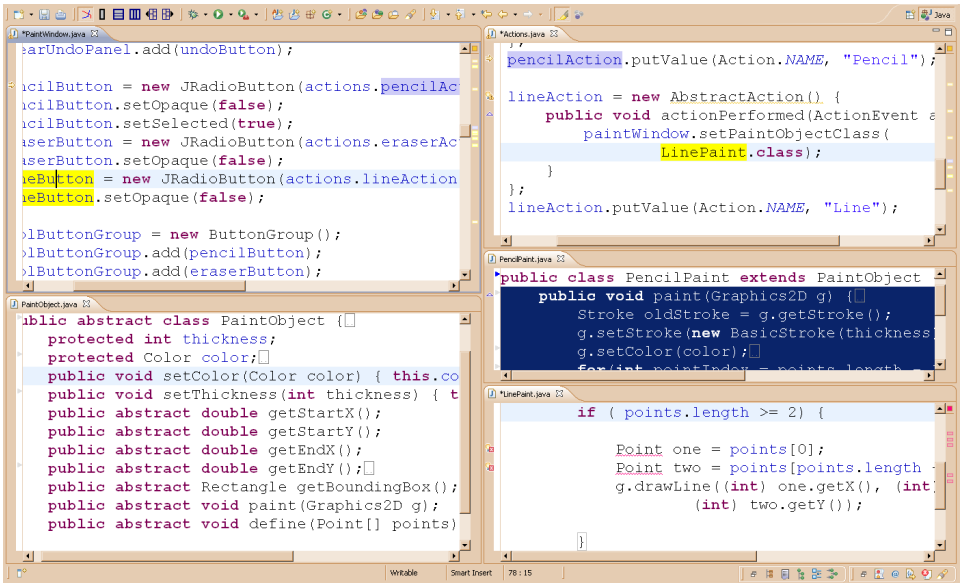


Figure 4.26: The Line task after modifying the PaintWindow constructor.

4.3 Case Study 2: jEdit

The study conducted in [19] consisted of a single, larger task: programmers were asked to modify the jEdit text editor[10] to allow disabling of the autosave feature, which saves backup copies of all open files ("buffers") at a user-specified interval. The version of jEdit in question (4.1-pre6) "consists of 64,994 noncomment, nonblank lines of source code, distributed over 301 classes in 20 packages." Successfully completing this task requires a significant level of understanding of the code and entails modifying the GUI and persisting the enabled/disabled state of the autosave feature as well as discovering how to disable it. Subjects were given one hour to investigate the program and a further two hours to complete the task, and were provided with detailed instructions on how to exercise the autosave feature, excerpts from the jEdit user manual, a detailed change request, and eight test cases that their solution had to pass.[11]

The quality of solutions was evaluated in terms of both correctness and conformance to the design of jEdit. Two subjects were deemed "highly successful" and completed the task in just over an hour. The other three took nearly the full two hours; one produced a buggy, "low-quality" solution while the other two failed on four of five subtasks.

4.3.1 Walkthrough

Subjects were also given the following "expert knowledge:" "a checkbox should be added to `org.gjt.jedit.options.LoadSaveOptionPane` to enable/disable the autosave. The autosave timer is in `org.gjt.sp.jedit.Autosave`." Thus, our imaginary developer begins by opening the class `LoadSaveOptionPane` and discovers that GUI components should be set up in the `_init()` method (Figure 4.27). Looking at the `_save()` method reveals that it reads the state of the GUI components and passes it to the `jEdit` class using its `set*Property()` methods (4.28). Searching for "checkbox"

[10] http://www.jedit.org

[11] The complete experimental package is available at http://www.cs.mcgill.ca/~martin/tse1/

within `LoadSaveOptionPane` finds code which sets up a `JCheckBox` and adds it to the GUI (4.29). Once the developer is ready to modify the GUI, this code will serve as a template.

Opening the `Autosave` class, the developer discovers that the method `setInterval()` will stop the autosave timer when the interval is set to zero (4.30). She also observes that the timer invokes the `actionPerformed()` method of `Autosave` which retrieves the list of open files and iterates through them, calling the `autosave()` method of the `Buffer` class on each one (4.31). The Call Hierarchy view reveals that the `setInterval()` method is called only by `jEdit.propertiesChanged()`. Opening `propertiesChanged()` reveals that the interval is retrieved using `getIntegerProperty("autosave")` (4.32), and the developer concludes that she will need to modify this method to call `setInterval(0)` instead if autosave is disabled. Returning to the `LoadSaveOptionPane._save()` method confirms that the `"autosave"` property is set to the value of the `autosave` text field.

One of the requirements is that the state of the autosave feature persist. So, the user returns to the `jEdit` class and looks in the Quick Outline for a save method. She selects the `saveSettings()` method and determines that it will save all of the properties to disk, including the new property she will need to add to encode the enabled/disabled state of autosave (4.33).

Returning to `Autosave.actionPerformed()`, and opening a call hierarchy on the call to `Buffer.autosave()` shows that it is only called by `actionPerformed()`, that is, the autosave feature is only activated by the timer, so disabling the timer will be enough to prevent all autosaving.

