Focusing knowledge work with task context

by

Mik Kersten

B.Sc. (Computer Science), University of British Columbia, 1999

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

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University of British Columbia

January 2007

© Mik Kersten, 2007

Abstract

By making information easy to browse and query, current software tools make it possible for knowledge workers to access vast amounts of information available in document repositories and on the web. However, when displaying dozens of web page search hits, hundreds of files and folders in a document hierarchy, or tens of thousands of lines of source code, these tools overload knowledge workers with information that is not relevant to the task-at-hand. The result is that knowledge workers waste time clicking, scrolling, and navigating to find the subset of information needed to complete a task. This problem is exacerbated by the fact that many knowledge workers constantly multi-task. With each task switch, they lose the context that they have built up in the browsing and query views. The combination of context loss and information overload has adverse effects on productivity because it requires knowledge workers to repeatedly locate the information that they need to complete a task. The larger the amount of information available and the more frequent the multi-tasking, the worse the problem becomes.

We propose to alleviate this problem by focusing the software applications a knowledge worker uses on the information relevant to the task-at-hand. We represent the information related to the task with a task context model in which the relevant elements and relations are weighted according to their frequency and recency of access. We define operations on task context to support tailoring the task context model to different kinds of knowledge work activities. We also describe task-focused user interface mechanisms that replace the structure-centric display of information with a task-centric one.

We validate the task context model with three field studies. Our preliminary feasibility study of six industry programmers tested a prototype implementation of the task context model and task-focused user interface for an integrated development environment. Our second study involved sixteen industry programmers using a production quality implementation of the task context model; these programmers experienced a statically significant increase in productivity when using task context. Our third field study tested a prototype implementation of the task context model for a file and web browsing application. The results of this study showed that task context generalizes beyond programming applications, reducing information overload and facilitating multi-tasking in a cross-section of knowledge work domains.

Contents

Abstract ii
Contents iii
List of Tables vii
List of Figures viii
Acknowledgements x
1. Introduction 1
1.1 Scenario 2
1.2 Thesis 5
1.3 Previous Approaches 7
1.3.1. Modularity Mechanisms 7
1.3.2. Annotations, Tagging, and Explicit Context Models 9
1.3.3. Implicit Context Models 9
1.3.4. Explicit Task Models 10
1.4 Approach 10
1.5 Example 12
1.6 Overview 15
2. Model & Operations 16
2.1 Encoding Interaction 18
2.1.1. Interaction Events 18
2.1.2. Element Identity 19
2.1.3. Event Classification 19
2.2 Task Context Construction 21
2.3 Task Context Operations 24
2.3.1. Composition 25
2.3.2. Slicing 26
2.3.3. Manipulation 27
2.3.4. Induction 28
2.4 Task Activity Context 29
2.5 Model Summary 30
3. Implementation 31
3.1 Architecture Overview 31
3.1.1. Bridges 34
3.1.2. Mapping to Interaction History 36

3.1.3. Interaction Monitoring 37
3.2 Integration 38
3.2.1. Task Context Projections 38
3.2.2. Task-Focused UI Mechanisms 39
3.2.3. Task Management 41
3.2.4. Interaction History Storage 43
3.3 Mylar IDE Implementation 43
3.3.1. Task-focused UI for Programming 44
3.3.2. Inducing Interest 47
3.3.3. Active Search 48
3.3.4. Context Test Suite 50
3.3.5. Context Change Sets 51
3.4 Mylar Browser Implementation 52
3.4.1. Navigation Support for Files and Web Resources 53
3.4.2. Capturing Interaction with Files 53
3.4.3. Capturing Interaction with Web Resources 54
3.5 Implementation Summary 55
4. Validation 56
4.1 Methodology 57
4.2 Study I: Programmer Feasibility Study 58
4.2.1. Subjects 58
4.2.2. Method and Study Framework 59
4.2.3. Results: Usage statistics 61
4.2.4. Results: Edit Ratio Change 62
4.2.5. Results: Model Feedback 63
4.2.6. Results: View Feedback 64
4.2.7. Threats 64
4.3 Study II: Programmer Field Study 65
4.3.1. Subjects 65
4.3.2. Method and Study Framework 67
4.3.3. Results: Analysis and Edit Ratio 68
4.3.4. Results: Qualitative analysis 69
4.3.5. Threats 71
4.4 Study III: Knowledge Worker Field Study 72
4.4.1. Subjects 73
4.4.2. Method and Study Framework 74

4.4.3. Results: Task Activity 74
4.4.4. Results: Task Context and File Structure 76
4.4.5. Results: Task Context Contents....78
4.4.6. Results: Task Context Model Accuracy 79
4.4.7. Results: Feedback....80
4.4.8. Threats 82
4.5 Summary 83
5. Discussion 84
5.1 Improving Task Support and Context Accuracy....84
5.1.1. Working on Multiple Tasks Concurrently....84
5.1.2. Related Tasks....84
5.1.3. Task Context Lifecycle....85
5.1.4. Tuning the Scaling Factors 85
5.2 Improving the Task-Focused UI 86
5.2.1. Decorating Elements to Indicate Interest Level....86
5.2.2. Exposing the Interest Thresholds 86
5.3 Collaborating with Task Context 87
5.4 Visualizing Task Context....90
5.4.1. Spring Graph of Task Context....92
5.4.2. Seesoft Visualization of Task Context 93
5.4.3. Interaction History Chart....94
5.5 Future Work 95
5.5.1. Transforming and displaying interaction history....95
5.5.2. Extending Support to Other Domains 96
5.5.3. Improving Interaction with Filtered Views 96
5.5.4. Unified Interaction, Context, and State Model....97
5.5.5. Cascading Contexts 97
5.5.6. Focusing the Workweek 98
6. Related Work 100
6.1 Mechanisms for Identifying Relevant Structure 100
6.1.1. Query and Concern Management Tools for Programmers....100
6.1.2. Search Engines and Collaborative Filtering 101
6.2 Using Interaction to Create a Context....102
6.2.1. Displaying Usage-Based Context....102
6.2.2. Ranking Search Results....103
6.2.3. Implicit Search and Usage-Based Scoping....103

6.2.4. Mining User Interaction ... 104
6.3 Task Management Tools ... 105
6.4 Information Focusing User Interfaces ... 106
6.4.1. Information Visualization ... 106
6.4.2. Software Visualization ... 107
7. Conclusion ... 108
Bibliography ... 110
Appendix A User Studies ... 116
Ethics Certifications ... 116
Study I Questionnaire ... 118

List of Tables

Table 2.1: DOI value intervals 17
Table 2.2: Task context operations overview 18
Table 2.3: Interaction event schema 19
Table 2.4: Interaction event classification schema 20
Table 2.5: Sample interaction history 21
Table 2.6: Illustrations of topological properties of the task context 24
Table 2.7: Composition operation 25
Table 2.8: Slicing operation 26
Table 2.9: Manipulation operation 27
Table 2.10: Induction operation 29
Table 3.1: Mylar frameworks 32
Table 3.2: Scaling factors used in Mylar implementation 37
Table 3.3: Thresholds used in the Mylar implementation 38
Table 3.4: Task context projection 39
Table 3.5: Task-focused UI mechanisms for exposing interest 40
Table 4.1: Subject comments related to information overload in Eclipse 58
Table 4.2: Demographics of the ninety-nine participants 65
Table 4.3: Field study data and percentage improvement 69
Table 4.4: Accepted subjects in knowledge worker study 73
Table 4.5: Task context activation 75
Table 4.6: Work items reported by subjects to be associated with tasks 75
Table 4.7: Task context content structure measurements 77
Table 4.8: Task context content size measurements 79
Table 4.9: Proportions of task context model slices (direct interest only) 80
Table 4.10: Feedback on how much information relevant to the task was displayed 80
Table 4.11: General feedback 81

List of Figures

Figure 1.1: Information overload in the Eclipse IDE 3
Figure 1.2: Information overload when browsing files and web pages 4
Figure 1.3: Number of files versus check-in transactions for Eclipse and Mozilla 8
Figure 1.4: Illustration of task context 11
Figure 1.5: Task context while working on T1 (above), and shortly after activating T2 (below) 13
Figure 1.6: Browsing with task context 15
Figure 2.1: Constructing and projecting a task context 17
Figure 2.2: Combining task context operations 27
Figure 2.3: Illustration of task activity context 29
Figure 3.1: Mylar architecture showing OSGi plug-ins and their dependencies 33
Figure 3.2: Mylar plug-in architecture and Java bridges 35
Figure 3.3: Task management facilities for activating and editing tasks 42
Figure 3.4: Decoration, filtering and ranking of Java elements 45
Figure 3.5: Decoration, filtering and ranking of various program artifacts 46
Figure 3.6: Automatic folding and content assist ranking in Java editor 47
Figure 3.7: Propagated interest of errors in the Package Explorer 48
Figure 3.8: Propagated interest of superclasses in the Active Hierarchy view 48
Figure 3.9: Degrees-of-separation for Active Search 49
Figure 3.10: Elements and relations with predicted interest in the Active Search view 50
Figure 3.11: Automatic context test suite 51
Figure 3.12: Automatic context change sets 52
Figure 3.13: Mylar Browser showing Task List and embedded Excel document 54
Figure 4.1: Ranking of elements, usage statistics 60
Figure 4.2: Mylar 0.1, used for IBM study 61
Figure 4.3: Mylar view vs. standard Eclipse view selections across subjects 62
Figure 5.1: Mockup of threshold control UI 87
Figure 5.2: Context sharing initiated from the Task List 88
Figure 5.3: Mockup of overlay showing overlap between two team members' contexts 90
Figure 5.4: Table layout showing interesting files 91
Figure 5.5: Task context tree view showing relations of elements 92
Figure 5.6: Force-directed layout of task context 93
Figure 5.7: Seesoft visualization of task context 94

Figure 5.8: Visualization of interaction history with decay ignored 95
Figure 5.9: Focusing the work week 98
Figure 6.1: Separation of interaction-specific rankings from structure-based rankings....................103

Acknowledgements

My deepest thanks go to my supervisor, Gail Murphy. Gail has fueled my ideas, directed my efforts, and taught me the value of research. Her incredible mix of unwavering encouragement, critical questioning, and insightful advice were key to the success of this work. Gail's clear principles and empirical approach to creating and validating new technologies will continue to guide me and provide an example that I hope to emulate throughout my career.

I also thank Gregor Kiczales and Tamara Munzner, whose questions and feedback have helped shape the directions of this work. I am grateful to the other members of my supervisory and examining committee who have generously contributed their time and expertise: Erich Gamma, David Poole, Yair Wand and Victoria Bellotti.

This effort would not have been possible without the advice and perspective of my wife, Alicia Kersten, who supported me throughout. I would also like to thank the rest of my family; our kitchen table arguments were my introduction to critical thinking. In particular, I thank my father, Gregory Kersten, who convinced me to do my doctorate, and who inspires me more than anyone I know.

I thank Julie Waterhouse of IBM for making the feasibility study possible and the IBM Centre for Advanced Studies and NSERC for supporting this work. Finally, I thank the Mylar user and developer community, whose feedback and efforts continue to help the technology grow.

Mik Kersten

The University of British Columbia, January 2007

1. Introduction

Knowledge workers develop and use information in the workplace [19]. The software tools that knowledge workers use to browse and query the information systems with which they work make the structure of those information systems explicit. As one example, file browsers enable the hierarchical navigation of directory structure. As another example, Web query tools often rank the hits they return using properties of the hyperlink structure. However, when used on very large information systems, these structure-centric views can overload the user with information because the relevant subset of information is often scattered across the structure. Whether browsing the hundreds of files and directories in a document repository for a few key files of interest or looking through dozens of web search hits for a particular page, users must manually sift, scroll, and click through deep hierarchies and long lists to find the subset of information relevant to the task-at-hand. This information overload problem is inherent to tools that display information based on structural properties instead of focusing on the subset of the information that is relevant to the user's task.

***Information overload problem**: Many knowledge work tasks cut across the structure of an information system. As a result, tools for browsing and querying system structure overload users with information that is not relevant to the task-at-hand.*

If knowledge workers always completed the current task before proceeding to work on another, perhaps they could commit all of the relevant information to their memory or manually maintain a listing of the artifacts of interest. However, one study found that a group of knowledge workers performed ten tasks a day on average, spending only eleven continuous minutes on any particular task before switching to another [26]. All too often, before completing the given task, a worker must switch to a higher priority task that requires immediate attention. There is a concrete cost to such task switches; with each switch, workers waste time repeatedly identifying the relevant information. This loss of context complicates multi-tasking, which is further exacerbated by the time it takes users to recreate their context manually in the presence of information overload.

