Mylar: a degree-of-interest model for IDEs

Mik Kersten and Gail C. Murphy
Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC, V6T 1Z4, Canada
{beatmik, murphy}@cs.ubc.ca

ABSTRACT

Even when working on a well-modularized software system, programmers tend to spend more time navigating the code than working with it. This phenomenon arises because it is impossible to modularize the code for all tasks that occur over the lifetime of a system. We describe the use of a degree-of-interest (DOI) model to capture the task context of program elements scattered across a code base. The Mylar tool that we built encodes the DOI of program elements by monitoring the programmer's activity, and displays the encoded DOI model in views of Java and AspectJ programs. We also present the results of a preliminary diary study in which professional programmers used Mylar for their daily work on enterprise-scale Java systems.

Keywords

Development tools, software evolution, program structure, program views, software tasks, task representation

1. INTRODUCTION

To make a change to a large software system, programmers must repeatedly visit multiple places in the code. For instance, they may need to examine the code corresponding to a cohesive portion of the program's execution as part of a change task. In an object-oriented program, this code is typically scattered across several methods in multiple classes. In an aspect-oriented programming (AOP) [11] implementation, this code may also be spread across advice in one or more aspects.

Integrated development environments (IDEs) views are intended to help a programmer find, examine, and navigate between the places of interest in the code. For example, a cross-reference search can produce a view listing all of the methods called from a constructor of interest. To support AOP, the IDE may provide a view showing all of the methods affected by a particular advice. When the system is small, the elements related to the task-at-hand are easy to find in such views. However, as the size of the system increases the utility of these views decreases. The elements relevant to the current task become a small subset of those shown in the IDE views. As a result, in an enterprise-scale system the large number of elements unrelated to the current task occludes the relevant information. Current IDE's force programmers to work with long scrolling lists when trying to find the elements related to a task, and to synthesize information spread across multiple lists in different views. As a result, programmers spend more time looking for related information than they do performing the task.

To help programmers focus and work on the code related to a task, we have developed the Mylar[1] tool. Mylar monitors a programmers' activities and captures the relevance of code elements to their task in a degree-of-interest (DOI) model, loosely based on the model introduced by Card [1] (Section 3). For example, when a programmer selects or edits a program element, Mylar increases the interest level of that element. Mylar uses the DOI model to populate Java [5] and AspectJ [10] views within the Eclipse[2] IDE

To provide initial evidence that there is value in this approach, we present the results of a diary study in which six senior programmers at IBM used the Mylar tool in their daily work on enterprise-scale systems implemented in Java (Section 4). Based on this experience, we have built additional IDE views and features intended to help programmers work with crosscutting and inheritance structure in large systems (Section 5.3). The use of Mylar has also provided feedback about how the DOI model can evolve to better capture and represent a programmer's activities, and how we can make views actively find and display code relevant to the current context (Section 5.1). Mylar is unique in surfacing the scattered code related to a task without requiring special behavior on the part of the programmer (Section 6).

2. EXAMPLE

Consider the case of a programmer trying to understand why some of the test cases for de-serialization are failing in the moderately sized Web Services Invocation Framework (WSIF)[3]. To complete this debugging task, the programmer must examine the test cases, the classes that are failing to de-serialize, and the serialization policy employed in the system. To describe the limitations of existing IDE views when working with this task, we

[1] Mylar is also a polyester film used for solar eclipse viewing. The Mylar tool is a DOI viewing layer for the Eclipse IDE.

[2] http://eclipse.org, AspectJ plugin: http://eclipse.org/aspectj

[3] http://ws.apache.org/wsif (1,897 classes)

first present the use of the Eclipse Java Development Tools (JDT) and then the AspectJ Development Tools (AJDT). We then introduce the Mylar views, which extend the JDT and AJDT.

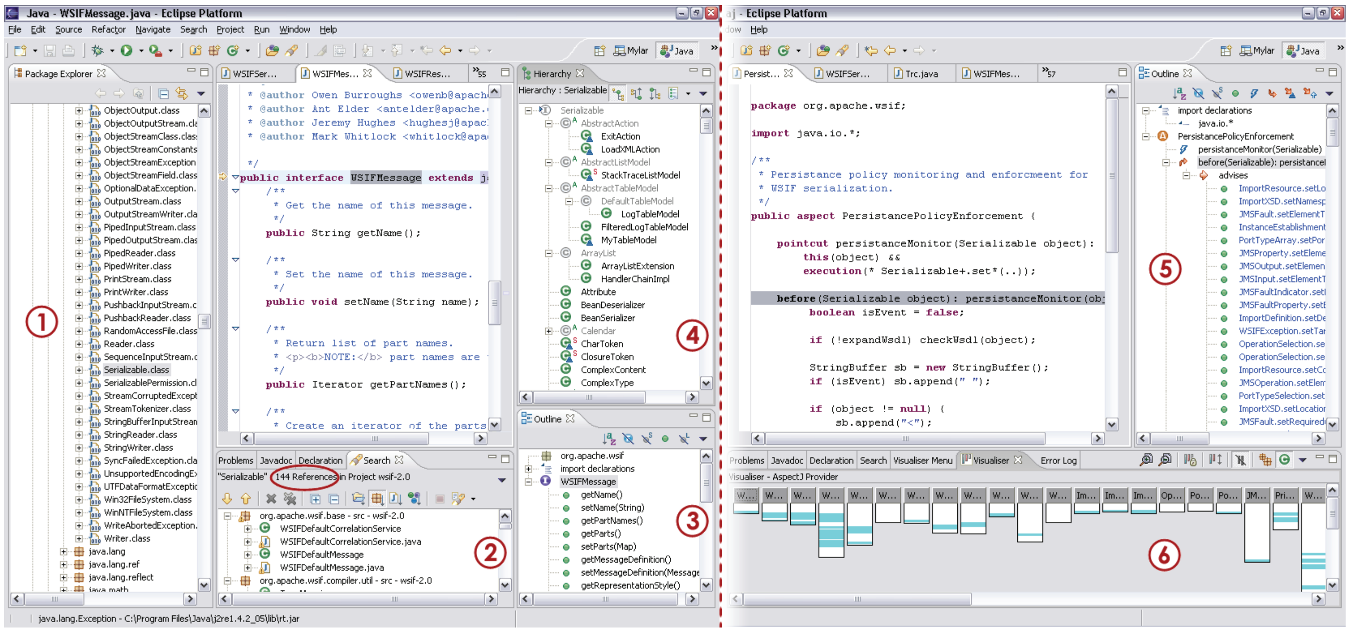
Using the Eclipse JDT, the programmer decides to find all subtypes in the WSIF code base that implement the `Serializable` interface, and to inspect the setter methods in those classes. Despite the support that Eclipse provides in the Java Type Hierarchy view and search functionality, the programmer finds it a lengthy and tedious process to investigate the relevant system structure, in part because the classes involved with the particular failures are a small subset of the de-serialization concern in WSIF. Figure 1 shows a snapshot of Eclipse after finding the `Serializable` interface, inspecting the interface in the hierarchy, and searching for all references to `Serializable` within the WSIF project.