Because the change request specifies that autosave backup files must be deleted as soon as autosave is disabled, the user opens the `Buffer` class and looks in the Quick Outline for a delete method. Not finding one, she uses Eclipse's Incremental Find feature to search for the word "delete." This brings her to a comment in the `finishSaving()` method pointing out the use of flags such as `DIRTY` and `AUTOSAVE_DIRTY` to represent `Buffer` states. She also sees that the autosave file is deleted whenever the buffer is saved using the method `File.delete()` (4.34).

Returning to `LoadSaveOptionPane`, the developer adds a `JCheckBox`

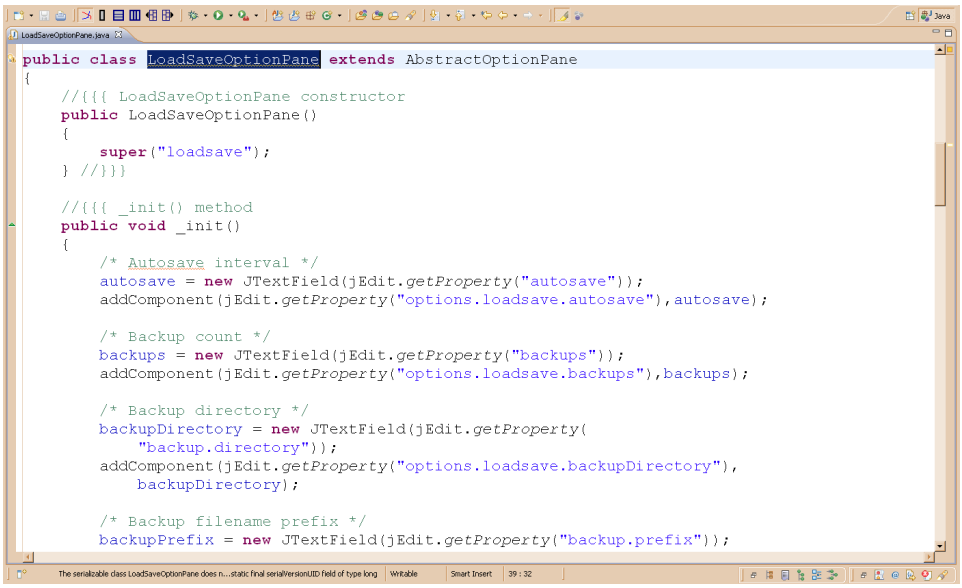
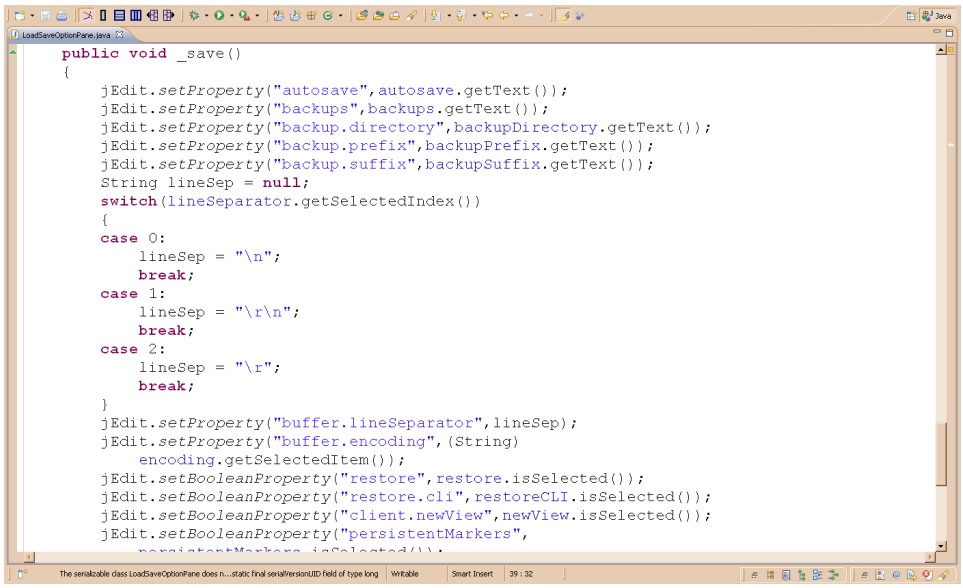


Figure 4.27: The jEdit task after opening the LoadSaveOptionPane class.