***Context loss problem**: Browsing and query tools burden users with finding the artifacts relevant to the task-at-hand. This context is lost whenever a task switch occurs. When multi-tasking, users continually waste time recreating their context.*

1.1 Scenario

The larger the information system that needs to be accessed by the knowledge worker, the more onerous the information overload and context loss become. In the software development domain of knowledge work, the information systems of interest are the software applications and frameworks that programmers create and integrate. Many modern applications and frameworks consist of millions of lines of code[1]. Current integrated development environments (IDEs), such as Eclipse IDE[2] and Visual Studio[3], make these millions of lines of code instantly accessible to the programmer through sophisticated indexing and search facilities. However, when working on any particular task, such as adding a feature to an application, the programmer is only interested in a very small portion of the code on which he is building. He may try to use query tools to help identify the information relevant to the task, but as this information often cuts across the system structure, it is difficult to formulate adequate queries within a reasonable amount of time. Alternatively, he may try to tag parts of the structure with bookmarks or other annotation mechanisms, but constant tagging and searching is burdensome. Long lists of tags can also contribute to information overload when a new and higher priority task needs attention, since they may not be relevant to the new task.

Consider the concrete example 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)[4]. To complete this task, the programmer must examine the test cases, the classes that are failing to de-serialize, and the serialization policy employed in the system. Using the Eclipse IDE, 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. Figure 1.1 shows a snapshot of the Eclipse IDE after the programmer is part way through completing this task.

1: The Package Explorer view 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 manual scrolling through the tree.

2: In part thanks to how easy Eclipse makes navigating structural relations, the number of open editors can quickly bloat to several dozen when working on a moderately-sized task, making the editor list a poor

---

[1] The full set of Eclipse frameworks is 7M lines of code and the Windows Vista OS is 50M, http://en.wikipedia.org/wiki/Source_lines_of_code [verified 2006-10-02]

[2] http://eclipse.org [verified 2006-10-02]

[3] http://msdn.microsoft.com/vstudio [verified 2006-10-02]

[4] http://ws.apache.org/wsif (1,897 classes) [verified 2006-10-02]

representation of the files currently relevant to the task [14]. Although using the "Close all editors" command cleans up this list, it also discards the editors that are relevant to the task.

3: Using Eclipse's Java[5] Search to look for references to `Serializable` within the project has returned 144 items. There is no convenient way to search for only those elements related to the task of fixing the failing test cases. Instead, the search result list requires manual inspection to find elements of interest.

4: Even though the Outline view only shows the structure of the current file, it is overloaded with dozens of elements that are not relevant to the task.

5: The Type Hierarchy view shows all types in the project that extend `Serializable`, and contains thousands of elements that must be manually inspected to identify those relevant to the task.

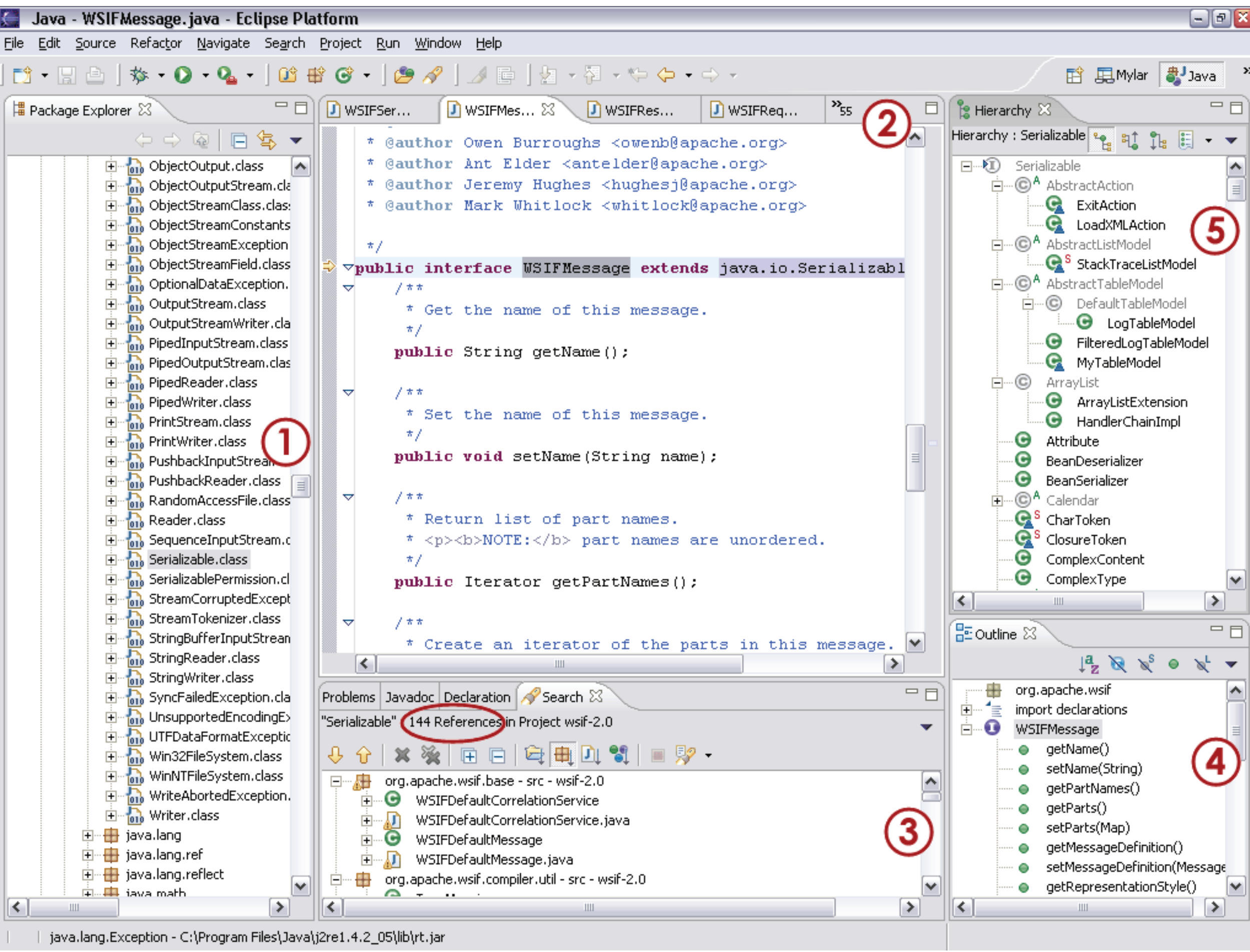

Figure 1.1: Information overload in the Eclipse IDE

Although these views and editors are overloaded, they still provide useful information to the programmer about the elements relevant to the task. However, as soon as the programmer starts working on a different

[5] http://java.sun.com/javase [verified 2006-10-02]

task involving other parts of the application, the context built up in these views is replaced with the results of new searches and new navigations.

This information overload problem is not limited to the Eclipse IDE. All current IDEs show aspects of whole-system program structure rather than helping the programmer focus on the program elements relevant to the task-at-hand. When working on a task that is 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, and inspect search results looking for those relevant to the task. In some cases, this can result in the programmer spending more time clicking than coding.

This problem is not limited to programmers, but pertains to any knowledge worker accessing large information systems, whether the structure of those systems is a well-defined file hierarchy or is a looser collection of hyperlinks and unstructured file contents. Consider the case of a knowledge worker returning to the task of getting advice from an immigration attorney (Figure 1.2). She knows that the name of the attorney is in a file that either she or a colleague created when last working on this task, but browsing the dozens of project directories and opening a few candidate files fails to identify it. Since a text search for "lawyer" would likely come up with hundreds of candidates, she proceeds to re-find the information by formulating a web query similar to the one she had made when first working on this task. After inspecting numerous web search results, she finally finds the page identifying the attorney. This scenario illustrates how effort is wasted in re-finding information when switching back to an earlier task.

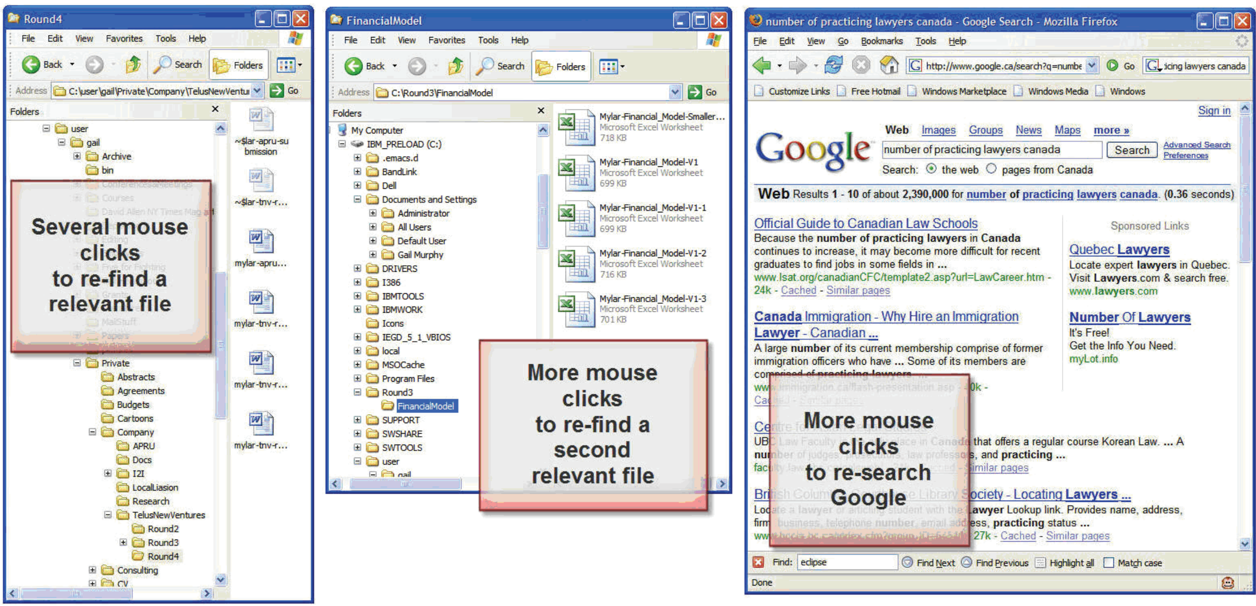

Figure 1.2: Information overload when browsing files and web pages

1.2 Thesis

The tools that knowledge workers use for working with information systems understand the structure of those systems and make that structure easy to navigate. For example, a Java IDE understands method call syntax and makes it navigable; an HTML browser understands hyperlink syntax and makes that navigable. Incorporating an understanding of information system structure into tools assists users by moving the burden of identifying the structural elements and their relations from the user to the tool.

The tools that knowledge workers currently use do little to assist them with understanding the content of an information system. Instead, knowledge workers rely on semantic memory to recall and identify the parts of the system relevant to a particular task. Semantic memory requires multiple exposures to each referent, with each exposure updating the memory [61]. Semantic memory is well suited to working with smaller information systems where knowledge workers can become familiar with a substantial portion of the system in a reasonable amount of time. However, when working with a large system, the limitations of semantic memory become apparent. It is not possible to remember everything when working with such a system and the information that is remembered can quickly become obsolete. The result is that knowledge workers spend an inordinate amount of time reminding themselves of the bits and pieces of an information system that they need to understand to complete a task.

There is another kind of memory at a knowledge worker's disposal. Episodic memory is referred to as a one-shot learning mechanism, in which only one exposure is required to remember the event, and is described as a map that ties together semantic memories [61]. Our goal is to increase the complexity of the information systems with which knowledge workers can work effectively. Our approach is to off-load semantic memory and to leverage episodic memory. A common form of episode for knowledge workers is the *tasks* that they perform, where a task is defined as "a usually assigned piece of work often to be finished within a certain time" [1]. Recalling tasks worked on previously uses episodic memory, making tasks inherently easier to recall than the semantics of a complex information system. Our approach makes tasks an explicit part of knowledge work. Instead of forcing knowledge workers to recall system structure, we provide a facility for recalling a task worked on previously and we then automatically present the system structure that was relevant to that task. By capturing the interaction the user has with the structure of an information system during each task episode, we bring together the parts of the information system that are relevant to the task. We call this subset of information relevant to a task the *task context* and follow the semantic memory concept of reinforcement [55] by automatically weighting the pieces of information according to frequency and recency of access. The only burden that we impose on knowledge workers is the need for them to indicate the episodes by defining and activating the tasks on which they work.

***Thesis**: A model of task context that automatically weights the relevance of system information to a task by monitoring interaction can focus a programmer's work and improve productivity. This task context model is robust to both structured and semi-structured information, and thus applies to other kinds of knowledge work.*

We present a model and mechanism for creating a *task context*, which captures and persists the elements of information and relations between elements of information that are relevant to a task, and weights the relevance of each according to the frequency and recency of access. We create a task context by monitoring a user's activity and extracting the structure of the underlying information system's elements and relations. Operations on task context integrate the model with knowledge work applications, providing features such as automatic search, automatic change management and context display. Together, the model and operations enable an application to present only the information relevant to the task-at-hand. This focus reduces information overload and prevents context loss when multi-tasking.

To test the task context model, we conducted field studies with professional programmers and other kinds of professional knowledge workers. Since the work items involved in programming are well-defined and broad, including adding code for features and editing code to fix bugs, we were able to define a measure of programmer productivity called edit ratio. We have shown with statistical significance that the edit ratio of sixteen programmers in our field study improved with the use of our task context model. Our field study involving knowledge workers was qualitative. For eight subjects representing a cross-section of professions ranging from administrative assistants to a CTO, we found that task context supports the information artifacts with which they work and that the frequency and recency-based weighting can reduce information overload in a cross-section of knowledge work domains.