1. The Package Explorer has become difficult to use because it includes thousands of nodes—a result of only a handful of navigation clicks through project files and related library classes. Hierarchical relationships are no longer visible without scrolling through the tree.
2. Using the Java Search features to look for references to `Serializable` within the project returns 144 items. There is no convenient way to search for only those elements related to the context of the failing test cases. Instead, the search result list requires manual inspection to find elements of interest.
3. The Outline view is less populated than the other views since the current class is small, but it still needs to be scrolled to find the setter methods of interest.
4. The Type Hierarchy shows all types in the project that extend `Serializable`, and contains thousands of elements. Despite the fact that the list is limited to the project's working set of elements, a very small subset of the types in the view is involved in the failing test cases of interest.

Part of the problem in this scenario is that the serialization policy is a crosscutting concern. To cleanly capture the structure and behavior of this concern, the programmer can use AspectJ to express the serialization policy in a single aspect as shown in Figure 1. The AJDT Eclipse plugin makes the resulting crosscutting structure explicit in the Outline view. Although the modularity of the code base improves, the programmer finds that the AJDT views manifest similar problems to the Java views.

5. As the programmer explores the many "advises" links looking for those related to the failure, the Package Explorer tree expands. The need to navigate the crosscutting effects of advice, spread across many classes, causes the number of elements in the Package Explorer to grow even faster than it did in the Java case.
6. The Aspect Visualiser view contains many files, the names of which are hard to read. This view communicates that the aspect affects many places in the current project. But it does not indicate which parts of this crosscutting structure are interesting to the task-at-hand. The large number of affected source lines presented occludes the subset of join points related to the failing test cases, and the programmer must manually inspect the view by repeatedly zooming, scrolling, and navigating.

Whether used with plain Java or AspectJ these IDE views fail to show the subset of the structure relevant to the programmer's task. The views cannot be configured in a way that captures the elements that the programmer needs to edit, the inheritance context of the `Serializable` classes that are causing the test case to fail, and the crosscutting context of the system-wide serialization policy. The IDE tools show whole-system slices of the program structure rather than helping the programmer focus on the program elements important to the task-at-hand. When working on any task not encapsulated by a single file module or structure view, the programmer must navigate between files, repeatedly refer to lists of open files, perform multiple searches,



```
 * @author Owen Burroughs <owenb@apach
 * @author Ant Elder <antelder@apache.
 * @author Jeremy Hughes <hughesj@apac
 * @author Mark Whitlock <whitlock@apa

 */
public interface WSIFMessage extends j
    /**
     * Get the name of this message.
     */
    public String getName();

    /**
     * Set the name of this message.
     */
    public void setName(String name);

    /**
     * Return list of part names.
     * <p><b>NOTE:</b> part names are
     */
    public Iterator getPartNames();

    /**
     * Create an iterator of the parts
```

"Serializable" 144 References in Project wsif-2.0

```
package org.apache.wsif;

import java.io.*;

/**
 * Persistance policy monitoring and enforcmeent for
 * WSIF serialization.
 */
public aspect PersistancePolicyEnforcement {

    pointcut persistanceMonitor(Serializable object):
        this(object) &&
        execution(* Serializable+.set*(..));

    before(Serializable object): persistanceMonitor(obj
        boolean isEvent = false;

        if (!expandWsdl) checkWsdl(object);

        StringBuffer sb = new StringBuffer();
        if (isEvent) sb.append(" ");

        if (object != null) {
            sb.append("<");
```

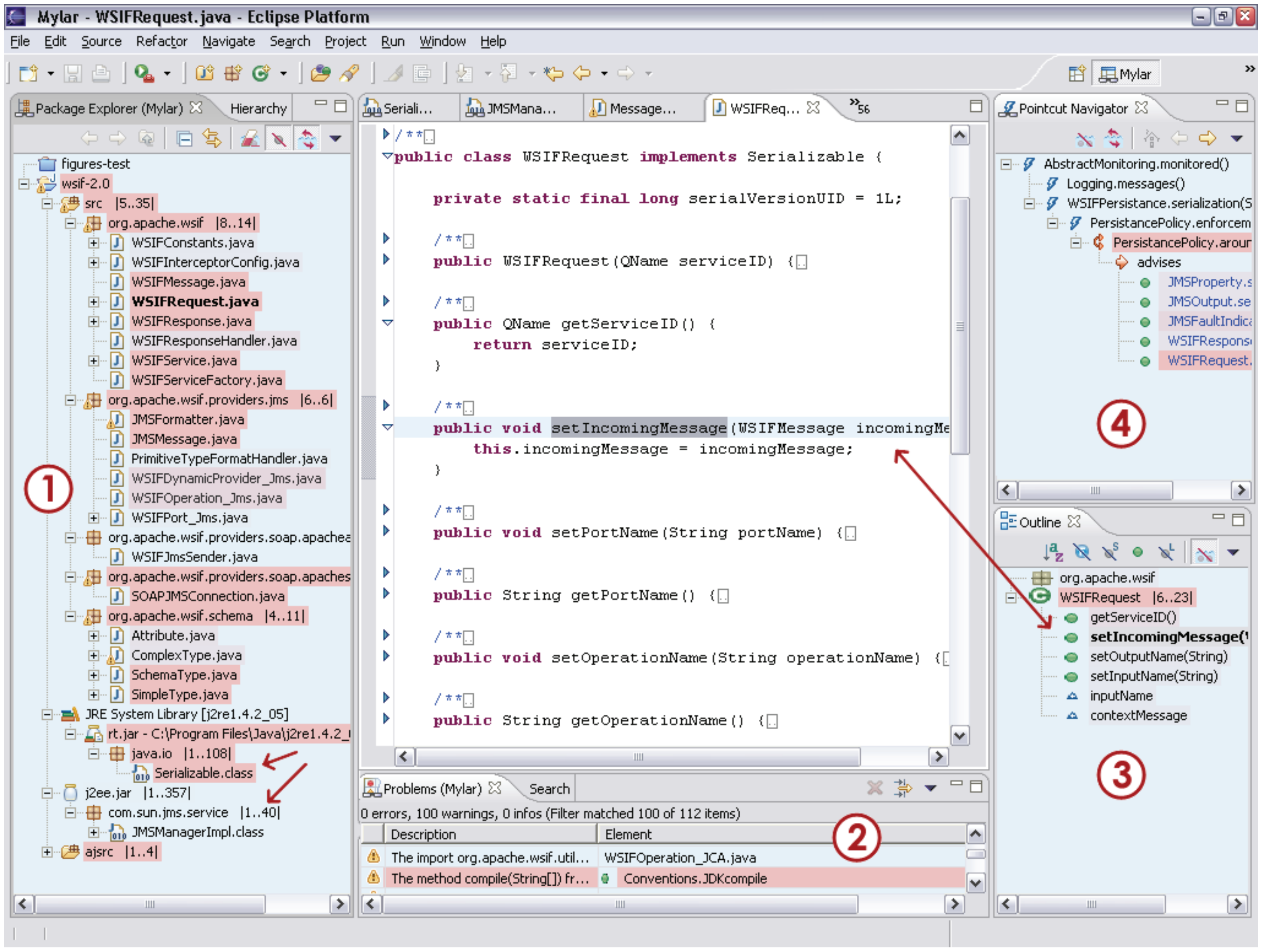
Java - WSIFMessage.java - Eclipse Platform
Package Explorer
Hierarchy
Outline
Search
Visualiser - AspectJ Provider
1
2
3
4
5
6