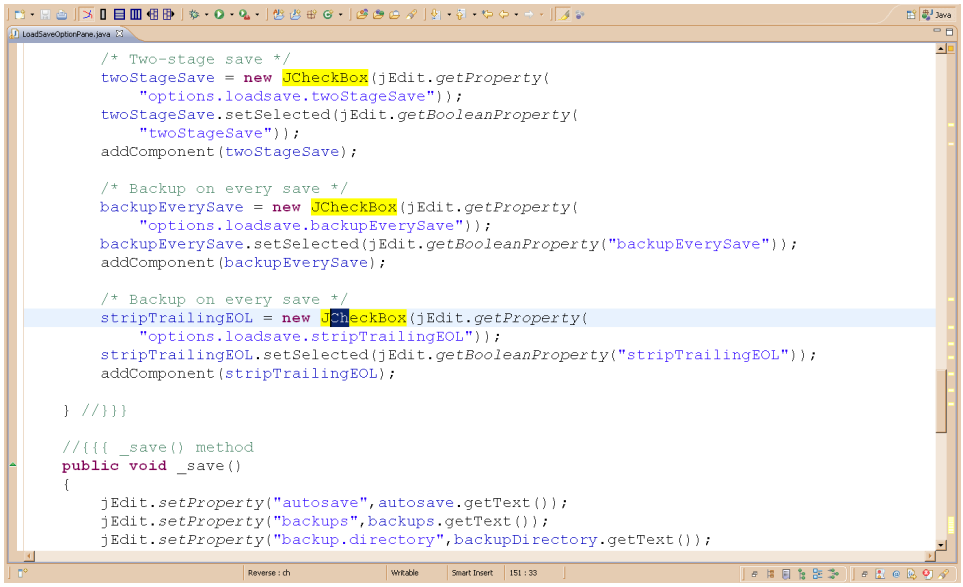
field to allow disabling autosave. In `LoadSaveOptionPane._init()`, she copies the template code found earlier and modifies it to set up the new checkbox and add it to the GUI, initializing it using a new boolean property, `"autosave.enabled"` (4.35). Then, she edits the `_save()` method so that it will set the state of this property based on the checkbox (4.36).

Returning to `Autosave.actionPerformed()`, she copies the code that retrieves the list of open files and iterates through them, then returns to `jEdit.propertiesChanged()` where she pastes this code and modifies it to call the yet to be created method `Buffer.deleteAutosaveFile()` on each open file (4.37). Using Eclipse's Quick Fix option to create the method takes her back to the `Buffer` class where she provides the implementation of the method . This requires a more careful inspection of the class to understand how the flags are used (4.38). She also modifies the `load()` method to prevent loading of autosave files when the feature is disabled. Finally, she returns to `jEdit.propertiesChanged()` and adds the necessary logic to call `Autosave.setInterval(0)` when autosave is not enabled (4.39).



```
    public void _save()
    {
        jEdit.setProperty("autosave",autosave.getText());
        jEdit.setProperty("backups",backups.getText());
        jEdit.setProperty("backup.directory",backupDirectory.getText());
        jEdit.setProperty("backup.prefix",backupPrefix.getText());
        jEdit.setProperty("backup.suffix",backupSuffix.getText());
        String lineSep = null;
        switch(lineSeparator.getSelectedIndex())
        {
        case 0:
            lineSep = "\n";
            break;
        case 1:
            lineSep = "\r\n";
            break;
        case 2:
            lineSep = "\r";
            break;
        }
        jEdit.setProperty("buffer.lineSeparator",lineSep);
        jEdit.setProperty("buffer.encoding",(String)
            encoding.getSelectedItem());
        jEdit.setBooleanProperty("restore",restore.isSelected());
        jEdit.setBooleanProperty("restore.cli",restoreCLI.isSelected());
        jEdit.setBooleanProperty("client.newView",newView.isSelected());
        jEdit.setBooleanProperty("persistentMarkers",
```

Figure 4.28: The jEdit task after scrolling to the _save() method.



```
        /* Two-stage save */
        twoStageSave = new JCheckBox(jEdit.getProperty(
            "options.loadsave.twoStageSave"));
        twoStageSave.setSelected(jEdit.getBooleanProperty(
            "twoStageSave"));
        addComponent(twoStageSave);

        /* Backup on every save */
        backupEverySave = new JCheckBox(jEdit.getProperty(
            "options.loadsave.backupEverySave"));
        backupEverySave.setSelected(jEdit.getBooleanProperty("backupEverySave"));
        addComponent(backupEverySave);

        /* Backup on every save */
        stripTrailingEOL = new JCheckBox(jEdit.getProperty(
            "options.loadsave.stripTrailingEOL"));
        stripTrailingEOL.setSelected(jEdit.getBooleanProperty("stripTrailingEOL"));
        addComponent(stripTrailingEOL);

    } //}}}

    //{{{ _save() method
    public void _save()
    {
        jEdit.setProperty("autosave",autosave.getText());
        jEdit.setProperty("backups",backups.getText());
        jEdit.setProperty("backup.directory",backupDirectory.getText());
```

Figure 4.29: The jEdit task after finding code which sets up a JCheckBox.

Figure 4.30: The jEdit task after opening the Autosave class.