The key contribution of this thesis is a generic task context model that represents interaction with any structured or semi-structured information, where the weights in the model correspond to the frequency and recency of a user's interaction with the information. We demonstrate that this weighting reduces information overload and that capturing context per-task reduces context loss when multi-tasking for both programmers [39] [41] and other kinds of knowledge workers [40]. Our secondary contributions are a realization of task context collaboration facilities [38] and a study framework suitable for monitoring productivity and tool usage in the field [52].

We have deployed our implementation of task context for Eclipse as an open source framework called Mylar[6], a programming tool called Mylar IDE, and a prototype file and web browsing tool called Mylar

[6] "Mylar" is a) an aluminized film used to avoid blindness when staring at a solar eclipse b) a focused user interface used to avoid information blindness when staring at Eclipse. The latter is hosted as an open source project led by the author http://eclipse.org/mylar [verified 2006-10-02]

Browser. We estimate that tens of thousands of programmers[7] currently use the Mylar IDE tool. The Mylar framework is also being used to support task context in other domains[8].

1.3 Previous Approaches

When working with a small information system, whether it is an example program or a small collection of web pages, information overload is rarely a problem because the information relevant to a task can be recalled. As the size of information systems has grown, several approaches have been introduced to help knowledge workers manage the increase in complexity. Each approach supports the capture of a subset of the system. The more a subset lines up with the structure a user needs to know about to perform a task, the more effective the mechanism is at helping the user work on the system. In this section, we describe approaches that apply to both programmers and other kinds of knowledge workers. When a distinction is needed, we identify programmers explicitly. We use the term system to refer to both software systems and other kinds of information systems, and specify the scenarios where a distinction matters.

1.3.1. Modularity Mechanisms

The better the modularity of a system, the more localized code changes become [3]. By enabling encapsulation and polymorphism, object-oriented programming (OOP) [13] helped localize many changes to one or a small number of places within the type hierarchy of a software system. As software systems continued to grow, aspect-oriented programming (AOP) mechanisms made it possible to encapsulate parts of the system that crosscut the object hierarchy [42]. Although modularity approaches have increased the system complexity that can be managed by a programmer, they have not solved the problem of information overload. They assume that a programmer will often be able to find a desired piece of the system by traversing the modular structure and that modifications will often fit within the modular structure so that once the point of interest is identified, it will be relatively easy to perform the desired modification.

We have observed two problems with these assumptions. First, many modifications to a system are not limited to one module. For example, we found that over 90% of the changes committed to the object-

[7] Each monthly release of Mylar IDE is installed by thousands of users from the main http://eclipse.org/mylar install site, and is also redistributed by commercial vendors. Links to vendors, user blogs, as well as articles citing productivity improvements are at: http://eclipse.org/mylar/archive.php [verified 2006-12-12]

[8] The CHISEL group at the University of Victoria uses Mylar's task context model for adaptive ontology visualizations in a Swing-based RDF/OWL browser: http://www.thechiselgroup.org/diamond [verified 2006-10-02]

oriented Eclipse and Mozilla[9] source repositories, over a period of one year, involved changes to more than one file [51] (Figure 1.3). We then randomly selected twenty changes from Eclipse and found that 25% of these transactions involved significantly non-local changes. Second, even when the actual changes related to a modification are within some form of module, say one Java package, a programmer often needs to know how this module works within the system, requiring him to access many other modules and understand their interconnections [58]. While AOP can help by improving the system's modularity, we have observed that there will still be tasks that are not localized in a single aspect, for example involving multiple classes, aspects, and extensible markup language (XML) files [39]. As such, modularity approaches alone are not sufficient for defining and managing the subset of information relevant to the various tasks on which a programmer works. The result is that a programmer must spend a substantial amount of time navigating around a system's modularity to identify the relevant information.

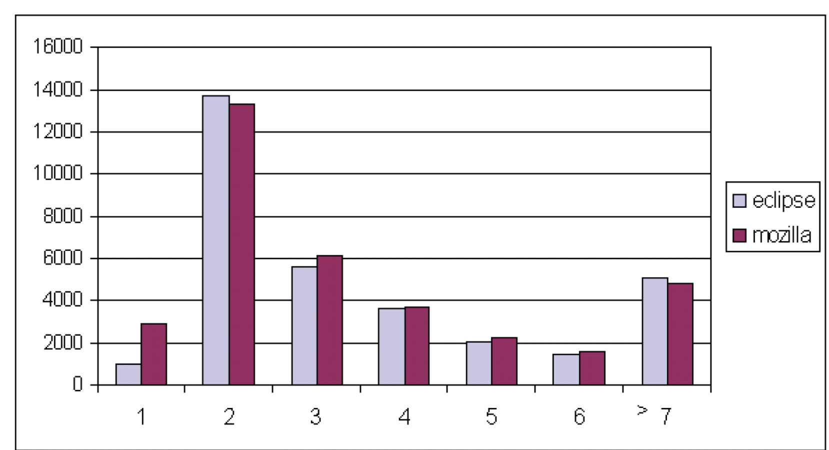

Figure 1.3: Number of files versus check-in transactions for Eclipse and Mozilla

IDE support for modularity mechanisms makes the structure of the modularity easily navigable by a programmer. Early versions of such tools made it possible to follow hyperlink-style references to other parts of the program. Current IDEs have augmented this support, providing views that make the entire object-oriented and aspect-oriented structure of the system navigable through structure views. For example, all subtypes of a given type may be found using a type hierarchy tree view. Another common mechanism for exposing modular structure is query and cross-reference support. For example, OOP tools commonly support a query to show all methods that could override the selected method. Similarly AOP tools display cross-references from a method to all of the advice that could apply to it [35].

Structure views and queries have made it possible to navigate the structure of a software system. As shown in Figure 1.1, these structure views typically show thousands of elements, only a small number of which are relevant to the task-at-hand. Similarly, query results often have hundreds of matches. The result

---

[9] http://mozilla.org [verified 2006-10-02]

is that programmers can spend more time scrolling and navigating then programming. Knowledge workers working with file and web pages face a similar situation when using recommender systems and search engines that provide heuristics for ordering or filtering results of searches. Heuristic rankings based on structural information, such as the link structure between documents [8], are ignorant of the user's task and thus include elements irrelevant to the task. Although recent research in search engines has attempted to address this limitation by using personalization data in rankings, these rankings are biased by all of a user's past activity and are not focused on her current task (Section 6.2.2).

1.3.2. Annotations, Tagging, and Explicit Context Models

One way to describe the additional properties of an information system is to use metadata. The metadata facilities exposed by current development environments for software systems vary in how closely related the data is to the program code and in the amount of structure present in the resulting metadata. For example, the Java and .NET[10] platforms' textual annotation mechanisms allow additional properties to be declared on program elements. The subset of the software system defined by annotations that match a particular property can then be operated on by annotation processing tools and exposed in the IDE's structure views (e.g., to show all methods marked @Transactional). Mechanisms for 'tagging' or 'bookmarking' elements through the IDE's editor and views can be used in a similar way to annotations to mark the elements that should form a particular subset of interest, which can then be viewed. Eclipse's support for working sets is a good example of how such a subset can be integrated into an IDE. Entire system components can be tagged as belonging to a particular working set. Views that show the hierarchical structure of the system can then group the system by working sets.

Annotation and tagging mechanisms do provide a way to identify relevant elements. However, they are not sufficient for capturing task context. The key problem is that they require manual work from the programmer to do the tagging. A second problem is that when used more heavily, for example with support for collaborative tagging [63], or as manifest by the behavior of Eclipse's automatically reported source code comments matching a particular pattern (e.g., "// TODO: fix"), views that report these tags become overloaded when used on large systems.

1.3.3. Implicit Context Models

An alternative to having the user manually create a context is to use interaction with the tool to define the context. The simplest examples are tools that use the currently selected element to show related elements, originating with Interlisp's Masterscope [67]. The Edit & Read Wear [31] document editing tool

---

[10] http://microsoft.com/net [verified 2006-10-02]

formalized this concept in the document processing domain by highlighting selection patterns across a document. As a programmer works in an IDE, the files that have been opened, often visible through the editor tabs similar to those in tabbed browsers[11], also reflect a context created from the programmer's interaction. Such implicitly determined models can be used to focus the user interface to show only the subset of information that is in the context. The key benefit of an implicit approach is that it removes the burden of creating the context from the user. However, considering the amount of interaction that a knowledge worker produces in a typical week [30], and the fact that it is typical for them to work on many tasks in a single day [26], a simple model of this form does not scale to resolving the information overload that occurs on long-running tasks and to the context loss that results from task switches [39].

1.3.4. Explicit Task Models

A task corresponds to a unit of work (Section 1.2). For a programmer, a typical task is a bug to fix or a feature to add. For another kind of knowledge worker, such as a lawyer, a typical task is completing a contract or drafting up a will. Approaches that make a representation of task explicit can provide a natural scoping mechanism to address some of the problems with the approaches described above. For example, some tools allow files to be attached to tasks, such as the Bugzilla bug tracker used by programmers[12]. However, this approach places the same burden on the user as explicit tagging. To address this burden, the TaskTracer tool [18] provides an alternative by combining explicit tasks with an implicit per-task tagging of resources, such as the files opened when working on a task. However, TaskTracer adds information to a task in a monotonically increasing fashion. In terms of reducing information overload, this approach assumes the information accessed for a task is well-scoped, which may not be the case for long-running or recurring tasks. It also assumes that a user will not make any missteps in deciding what information is relevant to the task. If the user does accidentally select the wrong resource, or if a resource loses relevance over time, this approach burdens the user with manually indicating which resources are no longer relevant to the task.

1.4 Approach

Our approach involves monitoring the user's activity with a software application in order to create an *interaction history* for a particular task. An interaction history is comprised of a sequence of interaction events, each of which captures a user's interaction with the specific elements and relations in an information system. The monitoring facility needs to report the elements and relations that are the target

---

[11] Browsers, such as Mozilla, provide tabs that allow one window to nest multiple pages.

[12] http://bugzilla.mozilla.org [verified 2006-10-02]

of the interaction, as well as the kind of interaction. We focus on capturing interaction events corresponding to the user's selections, edits, and commands. To support associating interaction histories with tasks, we also require a mechanism for determining task boundaries.

Given an interaction history for a task, we transform the sequence of interaction events into a task context, which weights each element according to its relevance to the task. Our task context model uses a degree-of-interest (DOI) [45] weighting for each element and each relation based on frequency and recency of access. Different kinds of events can correspond to different weightings. Our goals are to make the model predictable to the user and to ensure that the DOI values capture the relative relevance of elements and relations over the lifecycle of a task. The frequency portion of the weighting ensures that elements and relations accessed most have the highest interest weighting, while the recency weighting ensures that the interest of elements not accessed recently decays.

To integrate task context with the various activities that a user performs when working with an application, we have defined operations on task context. The slicing operation can gather a subset of the context, such as the elements above a particular DOI threshold. The composition operation makes it possible to combine multiple task contexts when an activity involves interaction with multiple tasks. We also provide an induction operation for propagating interaction events to structurally related elements; this adds the structurally related elements to the task context even though the user has not interacted with them directly. For example, when a user selects a file, an induction operation can propagate interest to each of the folders in the containment hierarchy of that file.

Operations such as induction need to understand the structure of the underlying information system. Each information system can be composed of one or more kinds of domain structure, such as a Java program and a file hierarchy. We define bridge mechanisms that map between domain structure and interaction events. The more detail the bridge understands of the domain structure, the higher the fidelity of the task context model. For example, if an application supports monitoring interaction with elements within a file, such as the classes and methods in a Java file, the task context model can weigh the relevance of that structure, rather than just weighing interactions at the level of the file and the folder hierarchy containing the Java class.

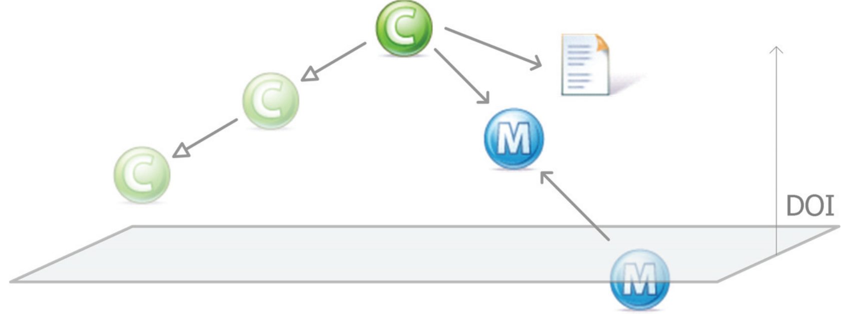

Figure 1.4: Illustration of task context

Figure 1.4 illustrates a task context as a graph in which the heights of the elements represent their DOI weightings. The Java class node at the top of the diagram is most interesting because the programmer has selected and edited it the most. Along the left of the figure, we see that interest has propagated to two superclasses that the user has not yet selected (indicated in the figure with lower saturation). We also see that a method in the model has decayed in interest to the point that it is below the DOI threshold depicted by the gray plane.