Figure 1: Java project, left and AspectJ project, right (figure numbers correspond to list items above)

and repeatedly inspect search results looking for those relevant to the task. The programmer must commit the context of the task to memory. The burden of filtering the views based on the task context is placed on the programmer instead of the tool.

To address these problems we built the Mylar Eclipse plugin, which automatically encodes the context of the programmer's task in a DOI model and exposes it in IDE views (Figure 2). The default highlighting scheme visible uses colored shading to indicate the programmer's relative interest in the element. A darker shade indicates a higher DOI. Since Eclipse already uses highlighting to indicate the currently selected element, Mylar uses bold font to indicate the currently selected element. Figure 2 shows how the Mylar views present program elements related to the task context. Section 3.2 describes the views in detail.

1. Mylar Package Explorer: interest-based filtering is enabled, so only the files and libraries relevant to the task are visible. The number of filtered elements is indicated on the parent label. The *auto expand and filter* mode reduces the need to manually expand and scroll the tree by actively maintaining the visibility of high-interest elements, which helps bring the hierarchical relationships into view. Note that a vertical scrollbar can appear in the Mylar views, but is less common when interest-based filtering is enabled. A highlight-only mode can be toggled, in which no elements are filtered and items of interest stand out through highlighting.
2. Mylar Problems List: problems of interest are highlighted to stand out from the large number of items typically populating this view. This view is populated identically to the JDT/AJDT problems list, but corresponding program elements are additionally displayed and used to highlight the DOI of the problem.
3. Mylar Outline: interest-based filtering is turned on to show only the members related to the task.. The Mylar editor has an option to actively fold and unfold elements according to interest—reflecting the filtering state of the Mylar Outline. If advice links are present in the Mylar Outline view (as in Figure 1, #5) they appear similar to the links visible in the Active Pointcut Navigator.
4. Active Pointcut Navigator: this view is actively updated to show how high-interest elements fit into the crosscutting structure of the system (Section 5.3.2).



```
/**
public class WSIFRequest implements Serializable {

    private static final long serialVersionUID = 1L;

    /**
    public WSIFRequest(QName serviceID) {

    /**
    public QName getServiceID() {
        return serviceID;
    }

    /**
    public void setIncomingMessage(WSIFMessage incomingMe
        this.incomingMessage = incomingMessage;
    }

    /**
    public void setPortName(String portName) {

    /**
    public String getPortName() {

    /**
    public void setOperationName(String operationName) {

    /**
    public String getOperationName() {
```

0 errors, 100 warnings, 0 infos (Filter matched 100 of 112 items)

	Description	Element
⚠	The import org.apache.wsif.util...	WSIFOperation_JCA.java
⚠	The method compile(String[]) fr...	● Conventions.JDKcompile

Figure 2: Mylar views (figure numbers correspond to list items above)

3. MYLAR

Mylar monitors programmer activity in the Eclipse IDE, encodes it into a DOI model, and displays the interest level in structure views.

3.1 Mylar DOI model

The Mylar DOI model is loosely based on the model proposed for DOI trees [1]. The Mylar model associates an interest value with each Java or AspectJ program element. When a program element is selected or edited, its DOI value increases. Over time, if the element is not selected or edited, its interest value decays. At any point in time, the interest values of the program elements reflect a relevance ranking of an element to a particular task. The model does not store any structural relationships between elements or navigation paths, and instead relies on structure views to display the relationships between interesting elements (Section 5.3).

From the programmer's point of view, the model represents the subset of program elements in the IDE that are relevant to the current task. The accuracy and stability of the model are determined by parameters for interest increase, decrease, and periodic decay. If interest increases too eagerly, the structure views will become overly populated with elements. Too fast a decay can result in only a handful of recently-selected elements remaining in the model. We discuss the tuning of encoding and decay parameters in Section 3.3.

One of our goals is to make the Mylar model predictable enough for programmers to avoid the frustrations that users have experienced with adaptive interfaces [3]. As a result, we designed the model to be an encoding of programming activity rather than a machine learning or statistical process. In the cases where there is a mismatch between the elements in the model and the task, the programmer can set the interest on an element manually, or erase the entire model (e.g., if they move on to a different task). Both of these events are logged in detail, and have provided input for making the Mylar model more aware of the programmer's task (Section 5.1).