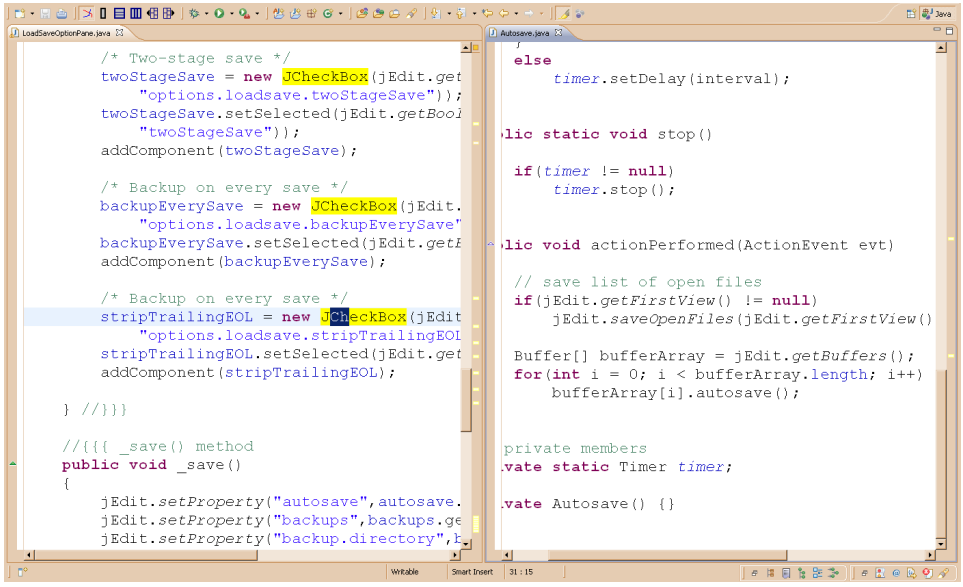


Figure 4.31: The jEdit task after scrolling to actionPerformed() in Autosave.

Figure 4.32: The jEdit task after navigating to propertiesChanged() in the jEdit class.

Figure 4.33: The jEdit task after navigating to saveSettings() in jEdit.

Figure 4.34: The jEdit task after opening the Buffer class.

Figure 4.35: The jEdit task after editing the _init() in LoadSaveOptionPane.

Figure 4.36: The jEdit task after editing the _save() method of LoadSaveOptionPane.

Figure 4.37: The jEdit task after editing jEdit.propertiesChanged().

Figure 4.38: The jEdit task after creating the method deleteAutosaveFile() in the Buffer class.

Figure 4.39: The jEdit task after further editing jEdit.propertiesChanged().

Task	Navigations		Returns
	Eclipse	Prototype	
Scroll	10	7	5
Undo	4	2	2
Line	7	3	4
jEdit	13	6	8
Total	34	18	19

Table 4.1: Case study results.

4.4 Results

Table 4.1 presents the results of the case studies. The central columns show the number of navigations for each task when using Eclipse and when using the prototype. These numbers count explicit navigations such as following a cross-reference or selecting an editor tab to bring its editor to the front (when not using the prototype). In some cases, a cross-reference is followed when using the prototype to a location which is already open and perhaps visible, but we still count this as an explicit navigation. Although it is possible the developer using the prototype would see that the location was directly accessible without following the cross-reference, we do not assume that they will. We do not consider scrolling to be navigation, partly because of the extremely unrealistically small screen used for the case studies. The final column of the table counts the number of times that a developer using the prototype would have returned to a file that was already on screen, whether they did so with an explicit navigation or not. While the difference in the first two columns measures how many explicit navigations a developer using the prototype would have saved, the last column represents how many scene changes were avoided.

The last row of the table shows the totals across all tasks. In total, the potential savings in navigations when using the prototype was 47%. Also, 56% of navigations would have taken the user back to a file that was already on the screen if she were using the prototype.

Although fairly small (especially as presented), these case studies reveal the general pattern that was experienced by the author when using the prototype for tasks of a much longer duration and which is to be expected given that it is based on Mylyn. Initially, the editor area goes through a period of evolution as files are added and resized. Then, as the degree-of-interest function stabilizes, it changes less and less as it converges to the most important files.

Chapter 5

Discussion

The author used various versions of the prototype over a period of months in the development of the prototype itself. This chapter describes some subjective impressions.

In general, the ability to see multiple files at once was very useful. Not only did it make it easier to reason across multiple files, it also made it easy to work on two sub-tasks at once, when each was mostly restricted to a different file. Each sub-task was perceived as being associated with a different part of the screen, and switching between them only required changing one's gaze. Like Mylyn's explicit support for task contexts, this made it easier to recover context when switching sub-tasks, even when these sub-tasks were not explicitly managed by Mylyn.

Often, the presence of an arrow pointed out that a relevant declaration was on the screen when it might not otherwise have been noticed, especially just after opening a file by following a cross-reference. Depending on the situation, this circumstance would either allow the user to follow a cross-reference without explicitly performing a navigation, or it would encourage them to look at information they would not otherwise have bothered to navigate to. This was generally found to be beneficial, but it does raise the possibility of distracting the user with too many details. At least once, the presence of an arrow resulted in the body of a method being glanced at without any thought. This supports the idea that simultaneously displaying important elements in different files and graphically overlaying structure on to the text might allow the user to synthesize information from multiple code locations much faster than if they had to explicitly navigate between them, no matter what navigation aids they might use.