To focus an application on a particular task, the task context model needs to be displayed. We call the mechanisms that display the task context model the *task-focused user interface (UI)*. We have defined several task-focused UI mechanisms that support the display of task context in existing applications, including DOI filtering, ranking, decoration, expansion management, and application window management (Section 3.2.2). As one example, a structure view that normally shows the entire system structure can be focused to show only the elements within the task context that are above a particular DOI threshold. To display task context in an application's existing structure views, the application needs to support one or more of these mechanisms. For example, if a file browser has a mechanism to filter elements prior to display, this mechanism may be extended to support filtering based on DOI values.

1.5 Example

In this section, we demonstrate how explicit task context and task-focused UI mechanisms address the information overload problem, using as an example the Mylar code base, which contains over a thousand Java classes. Consider the scenario of a programmer working on a similar task to that presented earlier: refactoring a policy defined by a class, which affects uses of that class in the system. However, this time the programmer uses a Task List to indicate the task on which he is currently working (via the "radio button" indicator visible in Figure 1.5-1 to the left of the task of interest):

T1: Refactor ResourceStructureBridge

This task involves identifying, inspecting and changing all of the clients of `ResourceStructureBridge`. Activating the task causes the task context model to track the parts of the system artifacts—the program elements and relations—that are accessed while the programmer works on this task.

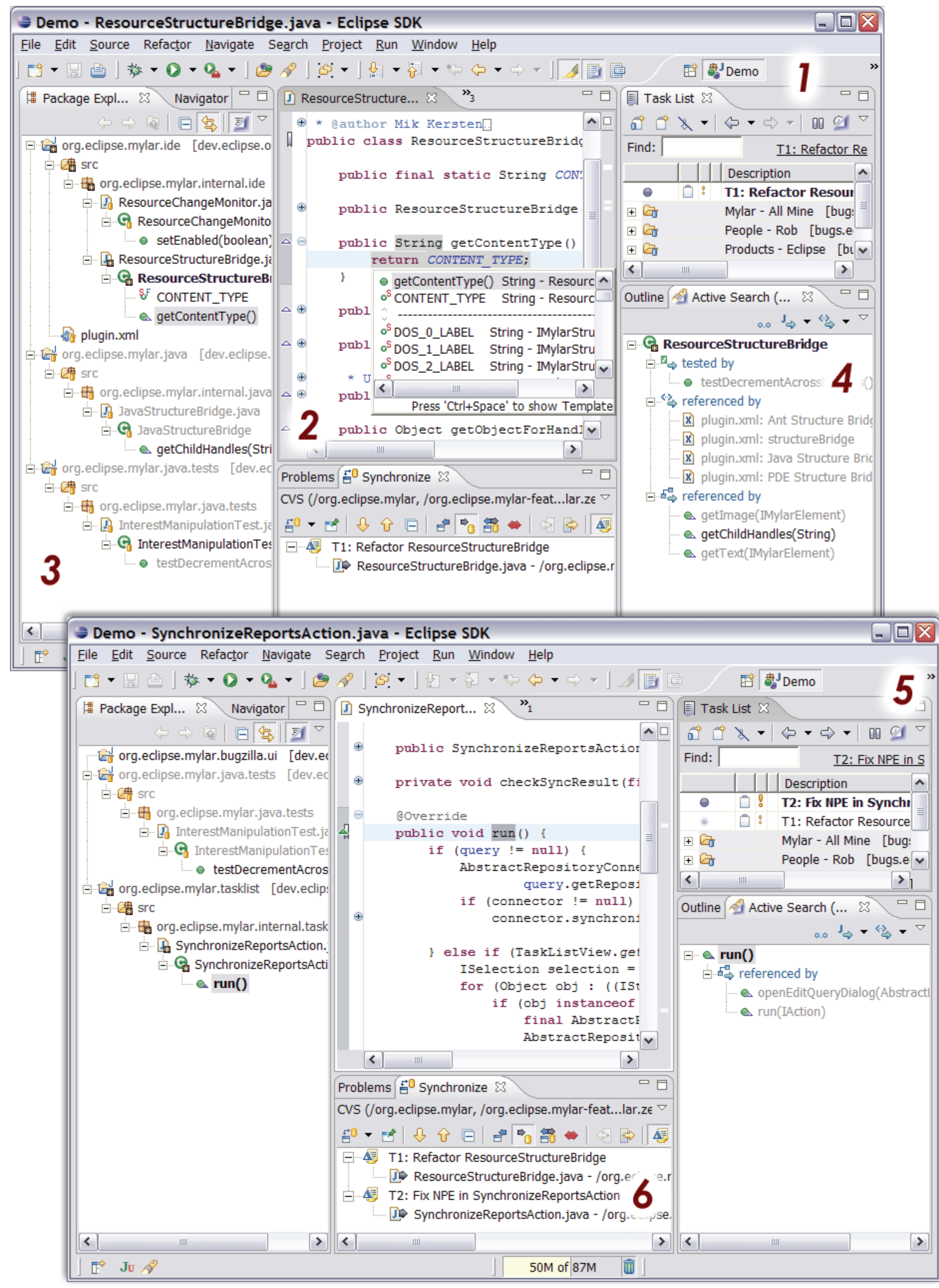

Figure 1.5: Task context while working on T1 (above), and shortly after activating T2 (below)

Mylar displays the task context within existing Eclipse IDE structure views in order to focus the programmer on the task-at-hand. The source code editor folds away all uninteresting elements and ranks content assist matches, visible in the popup window, to indicate those that are in the task context (Figure 1.5-2). Even though the code base contains over one thousand classes and numerous other kinds of artifacts, only the artifacts determined to be relevant to the current task context are visible in the Package Explorer view of the system (Figure 1.5-3). Structurally related elements that have not yet been interacted with, but have predicted interest, show in the Active Search view (Figure 1.5-4). When working with task context and the task-focused UI, the entire IDE becomes focused on the task instead of showing all of the parts of the system's structure, as was visible in Figure 1.1. Since the task context is actively trimmed to show only the most relevant elements, the programmer will rarely see a scrollbar in many of the task-focused views, reducing information overload in those views. While working in this focused way on T1, a new high priority bug is assigned to the programmer and must be attended to immediately.

T2: Fix NPE in SynchronizeReportsAction.

Using Mylar's Task List view, the programmer activates the second task (Figure 1.5-5), causing the context of the first to be stored and all files in that context to be closed. As the programmer works on the new task, a new context starts building up for the new task. Mylar tracks all of the files that the programmer modifies when working on each task, and groups outgoing changes by task (Figure 1.5-6). All that the programmer needs to do to return to the first task is to reactivate it, causing the views and editors to return to the state visible on the top of Figure 1.5.

In an almost identical way, task context enables knowledge workers to regain access almost instantaneously to the information that they need when multi-tasking. When a user returns to a task on which he worked previously, our focused file and web browsing tool displays the files and web pages related to that task in the Navigator view (Figure 1.6) and re-opens the most interesting documents for easy access. Instead of seeing all of the files in the visible directories, or all of the web pages they had visited as part of this task, the knowledge worker only sees the information most relevant to the task and is able to multi-task without losing context.

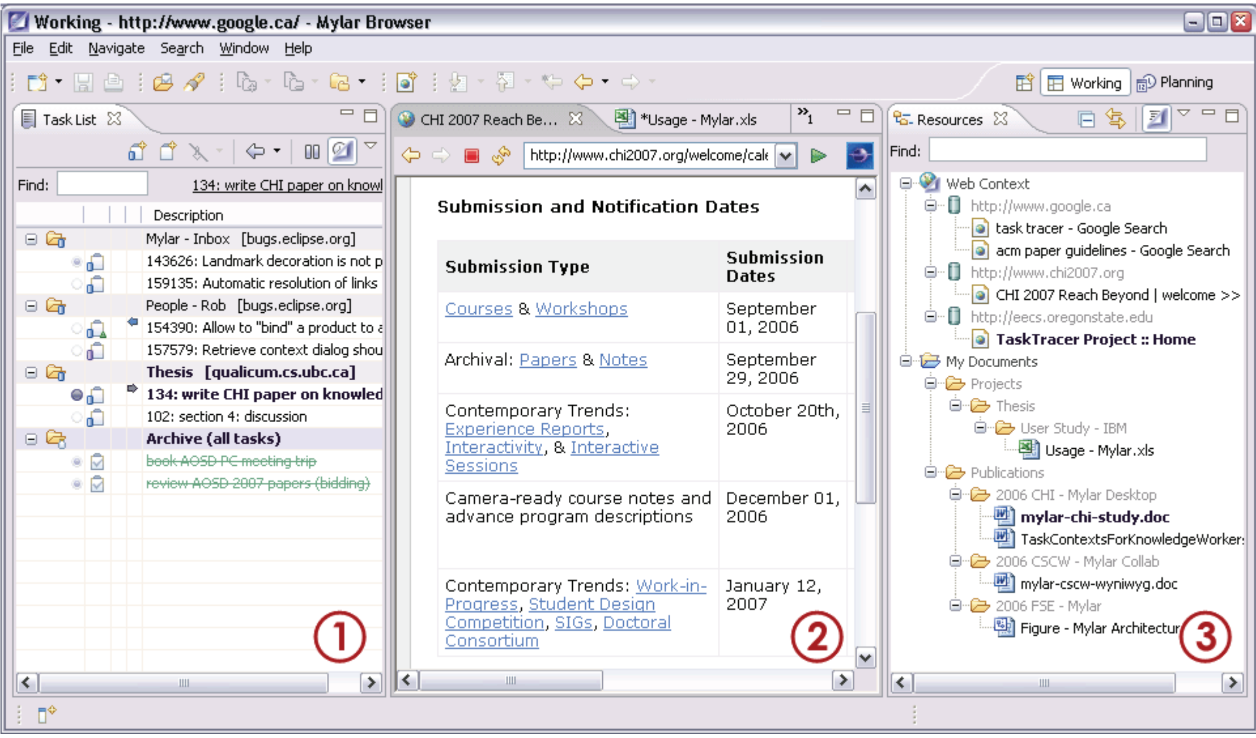

Figure 1.6: Browsing with task context

1.6 Overview

In Chapter 2, we present the task context model and operations. In Chapter 3, we describe the task-focused UI mechanisms that support integrating this model with IDE and browsing applications, and describe our implementation of task context for the programming and generic knowledge work domains. In Chapter 4, we present our validation in the form of three field studies of professional knowledge workers. We discuss potential improvements to the task context model and integration with collaboration and visualization tools in Chapter 5, overview related work in Chapter 6, and provide concluding remarks in Chapter 7.

2. Model & Operations

For a programmer, typical tasks include bug fixes, feature additions, and code base explorations. For other kinds of knowledge workers, tasks include other work activities that need to be tracked and prioritized [4]. For example, for lawyers, tasks can consist of creating contracts or researching prior cases. Some tasks are short-lived, requiring only a few minutes to complete; others are longer-lived, requiring some time each day over the course of weeks or months. Our model considers a task to be an atomic unit. Higher-level abstractions and organization of tasks, such as categorizations, hierarchies and sequences, may be layered on top. The examples we give in this chapter are for interaction with a software system, except where special provisions are made in the model to handle less well-structured information.

We define a *task context* as the information that a knowledge worker needs to know to complete a particular task. The information consists of a graph of the parts of artifacts (elements) and relations between artifacts. Each element and relation in the graph has a weighting that defines its relevance, a degree-of-interest (DOI), to the task. Our approach forms a task context based on the interactions that a programmer has with system artifacts and from the structure of those artifacts. Specifically, the information in a task context is defined entirely by an *interaction history*, which is comprised of a sequence of *interaction events* that correspond to the *direct* interaction that a programmer has with a system, and the *indirect* interactions that a tool can have on behalf of the programmer (each described in Section 2.1.1). Direct interactions include a programmer's *edits* and *selections;* indirect interactions include *predictions* and *propagations* that cause elements to be added to the interaction history that have not yet been interacted with directly in the task context.

Figure 2.1 provides a conceptual overview of this approach. As a knowledge worker works, interaction history is captured (Section 2.1) and is used to produce a task context graph (Section 2.2). The nodes and edges in this graph reference concrete elements and relations in the target information system, in this case different kinds of Java declarations (M=method, C=class, I=interface). As the interaction history is processed, our DOI function (Section 2.2) assigns a real number weighting to each element and relation, corresponding to the frequency of the access to the element or relation, less a *decay* factor that corresponds to the total number of interaction events processed. Accessing an element increases its weight, while accessing other elements decays the weight of infrequently accessed elements. Value ranges on the DOI specify which elements and relationships are interesting and uninteresting (Table 2.1). *Interesting* elements or relationships are those with a positive DOI value. *Uninteresting* elements or relationships are those with a negative or zero DOI value, which occurs either through decay or because the element or relation has never been the target of an interaction.