3.2 Displaying DOI in structure views

The driving principle of the Mylar User Interface (UI) design is to surface the DOI model seamlessly in the Eclipse views that display program elements. For example, Mylar overlays the interest level in the editor and in the three views that appear by default in the Java perspective: the Package Explorer, Outline, and Problems List views. The Mylar versions of the editor and views provide a superset of the functionality of those that they replace and are fully compatible with the programmers' existing use of those views. The Mylar views are intended to be used instead of the corresponding Java and AspectJ views. However, there is no restriction in how the views are set up, and both Mylar and standard views can be used simultaneously.

Mylar visualizes the DOI of a program element through highlighting. The default highlighting mode visible in Figure 2 is a hot/cold color scheme, where hot means interesting. As an element becomes more interesting its highlight color darkens. At this stage of the implementation, the goal was to ensure that programmers noticed the highlighting (rose coloring) and could distinguish the Mylar views from the standard views (light blue background). Determining the effectiveness of the highlighting scheme is left to future work.

Mylar views also support the filtering of uninteresting elements (Figure 2, #1). To make the filtering explicit, a parent element is annotated to show how many elements of the total number of children are visible (e.g., "3..10"). A parent node that filters children can be asked to temporarily show all its children so that a filtered element can be quickly added to the visible elements of interest. In views that support collapsing elements, such as the Package Explorer tree view and the Java editor folding support, automatic management of the expansion state of the views and editor ensures that only elements of interest are visible.

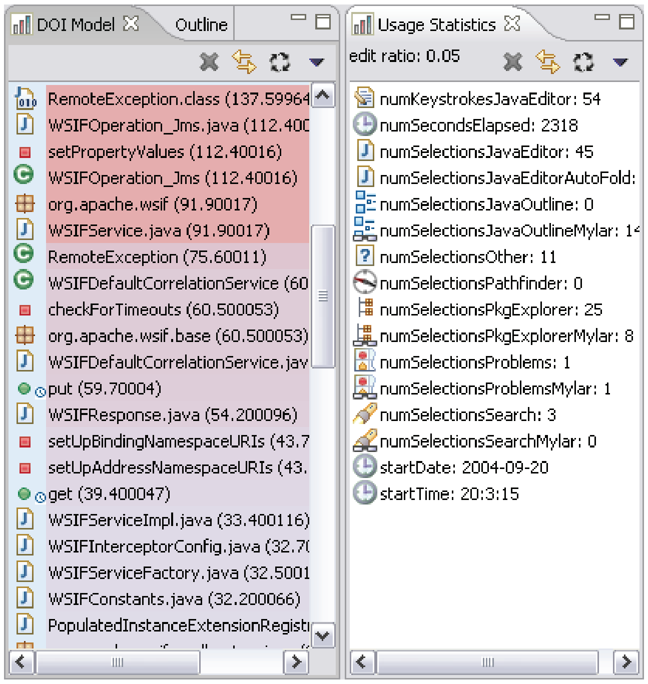
3.3 Integrating Mylar into Eclipse

Our design goals for Mylar include tight integration with Eclipse, production-quality performance, and robustness. These goals are required to support the study of real-world use of Mylar on large systems (Section 4). To help meet these goals, we decided to integrate the Mylar model with the existing Eclipse `IJavaElement` hierarchy[4] used by the Eclipse Java structure views. The Mylar model is best conceptualized as an actively updated index over this program element hierarchy. Each element in the model is a lazily-updated proxy for an `IJavaElement`. In addition, each element in the model stores a float value that represents the interest in that element. The float values have an unspecified range, and views render interest-highlighting relative to that range. By convention elements with a negative interest value are considered uninteresting, and hidden when interest-based filtering is enabled. Each selection of a Java element made by the programmer contributes to the interest level (+1 by default), as does each keystroke made while editing the Java element (+0.1 by default). Each selection also has the effect of decaying the interest values of the other elements in the model (-0.1 by default). To reduce memory overhead elements with a low value (-10 by default) are purged from the model. The scaling parameters for each of the model update operations affect the stability of the model. The default values were hand tuned based on usage statistics from developing Mylar in bootstrap mode[5]. Automatic tuning of model parameters is discussed in Section 5.1.

The Mylar `UserListener` monitors the Eclipse Workbench selection and viewer services, is notified of every selection and keystroke, and resolves the program element corresponding to the event (Figure 4). The listener informs the `DoiModelManager`, which updates the model based on the selection and editing activity. The same information is passed to the `UsageStatisticsManager`, which maintains a table of usage statistics for tuning of the DOI function and for study purposes (Figure 3).

[4] The `org.eclipse.jdt.core.IJavaElement` hierarchy represents the containment hierarchy of all Java elements, starting with the project and ending with members. Storing of the `org.aspectj.asm.IProgramElement` nodes is also supported since, as of AJDT v1.1.12, AspectJ elements are not integrated into the `IJavaElement` hierarchy.

[5] All but the initial implementation of the model and test suite have been developed using the Mylar views.



```
DOI Model   Outline
RemoteException.class (137.59964
WSIFOperation_Jms.java (112.400
setPropertyValues (112.40016)
WSIFOperation_Jms (112.40016)
org.apache.wsif (91.90017)
WSIFService.java (91.90017)
RemoteException (75.60011)
WSIFDefaultCorrelationService (60
checkForTimeouts (60.500053)
org.apache.wsif.base (60.500053)
WSIFDefaultCorrelationService.jav
put (59.70004)
WSIFResponse.java (54.200096)
setUpBindingNamespaceURIs (43.7
setUpAddressNamespaceURIs (43.
get (39.400047)
WSIFServiceImpl.java (33.400116)
WSIFInterceptorConfig.java (32.7(
WSIFServiceFactory.java (32.5001
WSIFConstants.java (32.200066)
PopulatedInstanceExtensionRegisti
```

```
Usage Statistics
edit ratio: 0.05
numKeystrokesJavaEditor: 54
numSecondsElapsed: 2318
numSelectionsJavaEditor: 45
numSelectionsJavaEditorAutoFold:
numSelectionsJavaOutline: 0
numSelectionsJavaOutlineMylar: 1
numSelectionsOther: 11
numSelectionsPathfinder: 0
numSelectionsPkgExplorer: 25
numSelectionsPkgExplorerMylar: 8
numSelectionsProblems: 1
numSelectionsProblemsMylar: 1
numSelectionsSearch: 3
numSelectionsSearchMylar: 0
startDate: 2004-09-20
startTime: 20:3:15
```

Figure 3: Mylar model and usage statistics views

The DOI model is kept in memory and is lazily written to disk in order to persist across Eclipse sessions. The memory footprint of the model is dominated by the Java elements that the model references, since Eclipse would normally reclaim memory used by these elements. However, the additional elements referred to by the model are by definition the ones that are used more frequently, so the relatively small space overhead of keeping the model elements in memory results in a small performance improvement, since the common elements do not need to be re-created. A relevant implementation detail of the model is the fact that it updates several indices on each modification. These indices include one that propagates the interest values of children to parents in order to speed-up rendering for filtering views, a list of highest-interest elements to speed-up auto-expansion of tree views, and a list of candidate elements that may be added to the DOI model with the next selection. The index maintenance is necessary to prevent costly computation that would need to be done when rendering the active Mylar views (Section 5.3) which are updated on each model modification (e.g., on every selection change in the editor).