Arrows also made it easier to locate cross-references that were known to

be present, and they were helpful in thinking about the call relationships between three or more methods without getting lost. In one case, the presence of calls to a `set` method in one code block and to the corresponding `get` method in a subsequent block, as pointed out by arrows, provided a strong hint as to the relationship between the two code blocks. When editing the code, the appearance of arrows denoting new method calls gave an increased feeling of confidence that the new code was correct, because it made it apparent that it had the right structure. Also, when editing every call to a method in a piece of code, the arrows identifying that method acted like the Mark Occurrences feature of Eclipse, with the advantage that they remained visible when the cursor was moved.

Unfortunately, the author had less experience with progressive elision, partly due to a bug that went unfixed for a long while, but also because it required more explicit action than the other features of the prototype. It is possible that making the elision slider a fixed part of the interface rather than a popup would encourage it to be used more. However, to the extent that progressive elision was used, it did not cause confusion and the transition between elision levels was not difficult to follow. Setting the elision level so that blank lines, delimiters and comments were hidden was found to be quite useful as it allowed the information density in the editor to be increased without significantly altering the appearance of familiar code.

Chapter 6

Related Work

The approach discussed in this dissertation seeks both to reduce the amount of navigation which the programmer must perform and to display those pieces of code which are most interesting in the context of a given task. Therefore, we structure our discussion of related work in terms of two somewhat overlapping categories. The first, discussed in Section 6.1, contains tools which try to ease the navigation task by providing some kind of outline or map of a part of a system. The second, discussed in Section 6.2, contains tools which provide a view, usually effective, of a set of program elements or code snippets which would otherwise be more scattered. This is achieved either by augmenting the editor with new features or by using special constructs to bring scattered elements together within the editor.

6.1 Navigational Aids

JQuery [12], described in Section 1.1, uses a query language to build a tree-structured model of some of the entities and relationships making up a part of a system. This provides a convenient navigable index to relevant source code locations while also helping the user to keep track of her exploration path. However, the IDE still only displays one file at a time. In contrast, Relo [22] allows the user to build a two-dimensional graphical diagram, similar to a UML diagram, by explicitly indicating the relationships and nodes to be included, rather than through queries. The user can incrementally expand the diagram by following inheritance, method call, and other relationships, or the diagram can be automatically constructed from the user's navigations in Eclipse. Notably, Relo allows the user to expand individual methods in the diagram to reveal their source code, and even allows the code

to be edited. Relo thus makes it possible to juxtapose code from different files, once the relevant items have been added to the diagram. However, because it would be cumbersome to perform a task involving significant reading or editing of the code using Relo alone, the user must switch between Relo and the standard Eclipse views. In fact, users in a preliminary evaluation of Relo were given two monitors, one for Eclipse and one dedicated to Relo. The approach described in this dissertation makes juxtaposition more natural by incorporating it into the standard Eclipse environment, rather than providing a special view. Still, this approach could be used in concert with Relo.

In [17], the behavior of developers performing maintenance tasks was analyzed and the authors concluded that tool support for collecting and simultaneously viewing code fragments and documentation could significantly reduce the time that developers spend performing navigations. Based on this study, JASPER [3] is an Eclipse plugin which allows the user to manually build a storable collage of "artifacts" called a working set. Each artifact is a contiguous region of code, a text note, or a web page. Code artifacts are not editable because of their tiny size and lack of surrounding context, but they link back to the code, so the working set serves as a sort of two-dimensional outline that also supports direct comparisons of code fragments. A working set might represent a complete task context or a more focused concern, and can be saved as documentation for future maintenance work. A key advantage of this approach is that it allows developers to keep track of interesting code at the level of arbitrary code fragments rather than at method and field granularity. However, this leads to difficulties keeping the working set synchronized with the source code, and users must explicitly indicate the code fragments to be added to the set.

6.2 Source Code Views

6.2.1 Fisheye Views

Progressive elision bears some resemblance to fisheye views [7] of source code which are intended to hide irrelevant details to make room for higher level context. Given that a particular line of code is the focal point of interest, a degree of interest function, unrelated to the Mylyn degree-of-interest, assigns an interest value to each line of code based on its distance from the focal point in the abstract syntax tree and its overall indentation level, with less indented lines and lines close to the focal point receiving higher scores. Lines with interest values below a threshold are elided. While this provides an excellent picture of the location of the focal point within the overall structure of the file (see Figure 6.1), much of this structure may be either irrelevant to the task at hand or already in the developer's head, and highly relevant details may be hidden. The ability to activate a fisheye view on demand seems likely to be useful, but they may not be suitable as the primary view of the code (hence their lack of widespread adoption in source code editors more than two decades after they were proposed).

Jaba [4] is an experimental Java programming environment which combines fisheye visualization and code folding with elements of literate programming. The editor and linked outline view both present a class as a hierarchy of collapsible chunks, each of which can be named. Chunks can correspond to methods or blocks, in which case they are automatically defined, or they can be arbitrary regions of code, defined by the user through the automated insertion of special comments. The user can collapse or expand individual chunks to explore the hierarchy, or she can automatically collapse or expand all chunks of a certain type. If she specifies a focal point, the set of expanded chunks can be automatically determined using the fisheye degree of interest function. As in our approach, an interest threshold slider allows the user to control how much of the file is elided.