Table 2.1: DOI value intervals

Interval	Description
$(0, \infty]$	Interesting
$[-\infty, 0]$	Uninteresting

A task context can be operated upon and displayed. For example, the bottom of Figure 2.1 shows, from left to right, a display of all elements that have changed in a system as part of a task regardless of DOI, a display of only elements of predicted interest, and display in which an existing view of system structure has been filtered to show only interesting elements. Since the task context model is typically overlaid on existing views of system structure tailored to specific activities, such as browsing the hierarchy of a Java program, we call these display mechanisms *projections* of task context, and describe them in the next chapter.

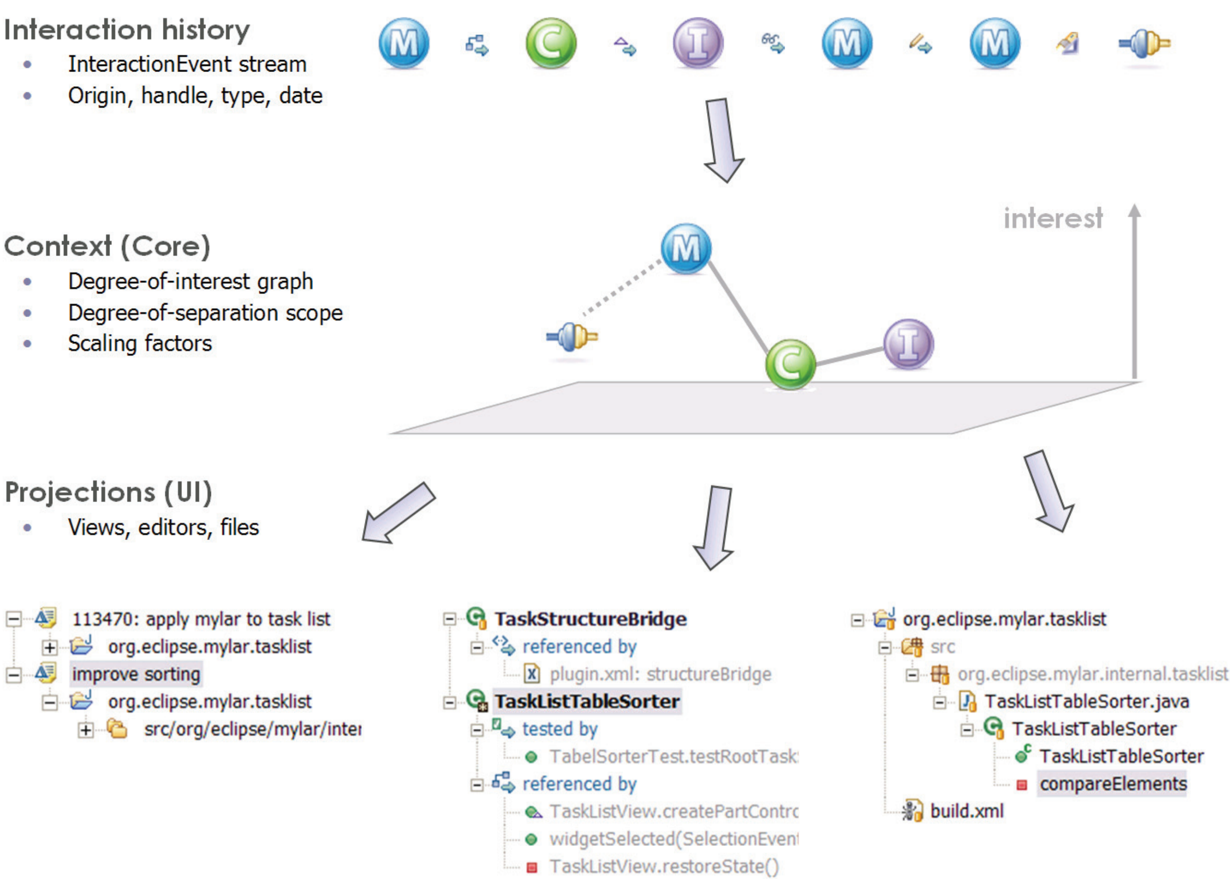

Figure 2.1: Constructing and projecting a task context

This chapter describes our core model and the minimal set of operations that we have defined to support integrating this model with existing information sources and user interfaces. We describe interaction

history in Section 2.1. We describe how to construct a task context from an interaction history in Section 2.2. We define task context operations in Section 2.3. To help guide the description of these operations, we provide an overview of the four operations in Table 2.2: the *composition* operation supports displaying and working with multiple task contexts concurrently, the *slicing* operation creates a new smaller context from an original context by applying a constraint that determines which elements and relations should be included in the resultant context, the *induction* operation induces interest on structurally related elements to support indirect interaction, and the *manipulation* operation supports direct manipulation of the DOI of elements by a user.

Table 2.2: Task context operations overview

<table>
<tr><th>Input</th><th>Operation</th><th>Output</th></tr>
<tr><td>1..n contexts</td><td>composition</td><td>New context composed of input contexts</td></tr>
<tr><td>context, constraint</td><td>slicing</td><td>New context resulting from applying constraint</td></tr>
<tr><td rowspan="2">context</td><td>manipulation</td><td rowspan="2">Events appended to interaction history for existing context</td></tr>
<tr><td>induction</td></tr>
</table>

Our definition of task context is not coupled to any particular kind of information source, application, user interface or task definition. However, an implementation of task context must provide bindings for each of these. We discuss our implementation of task context for an IDE and for a file and web browsing application in the next chapter.

2.1 Encoding Interaction

We derive a task context from an interaction histor*y*, defined as a sequence of interaction events that describe accesses of and operations performed on artifacts within a user's work environment.

2.1.1. Interaction Events

Each interaction event captures four pieces of information (Table 2.3). The *time* field stamps the event with the moment that it occurred. The *kind* classifies the event according to a schema representing the kind of interaction that occurred, such as a selection (Table 2.4). The *content type* provides an identifier that binds the element to a particular domain structure (e.g., "text/html"). The *handle* is an identifier that uniquely identifies that element for the given content type. This is the minimal schema needed to uniquely

associate typed events in time with structured elements, assuming that element identity is unique per content type.

Table 2.3: Interaction event schema

Time	The time of the event occurrence
Kind	Classification of event (Table 2.4)
Content Type	Identifier describing the kind of element operated upon
Handle	Identifier for the target element

An interaction history is naturally ordered by time. Events can be appended to the sequence. Deletions and insertions are not supported since the interaction sequence is a historical record and, as such, events cannot be "undone".

2.1.2. Element Identity

Interaction events are created by an interaction monitoring facility, such as the Mylar Monitor (Section 3.1.3), which is responsible for populating them with the kind, and with the identifying information of the concrete element corresponding to this event (i.e., the content type and handle). The content type and handle identifier fields bind the abstract task context elements (i.e., nodes in a graph) to domain structure elements (e.g., Java methods). The concrete relations between elements (e.g., method a "calls" method b) are defined at the domain level. A relation in the task context model is defined by an edge connecting the corresponding elements.

Over the course of an interaction history, the identity of an element can change. For example, if a file is renamed, and the identity is derived from the name, the handle field on past interaction events may not correspond to future interaction events. To preserve identity the interaction history supports updating the identity associated with past events. All other fields on an interaction event are considered immutable.

2.1.3. Event Classification

Table 2.4 summarizes our classification of interaction events. Some interaction events are the result of a programmer's *direct interactions* with program elements. For instance, a programmer may select a particular Java method to view its source, edit it, and then save the file containing it. Each of these actions corresponds to an event of a different kind being created and added to the sequence of interaction events for the task context, as demonstrated by the sample interaction history in Table 2.5.

Other interaction events are *indirect*, where program elements and relations are affected without being directly selected or edited by the programmer. For example, when working on T1 (Section 1.5), the programmer refactors the name of the `ResourceStructureBridge` class, causing all of the elements referring to that class to be updated. Each referring element updated through the refactoring results in an indirect *propagation* event of the edit to be appended to the interaction history. When the programmer directly selects the `getContentType` method (Figure 1.5-2), each parent of that method (its class, source file, package, source folder, and containing project) becomes relevant to the context, and a *propagation* event for each parent element is appended to the interaction history, so that a view can display these as if they had been interacted with directly.

Table 2.4: Interaction event classification schema

Event kind	Interaction	Description
selection	direct	Editor and view selections via mouse or keyboard
edit		Textual and graphical edits
command		Operations such as saving, building, preference setting and interest manipulation
propagation	indirect	Interaction propagates to structurally related elements
prediction		Capture of potential future interaction events

We also support *prediction* events, which describe possible future interactions that a tool anticipates the programmer might perform. An example of *prediction* is an event describing that a test may be of interest to the current task because it references a class in the task context. Whereas direct events are caused by user interaction, the indirect prediction and propagation events are issued by the induction operation, discussed in Section 2.3.4. The key differentiator between propagation and prediction events is that the former indicates an explicit intention of the programmer to interact with the elements involved in the propagation. In contrast, prediction events are a mechanism for capturing recommendations within the interaction history.

Table 2.5 provides an example of the sequence of interaction events that result from the programmer's initial work on T1 (Section 1.5). For simplicity, we use an event number to stand in for the time field of an interaction event. The programmer first selects a class. The selection event propagates to structurally

related elements, such as the file containing the class, the package containing the file, and so on[13]. The programmer then invokes a rename operation, and edits the name of the class. This edit propagates to each of the files that are subsequently renamed by the refactoring command[14]. The user then clicks a URL in the Java editor, opening the browser. This causes a selection of a web page, with another propagation to the web site that contains the web page.

Table 2.5: Sample interaction history

Event	Kind	Target(s)
1	selection	ResourceStructureBridge class
2..5	propagation	.java file, package, source folder, project
6	command	ResourceStructureBridge class
7	edit	ResourceStructureBridge declaration
8..16	propagation	4 XML and 5 Java references to ResourceStructureBridge
17	selection	URL to documentation page, reference to section
18	propagation	URL of web site containing documentation

2.2 Task Context Construction

We derive a task's context by processing an interaction history that describes the activity performed for the task. Each event from the portion of the interaction history related to the task contributes to a graph that represents the task context. If the handle of an event being processed refers to an element not yet represented in the graph, a node for the element is added to the graph. A *selection* event from the interaction history contributes an edge to the graph when the target element of the current *selection* event is structurally related to the target element of the last selection event processed. For example, if a programmer navigates from a method call to its declaration, the interaction history will contain the selection of the caller followed later by a selection of the callee. The result is an edge representing the Java reference relation between the two corresponding element nodes. The graph of task context can contain cycles (e.g., as a result of navigating recursive method calls) and can have multiple edges between nodes (e.g., both reference and inheritance).

[13] This propagation will cause each of those elements to become interesting and to show in the Package Explorer view (Section 3.3.1).

[14] This propagation causes the context change set to include the refactored files (Section 3.3.5).

We use a task's interaction history to compute a weighting for each element in the task context. The weighting is a real number value representing the element's degree-of-interest (DOI) for the task. This DOI value is based on the frequency of interactions with the element and a measure of the interactions' recency. The frequency is determined by the number of interaction events that refer to the element as a target. Each event kind has a different scaling factor constant, resulting in different weightings for different kinds of interaction. Recency is defined by a *decay* that is proportional to the position in the event stream of the first interaction with the element; like frequency, recency is also scaled. Since decay is proportional to the number of interaction events, it is independent of actual amount of wall clock time spent working with an element or relation. Instead, decay is based on the amount of interaction as punctuated by the events.

Algorithm 1 describes how we compute a DOI value for an *element*, given an interaction history represented as a sequence that contains one or more events with the element as the target. We iterate over a subsequence consisting of just the events involving the element (line 4), increment the *interest* value of the element based on the kind of the current event (line 5) and if the *interest* has not offset the decay, reset the decay to start at the last interaction with the element (lines 7-9). This algorithm ensures that elements that have decayed to a negative interest have their interest become positive when interacted with again.

```
DOI(element, events)
1   elementEvents = WITH-TARGET(element, events)
2   decayStart = elementEvents[0]
3   interest = 0
4   for each event in elementEvents
5      interest += SCALING(KIND(event))
6      currDecay = DECAY(decayStart, event, events)
7      if interest < currDecay then
8          decayStart = event                // reset decay
9          interest = SCALING(KIND(event))   // reset interest
10  totalDecay = DECAY(decayStart, LAST(events), events)
11  return interest – totalDecay

DECAY(fromEvent, toEvent, eventSeq)
12  decayEvents = SUBSEQ(fromEvent, toEvent, eventSeq)
13  return |decayEvents| * SCALING(KIND-DECAY)
```

Algorithm 1: DOI for Task Context

The SCALING function returns the constant associated with each event kind and with the KIND-DECAY constant. The DECAY function computes the decay to be proportional to the size of the subsequence

from `decayStart` to the most recent event and includes events not in `elementEvents`. As an example, consider how the interaction history from Table 3 contributes to the weighting of the `ResourceStructureBridge` element at the end of the interaction history. The element is most recently edited at event 7. Assuming SCALING returns 1 for selections, 0 for commands, 2 for edits, and 0.1 for KIND-DECAY, and noting that there were no propagated events with that element, the three iterations through the loop will result in 1+0+2 = 3 for *interest*, and (18-1)(0.1) for `totalDecay`, resulting in a DOI of 1.3. If 30 more interactions happened with another element, the DOI value would become -1.7. A subsequent selection would cause the DOI to be reset to 1 – 0.1 = 0.9.