4. VALIDATION OF THE MODEL

The goal of Mylar is to enable programmers to spend more time working on code than they spend navigating it. The larger a system, the more likely it is that a programmer will need to focus on a subset of the crosscutting code as part of a task. For this reason, we chose to do a diary study [13] on the use of Mylar by professional programmers who work on enterprise-scale systems. The programmers were asked to use the experimental Mylar tool and to provide daily qualitative reports of their experiences. We augmented the diary study format with quantitative measurement by recording the programmers' activity (Section 3.3).

Our study tested programmers working with plain Java code. We placed this constraint on the participants for two reasons. First, we wanted to test our encoding of interest values while ensuring that the programmers' tasks remained consistent with their daily work. We also wanted to validate the Mylar model on large systems. Since AOP is still early in the adoption phase for enterprise application development, we did not have access to programmers actively developing large production applications with AspectJ. We believe that the model requirements for the Java case are

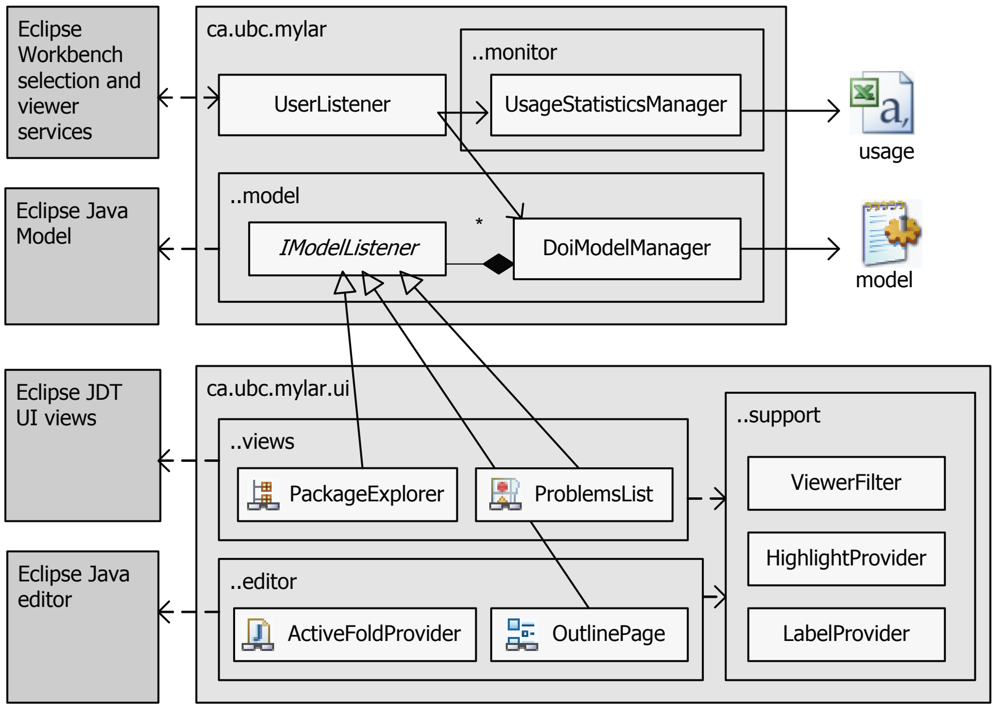


Figure 4: Overview of the Mylar architecture

similar to those for the AspectJ case, even though the views vary (Section 2).

4.1 Diary study format

The participants in our diary study were six senior IBM Toronto Lab programmers working in Eclipse on projects involving WebSphere[6], XDE[7] and Eclipse plugins. We also involved a summer intern for the purpose of having a more interruptible subject who could test any patches and releases made during the study. We did not include her in there results, which are limited to professional developers. For the duration of the study one of the authors was collocated with all but one participant. However, to minimize the time taken from the participants, support and interaction over the week was kept to a minimum and was provided through email. During the five day study, the programmers used a configuration of Eclipse that included the Mylar Package Explorer, Outline, and Problems List. Programmers were suggested to try, but not forced to use the Mylar views. To support our goal of producing an intuitive user interface that exposed DOI, without diverging too much from the feel of Java views, we provided no training on Mylar and required the programmers to read only a single page of documentation.

Before the week of the study we collected baseline data about the programmers' Eclipse usage, logging their edits and selections as they worked, and capturing summary data. A sample snapshot of this data, captured by the Mylar Monitor (Section 3.3), is visible in the Usage Statistics view (Figure 3). The total number of hours that Eclipse was active on the programmers' machines was 25.5 hours for the 3 days of baseline monitoring.

The following week we ran the diary study; during the study we logged 57.0 hours of Eclipse usage over 5 days. At the end of each day we asked the participants to send their usage data and answers to a one page survey of their day's experiences. At the end of the week we conducted half-hour wrap-up interviews with each of the programmers.

4.2 Results

Before the study, subjects were informed about the Mylar tool and each was given a questionnaire asking about their experiences using Eclipse. The problems cited include a dislike of the way in which editors and files are handled, and overpopulation of tree views such as the Package Explorer.

"I wish the content in the navigator view and the package explorer view can be more condensed."

"User has to filter out unwanted files explicitly… I use package explorer mainly for looking at what files or Java classes I have. Sometimes there are files I am not interested in."

"I don't like managing the expansion state of Trees"

The first two results reported below are quantitative and derived from the Mylar Monitor usage data. The latter two are qualitative results synthesized from the daily diary and wrap-up interview responses.