Jakobsen and Hornbæk [10] present an Eclipse plugin that replaces the standard Eclipse Java editor with a fisheye view of the source file. In contrast to previous work, the editor is divided into separate focus and context

```
   1 #define DIG 40
   2 #include <stdio.h>
...4 main()
   5 {
   6       int c, i, x[DIG/4], t[DIG/4], k = DIG/4, noprint = 0;
...8       while((c=getchar()) != EOF){
   9             if(c >= '0' && c <= '9'){
..16             } else {
  17                   switch(c){
  18                         case '+':
..27                       case '-':
..38                       case 'e':
>>39                             for(i=0;i<k;i++) t[i] = x[i];
  40                             break;
  41                       case 'q':
..43                       default:
..46                 }
  47                 if(!noprint){
..57                 }
  58             }
  59             noprint = 0;
  60       }
  61 }
```

Figure 6.1: A fisheye view of a C program (taken from [7]). Line numbers are shown at left and ellipses indicate elided text. Line 39 is the focal point.

areas, with the focus area in the middle acting like a normal Java editor and the non-editable context areas above and below it displaying lines selected using the degree of interest function, modified to incorporate a measure of semantic distance from the focus area and fixed rules about the relative importance of different kinds of source code lines. While using a focus area saves the user from having to indicate which single line should be the focal point, this approach has the disadvantage that the context area changes irregularly as the user scrolls. The authors use the overview+detail technique, providing an overview of the file along the side of the editor which shows its structure by rendering the entire file in an unreadable microfont. The mapping between the overview and the lines of code displayed in the context area is indicated graphically, allowing the user to see the location of context lines within the file. Additionally, lines in the context area with lower interest scores are displayed in a smaller font, despite previous evidence [5] that reduced font sizes result in slower performance than elision.

Progressive elision could be thought of as a fisheye view with multiple focal points defined by the task context and a degree of interest function based on Mylyn instead of syntactic distance measures. However, progressive elision is much less focused on structure than a true fisheye view: with the exception of method headers, no attempt is made to ensure that the syntactic ancestors of each visible line are also visible. This substantially increases the density of interesting information (in the Mylyn sense). Finally, because the semantic distance function used by Jakobsen and Hornbæk puts an emphasis on *declarations* of elements referenced in the focus area, their context area appears to function similarly to a Mylyn filtered outline. This is in contrast to our approach, which gives importance to lines which *reference* interesting elements.

6.2.2 Modularizing Views

Together with Mylyn, our work can be seen as a lightweight approach to virtual remodularization in that the developer's view of the code changes to reflect her current focus. Fluid AOP [9] is a more explicit technique, using pointcuts to localize scattered code within a single editor which propagates edits back to the appropriate locations. This supports the on-demand creation of crosscutting views which, even though they are effective, do not constrain the underlying code by refactoring it into a new decomposition. Rather, fluid AOP allows the user to create and work with multiple overlapping decompositions, each of which is suitable for a different task. While our approach uses views created implicitly from the user's interaction history and is not restricted to sets of elements that can be identified by pointcuts, it is still possible to use a query facility such as Eclipse's built-in search to gather scattered components into a single view. However, fluid AOP can go further, overlaying multiple similar pieces of code into one partially editable abstraction. Because fluid AOP allows a single edit to be propagated to multiple locations, it can also save the developer from having to make duplicated edits. To some extent, we could achieve the same benefit by incorporating some form of linked editing [24], however, without a mechanism

similar to pointcuts, the user would have to manually select the regions to be edited.

Decal [11] is an experimental object-oriented programming system which supports two mutually crosscutting decompositions, one which partitions the code into classes, and another which cuts it into modules. Each program element belongs both to a class and to a module, and virtual source files [1] are used to provide both a class-centric view and a module-centric view of the code. Because both classes and modules are linguistic constructs, these views are not derived from the program, they *are* (equally important facets of) the program, and they can be edited with clearly defined semantics.

Visual separation of concerns [2] supports remodularization by altering the program storage model, rather than using a linguistic or a purely presentational approach. A program is stored as a collection of potentially nested program elements such as classes, methods, and fields, and the editor is free to show whatever subset of the program elements is desired, without being constrained to showing a single and complete compilation unit. In addition to supporting multiple mutually crosscutting views, this technique permits revision control systems to operate at a granularity smaller than the file, potentially enabling automatic concern identification through the application of data mining techniques to the resultant revision histories.

Chapter 7

Future Work

Areas of future work include exploring other ways to leverage the multiple file interaction paradigm, investigating how to better support navigation history in a multiple file interface, and improving progressive elision. Importantly, user studies should be done to better evaluate the usefulness of this approach.

7.1 Taking Multiple File Interaction Further

While our prototype displays multiple files at once, the user can only interact with one file at a time.[12] Future work could extend some of the operations that currently apply to a single file so that they can be used across multiple files. Potential candidates include the find/replace operation and Mark Occurrences. Additionally, the Quick Outline view might be extended with the option to include all visible files. If Eclipse added support for selecting multiple blocks of text, allowing the user to copy, format, indent, or comment/uncomment them all at once, this would extend naturally to multiple files.