A relation in the graph is composed of a source and a target element. The DOI of a relation is computed using the same DOI algorithm, but using the relation's target element:

DOI-R(*relation*, *events*) = DOI(TARGET(*relation*), *events*)

For example, if a programmer navigates repeatedly between a method call and its declaration, the DOI of that relation will increase from repeated selections of the declaration[15]. If the programmer navigates 'back' and 'forward' between the two several times, the two resulting directed edges will both have the same DOI.

Our construction algorithm for task context takes as input any sequential stream of interaction events whether it is being gathered on-line (e.g., when the user is working on a new task), or was stored off-line (e.g., when resuming a previously worked-on task). At any point in the construction process, each node and edge in the task context's graph can be queried for its DOI value. This value is computed from the interaction history that is associated with the task context when the query is made. Task context can thus be built interactively as a programmer works, or recreated by parsing a previously stored interaction history.

The resulting task context is a graph representing each element and relation in the context. The weightings are not explicitly stored in the task context, and instead the DOI of each element and relation is computed from all the interaction with that element to date[16].

---

[15] Note that for some domain structures or usage scenarios it may be impossible to determine the relation between two elements that are consecutively navigated (e.g., a navigation that results from a suggestion made in a phone conversation). For such cases, the relation is an arbitrary navigation, with no binding to a concrete relation in the target domain.

[16] The DOI algorithm can be computed frequently in a typical application (e.g., in Mylar IDE thousands of DOI computations happen in what needs to be an instantaneous view refresh, where instantaneous is defined as <200ms by the Eclipse UI). An implementation can cache previous computation values or use a data structure that optimizes for the DOI computation. Optimizations for parsing long interaction histories are discussed in Section 3.2.4.

By capturing and weighting the interaction of both elements and relations that have been interacted with directly and indirectly, the task context exhibits topological properties that can be leveraged by operations (Section 2.3) and by projections that display the task context (Section 3.2.1). As illustrated in Table 2.6, these properties include groupings, paths, and interest propagations (Section 2.3.4) that emerge from the way that the DOI algorithm derives the interest of relations from interaction with elements. These properties can be displayed directly in graphical views of task context (discussed in Section 5.4). The set of properties described here is not intended to be comprehensive, but to illustrate the behavior of the algorithm when processing interaction sequences for structurally related elements.

Table 2.6: Illustrations of topological properties of the task context

Property	Description	Illustration
groupings	Elements worked on in conjunction will have similar DOI values (e.g., an API method, XML file, and web page all referred to while editing a method).	
paths	When elements are frequently navigated between via less interesting elements, the end-points and path will have a high DOI (e.g., when working on two ends of a method call chain).	
propagations	Repeated interaction with an element increases the DOI of structurally related elements (e.g., when editing a class the containing method and superclass will gradually increase in interest).	

2.3 Task Context Operations

We can use a task context as input to various operations that support integrating task context into applications. Sometimes a single task context may not contain all of the relevant information needed for an activity, such as a code review. We use the term *composition* to refer to operations that produce a composite task context from individual task contexts. Sometime the converse is true and a subset of the context may be relevant. Even though the size of a task context is typically smaller than the size of the system, it can still be too large or contain too many different kinds of elements and relations to assist with other activities, such as the unit testing that is done by programmers, or the searches that are performed by query tools. We use the term *slicing* to refer to an operation that produces a subset of a given task context (i.e., a subset of the elements and relations, which can result in one or more sub-graphs depending on the properties of the slice). To enable a user to tailor a task context manually we also support *manipulation* operations. In order to grow the task context to encompass structurally related elements, the *induction* operation supports indirect interactions in the form of predicted and propagated interactions.

All operations on a task context take one or more contexts as input (as well as other arguments) and either update that context or produce a new context. This ensures that all of the operations are orthogonal and can be applied or combined in any order. Since a task context is defined by the set of interaction events associated with it, all of the operations are expressed in terms of adding or removing interaction events to form the output context. Since events cannot be removed directly from a context, operations that go beyond appending interaction events produce a new context (referred to as a context′ below).

2.3.1. Composition

Each task context's interaction history corresponds to a single task. However, some activities can require displaying the contexts of several tasks simultaneously. For example, a programmer might want to create a *composite context* from T1 and T2 (Section 1.5) to perform a code review of the work that was done on those two tasks and have the element highlighted according to each context. The composition operation takes as input one or more task contexts, and combines them to form a single composite context[17].

The composition operation combines each of the interaction event sequences of the input contexts (Table 2.7). Task contexts maintain the sequence of interaction events in time-sorted order, ensuring that the DOI algorithm can process the events from a composite context identically to a non-composite context. When the DOI algorithm is run on the composite context it returns a DOI corresponding to the interest of the element representative of what it would be if all of the tasks had been performed as a single task. The resulting composite context is identical to a non-composite context, other than the fact that it is composed of an aggregate sequence of interaction events.

Table 2.7: Composition operation

Input	Output	Description
1..n contexts	context′	Combine the interaction events of each context to form a single time-sorted sequence Run the DOI algorithm on the merged sequence

Since a composite context is simply the combination of the interaction events of each composed task context, a composite context can be the target of interaction just as a single task context can be. The interaction mirrors the effect of the composition operation: interaction events are duplicated across each

[17] Our implementation treats composite contexts identically to regular contexts, having each implement the same interface. Mylar's task-focused UI is always operating on the currently active composite context, which by default only has one context that it is composed of, but enables additional contexts to be activated.

of the contexts that make up the composite context. The identity of interaction events plays an important role in supporting a user's interaction with a composite context, for example, when the user is interacting with more than one task concurrently. When the DOI algorithm is used on such a composite context, it needs to treat identical interaction events distributed across composing contexts as a single event. Otherwise working with three tasks active would cause the DOI of one interaction with one element to be three times that when a single context is active. As such, supporting this distribution of events in an implementation requires either a data structure that is aware of interaction event identity (as in our implementation), or for the composition operation to discard duplicate events.

2.3.2. Slicing

Task context slicing is an operation that takes as input a task context and outputs a task context containing all elements and relations of the input context that meet a particular constraint (Table 2.8). A constraint tests one or more of the kinds of interactions associated with an element or relation (e.g., includes only predicted ones), DOI values (e.g., include elements and relations with a high DOI), or the underlying domain structure (e.g., include elements that are Java methods). An example of a slice that tests the interaction events but ignores the elements' contents and DOI is to gather all interesting files that have interaction events of the kind *edit*; this slice can be used to determine which files to include in a source code commit (Section 3.3.5). An example of a slice whose constraint tests both the underlying structure and the DOI is to gather all interesting test cases for running a test suite of the active context (Section 3.3.4).

Table 2.8: Slicing operation

Input	Output	Description
context, constraint	context′	For each element, relation, and interaction event apply the test to determine if the corresponding element or relation should be included in the output context. If test passes, include all interaction with the element/relation in the interaction history for the output context.

Whereas composition produces a context that defines a superset of the interaction, slicing is an operation that defines a subset. These operations can be combined. For example, consider the depiction of the interest of two tasks that have a large number of interesting elements, represented by the two lines in Figure 2.2. Composing the task contexts creates a union of the elements in each, resulting in elements that occur in both contexts having a DOI corresponding to what it would have been if they occurred in a single context (line A in Figure 2.2). If all we are interested in is the intersection of the two task contexts, we can slice the composite context to include only the elements in both (line B in Figure 2.2). Such an

intersection can be used to identify 'hot spots' in code when composing a large number of contexts that result from the dozens or hundreds of tasks performed during a phase of software development.

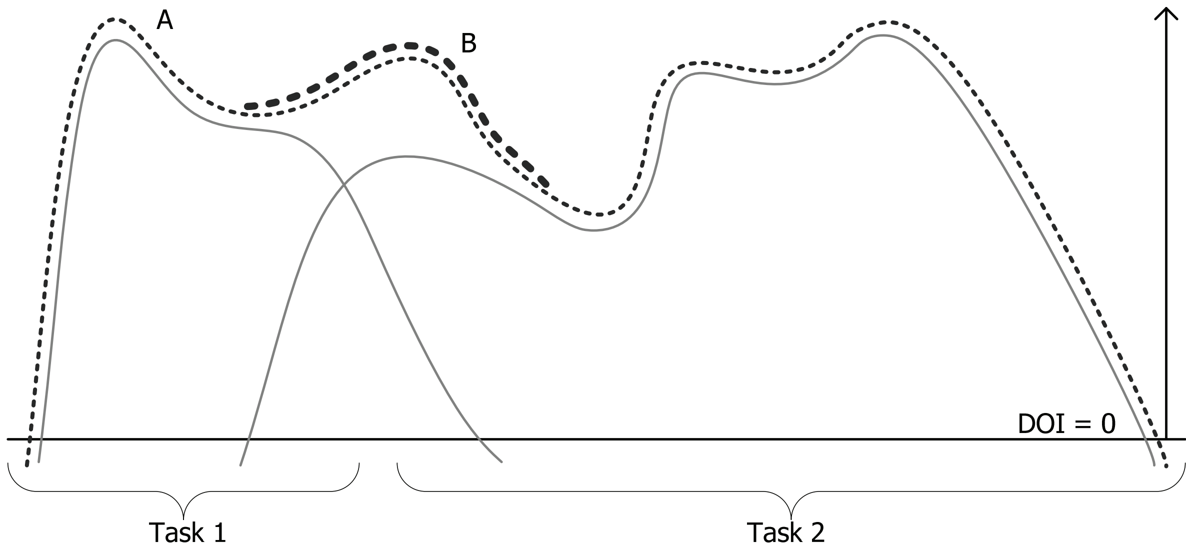

Figure 2.2: Combining task context operations

2.3.3. Manipulation

Our DOI function provides an approximation of interest and can produce a value that does not match the user's expectation either by being tuned incorrectly or by failing to capture a relevant interaction. We provide a mechanism for directly manipulating a task context by allowing the user to issue command events that result in predictable changes to the task context. For example, if an element is interesting but should not be, a "Make Less Interesting" command can issue the interaction events needed to reduce the interest of that element and make it disappear from a view, leaving only interesting elements visible. Since we assume that interaction cannot be undone, it is not possible to delete elements from a task context. As such, making an element uninteresting is supported via an interaction event based operation called task context manipulation (Table 2.9).

Table 2.9: Manipulation operation

Input	Output	Description	Example
context	context	Add the command interaction event(s) to the task context	User explicitly marks an element as a uninteresting via a UI affordance

A manipulation operation adds one or more interaction events to the task context so that the context is updated in order to match the user's expectation of the operation (e.g., in our implementation an element can be made explicitly interesting or uninteresting (as described in Section 3.2.2). Manipulation

commands differ from other interaction events because their processing must have not only a predictable effect on the task context, but one that causes a precise change in the DOI of the target element. For example, if a document has a DOI of 10, and the user issues a manipulation event that should make this element have a negative interest (i.e., disappear from view), the interaction must contribute -10 to the DOI. To avoid having the algorithm include a special rule for such events, a number of command manipulations corresponding to the right decrement can be issued[18], effectively undoing the interest contributions of the other interactions with that element.

2.3.4. Induction

A significant fraction of the commands executed by a user are commands related to searching for structurally related elements that need to be edited or referenced in order to complete a task [52]. We can use the information in a task context to predict what elements might be relevant to completing the task, but with which the user has not yet interacted directly. For example, if a programmer is working on a Java class, which is referred to by an XML element, the XML element can have a predicted interest if a tool determines it is likely to be part of the task context at a future time. By "predicted interest", we mean a DOI that results entirely from interaction events with the prediction kind (Table 2.4) and not by other events such as selections. Such predictions can come from running automatic searches on the programmer's behalf, for example using context slices as both input and scope for the searches (Section 3.3.3).

We can also use existing interaction information to propagate the interaction to structurally related elements. For example, if a programmer selects a class, events can propagate to elements directly related to the class, such as its parents in the containment hierarchy. The propagation events differ from the predicted events because they express an intention of the user to select structurally related elements known by the domain mapping to be structurally relevant, whereas the prediction events express a recommendation of elements that may be relevant.

We call the operation that adds both prediction and propagation events *induction* because it results in interest being induced on structurally related elements. The output of the induction operation is a set of interaction events that contribute to the task context, with an event kind of prediction or propagation indicating the indirect interaction (Table 2.10). As with the manipulation operation, these events are simply appended to the task context's interaction history. This approach enables the use of the DOI

[18] This can cause a large number of manipulation events if a high DOI element is reduced in interest. However, an implementation can compensate for this by accounting for the redundancy, e.g., by collapsing repeated events (Section 3.2.4).

algorithm to rank related elements, with the DOI of less frequent results decaying and more recent and more frequent results producing a higher DOI.

Table 2.10: Induction operation

Input	Output	Description
context	context	Use domain-specific relations or heuristics to find related elements, and add them to the context via prediction or propagation events.