4.2.1 Usage statistics

Programmers used the Mylar views more than the plain Eclipse views. The view they used most was the Mylar Package Explorer, which is consistent with the baseline ratio of view usage. The reason for the Outline's lower use is that the most active programmer, who contributed to 80% of that statistic, had not read the page of documentation and had not enabled the Mylar Outline view. Enabling this view was the only configuration required of the study subjects. Once enabled, she used the Mylar Outline almost exclusively. The complete usage statistics for the week using Mylar are in Figure 5. Note that the "editor" selections are the result of following references and links in the Java editor and are independent of the Mylar views. The "other" selections are dominated by use of the Type Hierarchy view.

4.2.2 Edit ratio

We defined the *edit ratio* as the number of keystrokes in the editor over the number of structured selections made in the editor and views (i.e., the total across the columns in Figure 5). We hypothesized that if the elements relevant to a task are visible and highlighted in the IDE views, programmers should spend less time trying to find those elements, and more time working on their task. The improvement we observed in edit ratio between the baseline usage data and the Mylar usage data is encouraging. Finding a meaningful statistic of this ratio was challenging not

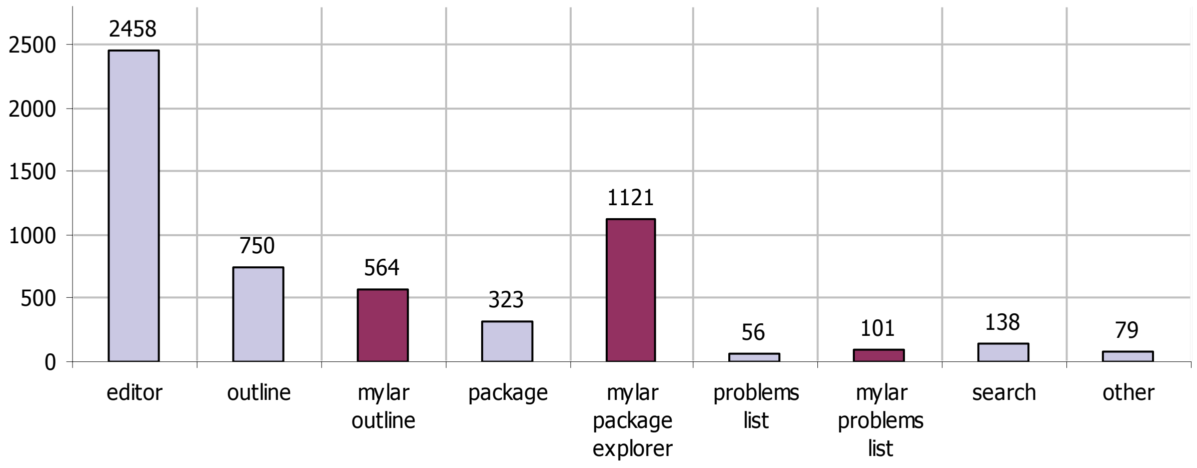


Figure 5: Number of selection in plain Eclipse views vs. Mylar views

[6] http://www.ibm.com/software/info1/websphere

[7] http://www.ibm.com/software/awdtools/developer/rosexde

only due to the small sample size, but also to the short duration of the study. From the daily diary responses we learned that several programmers switched tasks between the baseline week and the study week (e.g., one stopped developing code and moved to a debugging stage). A similar factor was a change in the amount of time of active Eclipse development between the baseline and Mylar week (e.g., two programmers spent less than ½ hour in Eclipse during the baseline week). So we feel that the average edit ratio improvement across subjects of 15% is promising, but overly noisy. However, the single most active programmer (she actively worked in Eclipse for 19 hours during the week and accounted for 40% of the activity across both weeks) reported that she worked on the same task both weeks. Her edit ratio improved by 49%.

During the wrap-up interview we asked the programmers if the significant increase in the edit ratio was consistent with their impressions. All of them agreed, stating that they did not need to navigate or search for elements as much as they did with the plain Eclipse views.

4.2.3 Model feedback

All of the programmers reported that the model accurately represented the context of their task. During the wrap-up we showed them the hidden DOI Model view (Figure 3) and asked how closely the ranking matched their work over the week. All reported that it closely represented the parts of the system on which they had worked. We had built it for internal debugging and inspection purposes. But some were surprised by the accuracy of this view, and expressed interest in using it for their programming activity.

Most of the programmers stated that the transparency of the model was important to them (e.g., they knew that clicking on a method in the editor would make it appear in the filtered Outline view). The key shortcoming reported was the inability of the model to understand task switching (e.g., to start on a new bug report they would have to clear the model, even though that model may be needed again).

Two programmers asked for a "silent activity" mode in which usage would not be recorded when the current task diverged momentarily. They wanted Mylar to better support debugging activity which overpopulated the model (e.g., single-stepping caused too many irrelevant elements to become interesting). Overpopulation was also reported when code not relevant to the current task was accidentally explored, and the UI for manual interest reduction was not intuitive enough for some of the programmers. From our own early use we knew that the stability of the DOI function could be a problem, causing the DOI of interesting elements to fall too quickly. As a result, we decided on an overly conservative tuning that led to the overpopulation.

4.2.4 View feedback

Although all of the programmers liked what the views exposed, there was a mixed response to the highlighting scheme. While three programmers liked it and one felt neutral, two programmers found it visually loud and disliked the intensity of the color added to the views. For the purpose of consistency, the programmers could not change the highlighting scheme to use a different color or an icon annotations instead of color range. The Mylar Package Explorer was the most liked view. Programmers found the automatic filtering and auto-expansion mode useful because it drastically reduced the amount of scrolling and inspection they needed to do. Some liked the auto-expansion idea but found that the UI interaction model differed too much from a typical tree view (users could not collapse nodes containing children of high interest since the collapse function was not mapped to an interest operation on the model). Against our intuitions most of the programmers were not interested in seeing the annotation of how many elements were filtered, and explained that they were used to elements missing from the Package Explorer since they regularly used other filtering mechanisms.

The Mylar Problems List was also well liked, which was surprising because in the baseline study only five Problems List selections were made over all of the programmers. The subjects reported that the interest highlighting helped with the overpopulation of the list, and some asked for interest-based sorting of that list.