We could explore what other kinds of graphical annotations are enabled by the multiple file interaction paradigm. For example, we could indicate overrides and implements relationships and navigation paths taken by the user, perhaps incorporating "brushing," where relationships are indicated only when the mouse passes over one of their participating elements. We could even allow the user to add their own annotations by drawing on the

[12] Eclipse allows text to be dragged from one file to another, but unfortunately, unlike copy and paste, this does not automatically update `import` statements. This is probably because the drag and drop feature was only intended to be used with one file at a time.

code, associating the drawings with the current task.

Layouts could be bookmarked or saved and shared as in JASPER [3], and when activating a Mylyn task context, the user could have the option to select from previously saved layouts.

Finally, given a sufficiently large screen, some form of prediction could be used to automatically populate the display. For instance, an artifact recommender such as [18] could periodically inject a new file into the display.

7.2 Better Navigation History

Eclipse maintains a history list of previously visited code locations and allows the user to move backward and forward through this list much as they would when using a web browser. Unfortunately, this feature is not as useful as it could be. For instance, the list only displays file names, making it impossible to distinguish multiple locations within the same file. Also, the tendency of programmers to move back and forth between the same locations may lead to many more duplicated history entries than is common when browsing the web, and Eclipse provides no special features to handle this. When multiple files are displayed side by side, repeatedly clicking on the back button can cause the cursor to jump back and forth between editors without actually revealing any new information, causing confusion and frustration. The multiple file interaction paradigm thus renders Eclipse's history feature even less useful.

Our prototype augments Eclipse with a history of editor layouts, but it could be more useful to provide a hybrid history list that includes both previous layouts and code locations within a layout, perhaps excluding locations which do not cause an editor either to open, to cease to be hidden, or to reveal a different location.

Another approach would be to provide separate navigation histories for each editor. Synergistically, the option to follow a cross-reference to another file without opening a new editor could also be provided. This would mean that when following a chain of cross-references to some destination, the user could choose whether to open each new file in a new editor, leaving the

navigation history visible on the screen, or to follow cross-references within the same editor, thereby preventing the allocation of screen space to uninteresting files visited along the way. Recently, Sherwood has experimented with giving the Eclipse user exactly this kind of control [21] in the context of the single file interaction paradigm, resulting in an interface which is very similar to a tabbed web browser. She observed that users exploited this power to mark waypoints for future exploration; combining her approach with ours would associate these waypoints with locations on the screen.

7.3 Improving Progressive Elision

The line-level interest function used in progressive elision is quite straightforward and could probably be improved. For instance, lines which do not contain any field or method references, such as those using only local variables, could have their interest level set based on the interest levels of nearby lines which use the same variables. Similarly, method and field declarations with a low degree-of-interest could have their line-level interests increased when they are used by interesting methods in the file. Both of these ideas would add an informal flavour of program slicing [25] to progressive elision. It might also be interesting to experiment with a two-dimensional elision control, where the user can adjust both the amount of elision and the relative weighting of the degree-of-interests of its parent declaration and its references. Another possibility would be to extend the Mylyn degree-of-interest model to include statements or even expressions, tracking their interest in terms of developer interactions with them. While it might not be possible to infer that the developer has read some piece of code, it would give the developer the power to explicitly indicate that a fragment of code is interesting, as JASPER [3] does.

Progressive elision could also be used as a form of semantic zooming when navigating within a file. Instead of scrolling a large distance, the user could zoom out, scroll a short distance, and then zoom back in; we have not explored this possibility. To better facilitate this use, we could provide explicit support for returning to a previous level of elision.

Finally, the kind of overview used in Jakobsen and Hornbæk's fisheye editor [10] (see Section 6.2.1) could also be used with progressive elision, perhaps in the form of a popup, to graphically map visible code fragments to their locations within the file. This could help to keep the user oriented when she changes the level of elision and might also help to compensate for the lack of structural information in partially elided methods as compared to fisheye views [7]. The overview could also be used to display Eclipse annotations, including annotations in elided code.

7.4 User Studies

An important piece of work that remains to be done is a more complete evaluation of our approach. User studies could investigate whether it actually improves programmer productivity, what effect, if any, it has on programmers' mental models, whether programmers are able to remember relevant code in terms of its location on screen, and whether programmers actually like using the system. It would also be worthwhile evaluating how the various facets of the system affect programmers' ability to make sense of what they see and to comprehend the system they are working on. For instance, we could explore whether users can make sense of the transition between more and less elided views, whether using elision in this way really makes the context "feel" accessible, and whether it prevents the user from discovering that hidden information is actually important. It would be interesting to find out if the code fragments displayed in partially elided elements are readily identifiable, if they provide useful context, and if they can reasonably be edited. It should also be established whether using a display which is larger, has a higher information density, and consists of more parts causes any significant problems. Finally, users may find graphical annotations annoying or distracting and may prefer them to be used in a more on-demand fashion.