2.4 Task Activity Context

To correlate each context to a particular task and to support multiple task contexts, we need a mechanism to associate interaction events with a task. We achieve this association by capturing a separate stream of interaction events in which the target elements are tasks instead of system artifacts. This stream of interaction events can be used to form an interaction history over tasks instead of system artifacts. This parallelism to how a task context is defined means that we can apply the same model and operations to this separate interaction stream, forming a meta task context. We call this the task activity context. As depicted in Figure 2.3, at the base level, the nodes in a task context are resources, and the interaction history contains events such as selections and edits of the corresponding files or web pages. At the meta-level, the nodes in the task activity context are the tasks, and the interaction history contains events such as the selections and edits of those tasks.

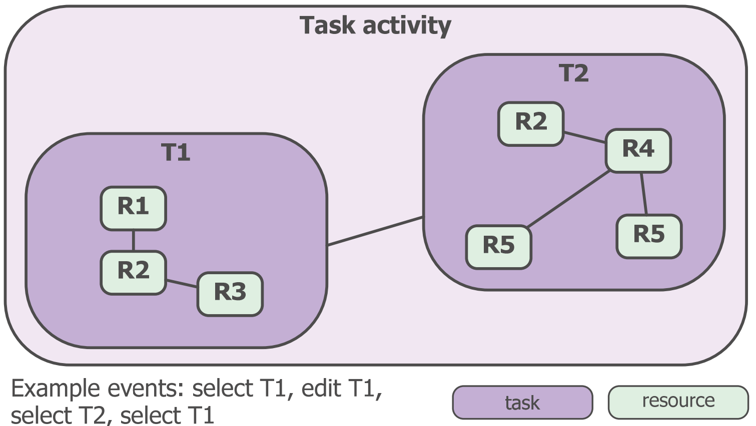

Figure 2.3: Illustration of task activity context

We process the interaction history for the task activity context the same way that we process a task context. For example, a programmer can indicate that work should be associated with a particular task by opening a bug report; this action causes a *selection* event on that task. Commenting on the bug report causes an *edit* event. The act of a programmer switching to another application window causes a *command* event that indicates work on a task has stopped. Supporting this meta-context ensures that the

DOI algorithm and operations that we have defined work for the task activity context just as they work for the base task contexts. Keeping this interaction history separate from the task context interaction history ensures that working on tasks does not cause the DOI of the elements in the active task context to decay. The task activity context also enables an implementation to focus the UI on the highest DOI tasks, as we discuss in Section 5.5.6.

2.5 Model Summary

Our goal with the task context model has been to make it as simple as possible, while ensuring that we could capture any kind of interaction that a user could have with programming and file and web browsing applications. We have described a set of operations intended to focus an application on task context, and discuss our implementation of task context and operations in the next chapter.

3. Implementation

There are two key challenges to implementing task context for a particular application: determining the mapping from application usage to the more abstract interaction history, and implementing the necessary task context operations to integrate task context with the application. We implemented task context on the Eclipse platform, which is a generic application framework used to build applications that include the Eclipse Java IDE. Eclipse provides several mechanisms that support integrating task context: a unified selection service for monitoring user interaction[19], extensible UI mechanisms[20] that support a modular implementation of the task-focused UI, and the OSGi plug-in mechanism[21] that supports a loose coupling of mappings to different domain-specific artifacts (Section 3.1.1) and enables deployment to different application configurations. Task context can be implemented on another application platform if equivalent mechanisms are available on that platform. In this chapter we describe the architecture in terms of our Eclipse and Java-based implementation, but the concepts are generic and have been implemented by another research group on a different application platform [12].

In this chapter, we describe the Mylar architecture that supports integrating task context with a diverse set of domain structures (Section 3.1). We define the user interface mechanisms that present task context to the user (Section 3.2) and we describe our two implementations of task context: one for an IDE application called Mylar IDE (Section 3.3) and one for a file and web browsing application called Mylar Browser (Section 3.4).

3.1 Architecture Overview

To adequately test task context, we needed an implementation that could scale to handle very large systems with several kinds of structured and semi-structured artifacts, that could support task management and that could integrate with existing tools used by knowledge workers. The Mylar Architecture supports these criteria by providing three frameworks (Table 3.1): Monitor, Context and Tasks. Each framework is broken into two parts. The "core" part provides a model and operations not

---

[19] http://eclipse.org/articles/Article-WorkbenchSelections/article.html [verified 2006-10-02]

[20] http://eclipse.org/articles/Article-UI-Workbench/workbench.html [verified 2006-10-02]

[21] The OSGi plug-in model is a component model that allows a number of Java classes to form a single component and supports the loose coupling of components: http://eclipse.org/osgi [verified 2006-10-02]

coupled to any particular application platform and suitable for use in server-side applications or for embedding in another application framework, whereas the "UI" part is coupled to the Eclipse UI framework. As we describe each of the three frameworks, we also discuss how to extend each framework component[22].

The Monitor framework transforms a user's interaction with the application into the interaction events that are processed by the Context framework. The Monitor may be extended by user study plug-ins to support measuring usage statistics in field studies, as described in the next chapter.

The Context Framework implements the task context model, including interaction histories and task context operations. The Context Framework requires a definition of tasks and is flexible in this definition because it is not coupled to the Task framework. As such, an alternative task management framework can be used for task management and activation purposes. The UI portion of the Context framework implements the task-focused UI (Section 3.2.2) which we implemented both for programmers and for knowledge workers (Sections 4, 4.4).

The core of the Tasks framework provides the definition of tasks that maps user-defined work items to tasks. The UI portion of the Tasks framework provides task management facilities (Section 3.2.3).

Table 3.1: Mylar frameworks

Framework	Core	UI	API clients
Monitor	Interaction history	Interaction monitoring	Monitors
Context	Task context	Task-focused UI mechanisms	Bridges
Tasks	Task management	Task views and editors	Connectors

Each framework supports a different kind of API client. The Monitor Framework supports *monitors* that observe user interaction. For example, user observation mechanisms described in the next chapter incorporate monitors for the purpose of measuring productivity and reporting on task activity. The Context framework supports *bridges*, which map between the abstract elements in the task context model and the concrete elements in some domain, for example, the Java programming language. The Tasks

[22] The Mylar 0.7 implementation is 82,684 source lines of Java code (SLOC), 8,042 lines of XML and XML schema, 107 packages, 950 classes, and 50 interfaces. The Mylar framework components are 3,695 source lines of code, 24 packages, 399 classes, and 40 interfaces. I implemented over 80% of the total code in Mylar 0.7, the entire Context and Monitor frameworks, and led the implementation of the other components, such as the Tasks framework. Credits to the many open source contributors are summarized at: http:// eclipse.org/mylar/team.php [verified 2006-12-20]

Framework supports *connectors*, which define the unit that makes up a task. For example, the Bugzilla connector defines tasks as bug reports or enhancement requests.

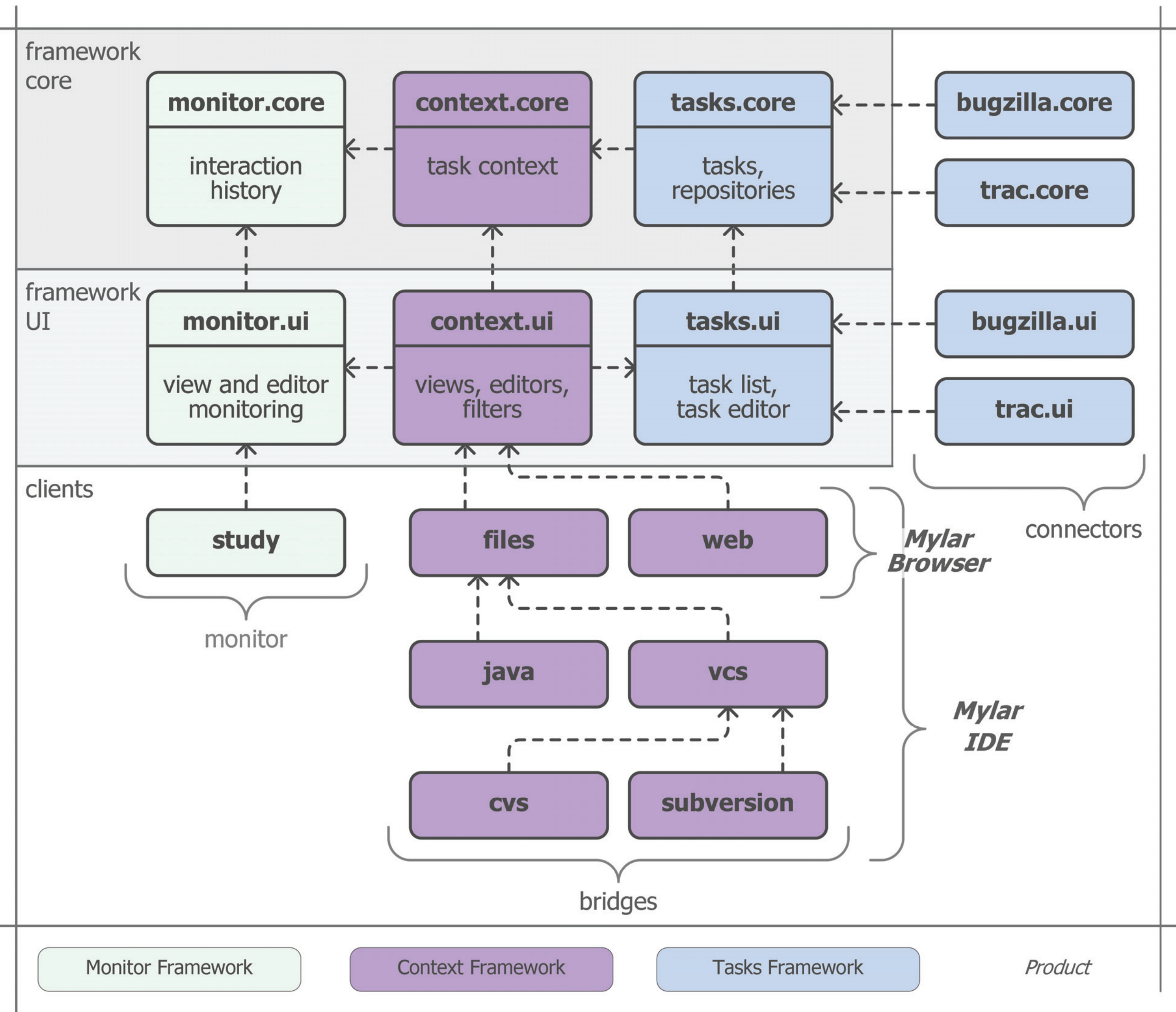

Figure 3.1: Mylar architecture showing OSGi plug-ins and their dependencies

Figure 3.1 demonstrates our component model, with each box corresponding to an Eclipse OSGi plug-in[23]. The key property of this architecture, that is needed to support our validation, is the loose coupling of the framework components. For the IDE study, we required the ability to deploy only the monitoring facilities to support gathering baseline data of Eclipse IDE usage (i.e., the `monitor.core`, and `monitor.ui` components). We then needed to deploy the full Mylar IDE integration and a mechanism for integrating the Bugzilla task repository used by programmers (all components listed in Figure 3.1). For the knowledge worker study, we needed to deploy the Mylar Browser without any of the IDE-specific components or the Bugzilla Connector, and to integrate task context with a subset of the Eclipse platform

[23] An Eclipse-based application, such as the Mylar Browser, is a collection of OSGi plug-ins.

that was suitable for file and web browsing. The other plug-ins visible are the Trac[24] connector (Section 3.2), and the version control system (VCS) bridges used for integrating change sets support CVS[25] and Subversion[26] (Section 3.3).

In the next section, we zoom into part of this diagram to show how bridges integrate with the underlying application.

3.1.1. Bridges

While our task context model is defined in terms of a generic set of interaction events, the actual events need to be issued by a mechanism that understands both domain structure (e.g., Java) and the UI of the tool for working with that structure (e.g., the Eclipse UI for Java development). We call this mechanism a *bridge* from the task context model to the domain structure. Each bridge handles a single content type. We have created bridges for Java, two XML dialects (Ant[27] and Eclipse plug-in descriptors), files, web resources, and the tasks themselves.

A *structure bridge* is responsible for mapping elements and relations in the task context to and from the domain structure of a particular content type. This involves mapping context elements to domain model elements and resolving relations between elements. Structure bridges must also update the identity of elements in the model when elements move within the domain structure as a result of refactoring. For example, the Java structure bridge integrates with Eclipse's Java model (derived from a Java AST), and is able to map between the handle identifiers of Java elements in the task context and the objects corresponding to those elements in the IDE. It resolves the relations between Java elements including references, inheritance, and read/write access of fields. When elements are moved or refactored and their handle identifier changes, the Java structure bridge notifies the task context model so that the identity of those elements, and the previous interaction with hose elements, can be preserved.