The persistence of the model was well-liked by all the programmers—when they restarted Eclipse after a long break the last working context was retained. The most commonly asked for feature was a Mylar version of the Type Hierarchy view (discussed in Section 5.3.1) and the Content Assist popup view. All of the programmers expressed interest in using future releases of Mylar[8].

5. DISCUSSION

5.1 Expanding the Mylar Model

A single Mylar model exists per Eclipse workspace. The programmers in our study indicated the desire to extend Mylar's model to capture the multiple and possibly disjoint tasks that they often have active in a single workspace. To support multiple tasks, we plan to extend the Mylar to associate a separate DOI for each task. However, the study questionnaires pointed out that a desirable property of the model is its close correspondence to the programmer's overall familiarity with a system. For example, commonly used APIs tend to be retained by the model, making it easy to access information that was hard to find initially. To preserve the property of representing the programmer's memory of the system, we plan to percolate the task-specific interest values to global workspace values. The Mylar views will need to be extended to differentiate between task-specific and global interest.

We also believe that there is utility in extending the lifetime of a Mylar model to enable a programmer to reuse a DOI model when working on a similar task in the future. Often, tasks are related to bug reports. We plan on extending a Bugzilla plugin developed in our research group to allow programmers to manage multiple DOI models along with bug reports by associating each model with a report. The DOI model can then be externalized as an attachment to the Bugzilla report, and can be reloaded into the workspace. When the report is revisited, possibly by a different programmer, the key elements related to the report will be explicit. To indicate

[8] Since we wanted to focus development effort on incorporating study results and not supporting the study release, we asked the subjects to uninstall the tool at the end of the week. The following week we were forwarded an email stating that one of the programmers found the tool too useful to uninstall, and continued to use it.

the elements that changed with the task we plan to extend the DOI encoding to preserve the parameters such as the number of selections and edits.

In addition to capturing the programming task, we need the Mylar Monitor to understand the programming mode. This was evident from a study participant's suggestion that debugging activity overpopulated the model, since single stepping caused too many elements to become automatically selected (Section 4.2). Tuning interest increase and decay parameters based on the programming mode could help improve the stability of the model. In addition, we plan on modeling the interest contribution of each view. We already make a distinction in contributions between the structure views and the editor (Section 3.3). The stack trace selections were not made intentionally by the user, but indirectly by the single-stepping mechanism. Capturing the intentionality of the selection could also enable other kinds of automated contributions to the interest model (e.g., the profiling information of all advice executed by the Java Virtual Machine).

The current Mylar model is not as helpful at the beginning of a task, where there is little encoded DOI context. To facilitate working with unfamiliar code we plan to extend the model to support predicted interest. Mechanisms similar to the automated search facility described in Section 5.3 could then contribute a predicted interest level to elements that do not yet have an encoded interest.

One programmer stated that she wanted the Mylar model to extend to XML files and elements. Since the DOI model makes few assumptions about the elements it captures, non-Java elements can be represented in the model. Support for XML could help with managing enterprise application descriptors and aspects declared in XML languages (e.g., Spring AOP[9]).

5.2 Improving DOI visualization

Mylar's default DOI visualization uses background coloring to indicate the relative interest level of each element. Although the study subjects liked the effect of the visualization, two said that they prefer their IDE to be "less colorful". In addition to supporting configurable color schemes, we plan to provide icons that indicate interest value.

We are also exploring DOI-specific visualizations that focus on showing structurally similar elements arranged according to their interest level. For example, the prototype view in Figure 6 is intended to replace Eclipse's editor tabs and editor list with a visually stable rendering of files arranged according to their interest, and ordered in columns corresponding to packages. Another visualization of the DOI model that we are exploring is the UML[10] static class diagram notation, which can be rendered similarly, with additionally display the class associations.

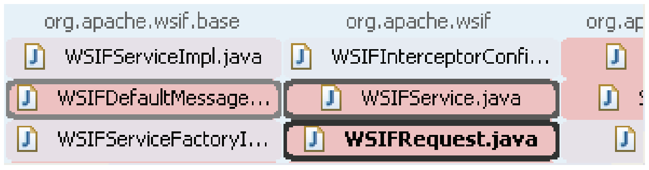


Figure 6: 2D layout of high-interest files

[9] http://www.springframework.org

[10] http://www.omg.org/uml

5.3 Active Views

For the core Java and AJDT views described in Section 3.2 the Mylar model need not capture any structural relationships between the elements of interest. But when the programmer's task is concerned with the inheritance or crosscutting, the relationships between elements of interest become important.

The active Mylar views differ from existing Java and AspectJ views by eagerly presenting elements of interest. For example, Eclipse's Type Hierarchy view is only populated when the user asks to see the inheritance structure for the selected element. In contrast, Mylar's Active Type Hierarchy is eagerly populated by all elements of high-interest. Active views allow a programmer to be continually aware of how the elements that are a part the task context fit into the overall structure of the system. For this purpose the Mylar model supports what we call *implicit search*—a continuous structured search on elements with the highest interest. The DOI model reduces the search scope to the extent that the implicit searches can be run as low-priority background threads that do not break the programmer's workflow.

Another benefit of the Active Mylar views is similar to what was reported by study subjects' use of the Mylar Package Explorer: the DOI model makes the structure views visually stable and provides guaranteed visibility [12] for the elements of high interest. This property encourages the use of visual memory for quickly finding elements in the Mylar views. Since the study we have implemented. but not yet validated, two Active View prototypes—one for crosscutting structure and the other for inheritance.

5.3.1 Active Type Hierarchy

Mylar's Active Type Hierarchy continually invokes Java structure searches on the program elements of highest interest, and displays the results in a view based on the Eclipse Type Hierarchy. When a programmer changes the body of a method of high interest, they immediately see all of the methods overriding the one they are editing. In addition, the programmer sees the elements of high interest within the context of the entire system's inheritance structure. Whereas asking for the Type Hierarchy of library classes often yields an over populated view (Figure 1) because it includes the entire scope of the workspace, the Active Type Hierarchy continually searches the inheritance structure of the current DOI context. Any change in the DOI model invokes a search of related suptertypes, subtypes, implementers, and overriders. By including only the elements of high interest it presents a concise summary of the interesting inheritance structure even when working on systems with large library dependencies and deep type hierarchies. The corresponding visualization is similar to the Active Pointcut Navigator (Figure 2, #4).