Chapter 8

Conclusion

This research proposes that IDEs take advantage of increasing screen sizes by juxtaposing parts of multiple files in order to present as many of the interesting parts of the code as possible. We illustrated the feasibility of this concept with a prototype that uses Mylyn to allocate screen space to files and to provide a variable level of elision within files. We also used graphical annotations in an attempt to help the user understand the connections within and across files. Case studies were presented to illustrate the approach in action. They also provided a rough measure of 50% as the potential reduction in the time spent navigating.

While this approach can be seen as providing a new view of a Mylyn task context, allowing the user to work with it at the source code level, it could also be seen as a presentation-level mechanism for modularity. Progressive elision is intended to allow you to move smoothly between a viewing a complete source file and progressively smaller subsets which are focused on a task. Unlike traditional code folding approaches and approaches based on virtual source files, the boundaries of these views are not crisply defined. Displaying multiple editors at once is meant to support the developer in thinking outside the class, juxtaposing the elements of a concern even when they are scattered across multiple files, but without orphaning them from their lexical context. Using graphical annotations, quite possibly beyond those implemented here and perhaps including ad-hoc user-created annotations (e.g. by allowing the user to draw on the code) might help to transform the display into something more than a text editor, illuminating connections within the code and supporting the user in thinking about a concern as a concrete entity.

Bibliography

[1] Mark C. Chu-Carroll, James Wright, and David Shields. Supporting aggregation in fine grained software configuration management. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 99–108, New York, NY, USA, 2002. ACM.

[2] Mark C. Chu-Carroll, James Wright, and Annie T. T. Ying. Visual separation of concerns through multidimensional program storage. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 188–197, New York, NY, USA, 2003. ACM.

[3] Michael J. Coblenz, Andrew J. Ko, and Brad A. Myers. JASPER: an Eclipse plug-in to facilitate software maintenance tasks. In *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 65–69, New York, NY, USA, 2006. ACM.

[4] Andy Cockburn. Supporting Tailorable Program Visualisation Through Literate Programming and Fisheye Views. *Information and Software Technology*, 43(13):745–758, 2001.

[5] Andy Cockburn and Matthew Smith. Hidden Messages: Evaluating the Efficiency of Code Elision in Program Navigation. *Interacting with Computers: The Interdisciplinary Journal of Human-Computer Interaction*, 15(3):387–407, 2003.

[6] Michael Desmond, Margaret-Anne Storey, and Chris Exton. Fluid Source Code Views for Just In-Time Comprehension. In *Workshop on*

Software Engineering Properties of Languages and Aspect Technologies, Bonn, Germany, March 2006.

[7] G.W. Furnas. Generalized Fisheye Views. In *Human Factors in Computing Systems III. Proceedings of the CHI'86 conference*, pages 16–23, Amsterdam, 1986. ACM.

[8] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[9] Terry Hon and Gregor Kiczales. Fluid AOP Join Point Models. In *Proceedings of the Asian Workshop on Aspect-Oriented Software Development*, pages 14–17, 2006.

[10] Mikkel R. Jakobsen and Kasper Hornbæk. Evaluating a fisheye view of source code. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 377–386, New York, NY, USA, 2006. ACM.

[11] Doug Janzen and Kris de Volder. Programming with Crosscutting Effective Views. In *Proceedings of the European conference on object-oriented programming*, pages 197–222, Oslo, 2004. Springer-Verlag.

[12] Doug Janzen and Kris De Volder. Navigating and Querying Code Without Getting Lost. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187, New York, NY, USA, 2003. ACM.

[13] Mik Kersten, Matt Chapman, Andy Clement, and Adrian Colyer. Lessons Learned Building Tool Support for Aspectj. In Matt Chapman, Alexandre Vasseur, and Gnter Kniesel, editors, *AOSD 2006 - Industry Track Proceedings*, Bonn, Germany, March 2006.

[14] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD '05: Proceedings of the 4th international conference on*

Aspect-oriented software development, pages 159–168, New York, NY, USA, 2005. ACM.

[15] Mik Kersten and Gail C. Murphy. Using Task Context to Improve Programmer Productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM.

[16] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[17] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[18] Martin P. Robillard. Automatic Generation of Suggestions for Program Investigation. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 11–20, New York, NY, USA, 2005. ACM.

[19] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004.

[20] Martin P. Robillard and Gail C. Murphy. FEAT: a tool for locating, describing, and analyzing concerns in source code. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 822–823, Washington, DC, USA, 2003. IEEE Computer Society.

[21] Kaitlin Duck Sherwood. Path Exploration during Code Navigation.

Master's thesis, The University Of British Columbia, Vancouver, B.C., Canada, July 2008.

[22] V. Sinha, D. Karger, and R. Miller. Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 187–194, Sept. 2006.

[23] Margaret-Anne Storey, Li-Te Cheng, Ian Bull, and Peter Rigby. Waypointing and Social Tagging to Support Program Navigation. In *CHI '06: CHI '06 extended abstracts on Human factors in computing systems*, pages 1367–1372, New York, NY, USA, 2006. ACM.

[24] M. Toomim, A. Begel, and S.L. Graham. Managing Duplicated Code with Linked Editing. *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 173–180, Sept. 2004.

[25] 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.