5.3.2 Active Pointcut Navigator

The Mylar DOI model enables a new kind of AspectJ view capable of showing all of the crosscutting related to the current context. The Active Pointcut Navigator (Figure 2, #4) shows the programmer the effect of all the pointcut declarations in the system. It exploits the naming of pointcuts to organize them in a tree view showing the named pointcuts as nodes and any pointcuts that refer to them as children. At the leaves of the tree are advice and the corresponding affected program elements. This view makes the effects of changing a pointcut explicit.

When a pointcut used by several other pointcuts and advice is changed, the effects are immediately visible lower down in the Pointcut Navigator tree, which shows the changed set of affected elements. Note that since pointcuts are not restricted in what other pointcuts they use, the tree structure can represent a graph. This is not the common case, and repeated nodes are annotated.

The Active Pointcut Navigator view shows the crosscutting structure relevant to the current context. Similar to the Active Type Hierarchy, it accomplishes this goal by querying the crosscutting structure of the high-interest elements and populating the tree view with those elements, in the context of the pointcut usage hierarchy. For example, if the high interest elements are five method declarations and one abstract pointcut, the view is populated with the concrete pointcut, advice that uses it, and any advice that affect the five method declarations of interest. Updates to the DOI model cause this active view to update by means of the implicit search, so the programmer is continually made aware of how the aspects in the system affect elements corresponding to high-interest join points.

6. RELATED WORK

Several research tools provide facilities to help the programmer explicitly declare the elements related to a task. FEAT allows the programmer to create views of structurally related elements by explicitly adding them to a Concern Graph [15]. JQuery can capture the elements in Java structure queries whose results persist in a view [8]. The Concern Manipulation Environment provides similar structure search features in its query engine [6]. Whereas these approaches use a new view to show the program elements related to the task-at-hand, Virtual Source Files provide similar functionality but use a source file metaphor to group the code itself instead of displaying links to the code [9]. These approaches all place burden on the programmer with declaring the task-specific program elements and queries that identify those elements. In contrast, Mylar captures these program elements implicitly, reducing the programmer's effort. Mylar could also benefit from incorporating query and concern inference results, and correlate them to interest-increasing operations on the DOI model.

Many IDE tools have monitored the programmer's context to present related program elements, starting perhaps with Interlip's Masterscope [16], which surfaced elements structurally related to the current one. A key difference offered by the Mylar model is that it provides a context beyond that of the current selection.

The filtering and folding support is related to the way in which the Jaba tool [2] elides code regions by means of the DOI model used in Fisheye views [4]. In contrast, Mylar's use of DOI is not based on tree structure navigation but on programming activity. Mylar's recording of programmer activity is similar to that done in the document processing domain by the Edit and Read Wear visualization tool which marks the frequently accessed places in a text document [7]. Instead of capturing the unstructured places that a document is edited, Mylar's interest model encodes the relevance of structured program elements. A programmer's context can also be inferred from analyzing the structural navigation paths, as suggested by the automatic extraction of concerns [14]. Mylar does not model navigation paths and instead associates editing and navigation activity with program elements alone. Representing navigation paths along with their DOI could be useful extension to the model.

7. CONCLUSION

The Mylar tool focuses the elements visible in IDE views on the context of a programmer's task. This helps programmers spend more time working with the multiple places in the code relevant to their task and less time looking for those places.

Mylar demonstrates that a straightforward encoding of a degree-of-interest model has value to programmers working on large systems and can be surfaced predictably in IDE views, that exposing an interest model makes programmers more productive by helping them focus on their task (i.e., the programmers edit more than they navigate), and that the interest model can be used to actively show views of the crosscutting and inheritance structure related to the task-at-hand.

8. ACKNOWLEDGMENTS

This research was funded by an IBM CAS fellowship and by NSERC. We would like to thank Julie Waterhouse of IBM CAS for her invaluable help in setting up the study and coordinating participants, the IBM Toronto lab participants for their time and feedback, Tamara Munzner for providing input on visualization issues, and Joanna McGrenere for her input on study design. Gregor Kiczales, Adrian Colyer, Brian de Alwis, Wes Coelho, Andrew Eisenberg and Robert Hirschfeld also provided feedback on the paper and/or implementation.

9. REFERENCES

[1] Card, S. K. and D. Nation. *Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface*. Advanced Visual Interfaces Conference, Trento, Italy, 2002.

[2] Cockburn, A. *Supporting tailorable program visualisation through literate programming and fisheye views*. Information & Software Technology 43(13), 2001, 745-758.

[3] Findlater, L., McGrenere, J. *A comparison of static, adaptive, and adaptable menus*. In Proceedings of Computer-Interaction, 2004, 89-96.

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

[5] Gosling, J., Joy, B., and Steele, G., Bracha, G.: *The Java Language Specification*. Second Edition. Addison-Wesley, Reading, Massachusetts, 2000.

[6] Harrison, W., Ossher, H., Tarr, P., Kruskal, V. and Tip, F. *CAT: A Toolkit for Assembling Concerns*. Research Report RC22686, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, December, 2002.

[7] Hill, W. C., Hollan, J. D., Wroblewski, D., and McCandless, T. *Edit wear and read wear*. In Proceedings of the Conference on Human Factors and Computing Systems, 1992, 3-9.

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

[9] Janzen, D. and de Volder, K. *Programming With Crosscutting Effective Views*, In Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, Oslo, 2004, 197-222.

[10] Kiczales, G., et al. *An Overview of AspectJ*. In Proceedings of the European Conference on Object-Oriented Programming. Springer-Verlag, Budapest, 2001, 327–353.

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

[12] Munzner, T., Guimbretiere, F., Tasiran, S., Zhang, L. and Zhou, Y. *TreeJuxtaposer: Scalable Tree Comparison using Focus+Context with Guaranteed Visibility*. SIGGRAPH published as ACM Transactions on Graphics 22(3), 2003, 453-462.

[13] Rieman, J. *A field study of exploratory learning strategies*. ACM Transactions on Computer-Human Interaction, 3, 3, 1996, 189-218.

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

[15] Robillard, M. P., Murphy, G. C. *FEAT. A Tool for Locating, Describing, and Analyzing Concerns in Source Code*. In Proceedings of the International Conference on Software Engineering, 2003, 822-823.

[16] Teitelman, W. and Masinter, L. *The Interlisp programming environment*. IEEE Computer, vol. 14, 1981, 25-34.