The implementation of a structure bridge defines the level of granularity of domain structure supported, which can include both well-structured and semi-structured data. For example, the Java structure bridge is

---

[24] http://trac.edgewall.org [verified 2006-10-20]

[25] http://www.nongnu.org/cvs [verified 2006-10-20]

[26] http://subversion.tigris.org [verified 2006-10-20]

[27] http://ant.apache.org [verified 2006-10-02]

well-structured and maps the identity of elements down to the member level[28]. In contrast, the file structure bridge is only aware of the file and directory structure, and not of the contents of files. This means that it works equally well for both binary images and Microsoft Word[29] files. However, if semi-structured data is interacted with inside such a file, for example, a section in a Word document, the interaction will only be registered at the level of the file. The finer the granularity of structure bridges in an application, the higher the fidelity of the task context model (discussed further in Section 5.5.2).

A *UI bridge* is responsible for monitoring interaction with the parts of the IDE for which it is implemented, such as Eclipse's Java tools in the case of the Java UI bridge. It maps the programmer's interaction with the UI to the interaction history schema of selections, edits, and commands. These can include keystrokes in the Java editor, refactoring commands, and element selections. Each UI bridge also specifies which views and editors participate in task context projection (Section 3.2.1).

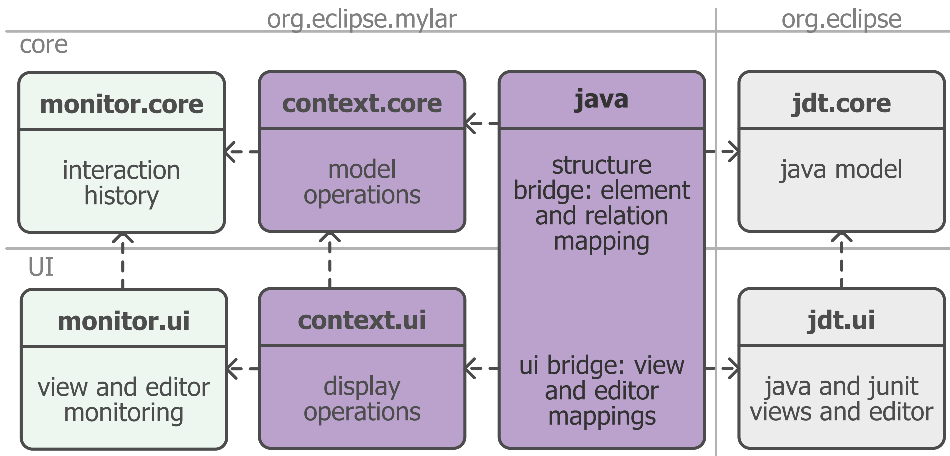

Figure 3.2: Mylar plug-in architecture and Java bridges

Figure 3.2 shows the dependency structure of a bridge implemented as an Eclipse plug-in, with arrows corresponding to plug-in dependencies. The `context.core` and `context.ui` plug-ins provide the structure and UI bridge extension points that the `java` plug-in uses to map and display the concrete Java elements on which the programmer is working. Bridges for other languages may exist alongside the bridges for Java. Bridges can also be composed. For example, the `java` structure bridge[30] extends the

---

[28] It is feasible to implement an even finer-structured bridge, which maps identity down to the statement and expression level. However, as the structure views in IDEs, such as Eclipse, show structure only down to the member level, this was not necessary.

[29] http://office.microsoft.com [verified 2006-10-02]

[30] It is also possible to implement bridges as separate core and UI plug-ins, for example, if the core plug-in is to be reused in a non-UI application.

resource structure bridge responsible for understanding file and directory structure (not shown in Figure 3.2 for the sake of simplicity).

3.1.2. Mapping to Interaction History

Bridges should not issue interaction events directly[31], but instead provide the identity and relations between elements. The monitoring framework issues interaction events corresponding to the concrete domain elements specified by the bridges. For example, when a propagated event needs to be issued for the parent of a selected element, the bridge corresponding to the element is asked for the identifier of the parent, and the interaction history facility uses the induction operation to issue the event. Since bridges can be composed, the induction operation can involve multiple bridges. For example, if a Java method is selected, the Java structure bridge provides the class containing the method. When the next propagation needs to occur, from the class to the file containing the class, the file structure bridge provides the identity of the file. The same is true for predicted events where the relations may be all Java references to a particular method. In this case, resolving the relation involves performing a Java search and returning the results, which the induction operation then appends to the interaction history.

To support voluntary use of the tool for daily work on real systems, we needed to ensure that the interaction event architecture scaled to large systems without excessive memory or performance overhead. Task context grows with the amount of interaction, not with the system size, ensuring scalability for large systems. Scalability to large or long-running tasks results from the relatively small amount of interaction that users perform in relation to the system size (Section 3.2.4). Since the number of propagation events can vary with system size, the number of propagation steps is bounded by an exponential drop off which limits the number of events issued from each interaction. The search scopes used for prediction are based on context slices and scale with the size of the context. For example, whereas a user-driven search through the workspace might include every resource and can be slow to run on a large workspace[32], a context search can query only the slice of interesting elements (Section 3.3.3).

---

[31] This is not yet strictly enforced by the Context framework to support flexibility and experimentation, but would be beneficial to prevent a poorly implemented bridge from issuing an unrepresentative number of interaction events.

[32] Although Eclipse provides rich text indexing facilities, making many searches fast, the size of an enterprise programmer's workspace is still too large for such searches to constantly run in the background on current commodity PCs (e.g., 2GHz CPU 2GB RAM).

3.1.3. Interaction Monitoring

The Mylar Monitor component provides interaction monitoring. The Mylar Monitor is a separately installable component that collects information about a user's activity in Eclipse. Clients of the monitor include the Context UI and user study plug-ins, which can report on Eclipse usage trends [52]. The `monitor` plug-in accepts listeners to different Eclipse events, including preference changes, perspective changes, window events, selections, and commands invoked through menus or key bindings and URLs viewed through the embedded Eclipse web browser. The `InteractionEvent` class used by the monitor encapsulates direct and indirect interactions. Since the monitor encapsulates the mapping between user interaction and the interaction history, consistency between how different kinds of affordances generate interaction events is important. Scaling factors are used to determine the bias between different kinds of interactions (e.g., bias of edits vs. selections). We have used a single set of scaling factors throughout the study versions and latest released version of Mylar 0.7, listed in Table 3.2. We discuss the formulation of these scaling factors in Section 5.1.4.

Table 3.2: Scaling factors used in Mylar implementation

Factor	Value
selection	1
edit	0.7
decay	0.017
command	0 (ignored)
propagation	1
prediction	1

The `monitor.ui` plug-in supports the transparent capture of these events into a local file and the upload of these events to a web server. The uploading mechanism includes functionality to track anonymous IDs for users, to obfuscate the handles of targets of selections and other such user data, and to prompt the user to view and send the log to a web server. The Monitor also detects and reports when there has been a period of inactivity with Eclipse. User study plug-ins that extend the monitor were used for the validation of both the Mylar IDE and Mylar Browser applications. To help with the analysis of traces collected with the Mylar Monitor, an additional `monitor.reports` plug-in provides an API to process records for a user across one or more interaction histories, enabling the analysis of interaction history data (Section 5.4.3).

3.2 Integration

Integrating task context into an application requires mechanisms to project the context onto existing data structures and UI facilities to present these projections to the user. Together we refer to these mechanisms as the task-focused UI. To support task context, the task-focused UI mechanisms must be combined with task management facilities that allow a user to indicate on which tasks work is occurring. In addition, a mechanism for storing and retrieving contexts is required. These mechanisms, described in the following sections, are common to any application supporting task context, including our Mylar IDE and Mylar Browser implementations.

Each of the integration mechanisms relies on one or more of the task context operations described in Section 2.3. Projections rely on slices, task-focused UI mechanisms rely on both slices and manipulation, and task management mechanisms rely on composition. The Active Search facility, discussed in Section 3.3.3, uses the prediction mechanism.

The integration mechanisms also use a categorization of elements and relationships based on their DOI. Table 3.3 defines the two thresholds we use to define three categories. *Interesting* elements (and relationships) are all of the elements with a non-zero DOI, meaning all elements (and relationships) that have been previously selected and have not decayed to be uninteresting (Section 2.2). *Landmark* elements (and relationships) are elements (and relationships) that are considered very interesting because their DOI has passed a threshold. *Uninteresting* elements (and relationships) have a negative DOI.

All of the task-focused UI mechanisms that we have implemented rely on fixed thresholds. Later in this thesis, we discuss alternative implementations of thresholds, including the possibility of implementing adaptive thresholds, which can be combined with filtering and tree expansion management to implement features such as the guaranteed visibility [50] of landmarks in a filtered view (Section 5.5.3).

Table 3.3: Thresholds used in the Mylar implementation

Threshold	Value
interesting	0
landmark	30

3.2.1. Task Context Projections

Task context projections use the task context to decorate an existing data structure (Table 3.4). Projections use the DOI values from the task context to display a weighted version of the data structure. Projections can be combined with the task context slicing operation if the display mechanism is focused on displaying one kind of element or one kind of relation. For example, we project a task context onto a

hierarchical view of system structure (the Package Explorer, Figure 1.1) to show only elements with a positive interest in the current task context, filtering all uninteresting elements. Similarly, a projection of a task context onto a table can be used to sort elements by interest. To integrate projections without needing to modify the original data structure, the user interface facilities of the underlying platform must support third-party extensions. For example, the Eclipse user interface allows decorators, filters, and sorters to be added to any view in Eclipse.

Table 3.4: Task context projection

Input	Output	Description	Example
context, data structure	decorated data structure	For each element and/or relation in the data structure decorate it with the interest level	Show only interesting elements, and making the most interesting elements bold

Projections can support any view of the structure in the task context. In this chapter, we describe projections onto tree and list views. Graphical layouts are also possible and are discussed in Section 5.4.2.

3.2.2. Task-Focused UI Mechanisms

Task-focused UI mechanisms (see Table 3.5) integrate task context with the existing UI facilities in an application framework. The goal of each mechanism is to focus the UI by projecting the task context onto a particular UI facility; for example, ensuring that only the elements within the context are visible in a structure view and that the interesting elements are always expanded. In this section, we describe the task-focused UI mechanisms that we defined for the Eclipse framework. These mechanisms generalize to other application frameworks that use similar UI mechanisms[33].

Mylar implements each mechanism listed in Table 3.5, using either the entire task context or a slice of the task context as input. For instance, in Mylar IDE, the filtering, sorting, highlighting, text folding, and tree expansion management facilities use a slice of all interesting elements as their input. The editor management uses a slice of all interesting files. The Eclipse perspective[34] management uses a slice of all views last visible in the context[35]. It is not necessary for an implementation to support all of these

[33] A key difference between Eclipse and window managers such as Microsoft Windows is that Eclipse uses views, editors, and perspectives in place of windows. As such, the “application window” section of Table 3.5 describes mechanisms for managing each. The same facilities could also apply to managing windows.

[34] An Eclipse perspective is a layout of structure views and editors that can be changed and later restored.

[35] View opening and closing issues command interaction events.

mechanisms. The choice of mechanisms can be tailored to the particular activities supported by the implementation. For example, an implementation of task context for a text viewer could use the interest decoration mechanism alone to help the user pick out relevant regions of text.

Table 3.5: Task-focused UI mechanisms for exposing interest

Target	Mechanism	Description	Applies to
structure views	Filtering	Elements below a certain DOI threshold are excluded	List, tree, or graphical view
	Ranking	Elements are ranked or sorted by their DOI	List views, child nodes within tree views
	Decoration	Foreground or background color of elements indicates DOI	Any view, can be continuous (e.g. background gradients) or discrete (landmarks are bold)
	expansion management	Tree nodes are automatically expanded to correspond to a slice of interest Text corresponding to uninteresting automatically elided	Any tree view (e.g. landmark nodes always expanded) Any text viewer supporting folding
application window	view management	Automatically apply focusing mechanisms to a view on task context activation	Any structure view
	editor management	Editors corresponding to uninteresting elements are automatically closed	Any editor listing (e.g., open files in an application)
	perspective management	The views associated with a context are automatically restored on activation	Any view management facility (e.g. Eclipse's view groupings)

The first four mechanisms listed in Table 3.5 can be applied to any structure view, where a structure view is defined as a UI mechanism that displays elements and may additionally display relations between elements. For example, the "Focus on Active Task" button that Mylar can add to any structure view in Eclipse causes all four of these facilities to be enabled. The filtering causes any element without a positive DOI to be removed from the view. The ranking causes items within the view to be ordered according to DOI value. Decoration provides a visual representation of DOI level with one or more of foreground color, background color, font, textual annotations and iconography. Expansion management causes the nodes in a view with nesting, such as a tree, to be automatically expanded according to some task context slice. For example, Mylar's extension to the Package Explorer has a mode where it keeps all of the

interesting elements expanded. The text folding mechanism is analogous to the tree expansion management but works for text editors that support folding (Section 3.3.1).

The three application window management mechanisms listed in Table 3.5 enable the views and editors visible in the application window to match those relevant to the context. When a task is activated, Mylar automatically opens all of the views that were last used when the task was active (perspective management), opens editors for all interesting elements (editor management) and applies the view focusing mechanisms to each view (view management)[36]. As the user works with editors, other editors for elements that have sufficiently decayed in interest are automatically closed. When a task is deactivated, all editors are closed and views are returned to their original configuration. The goal of each of these mechanisms is to focus the application on showing the task context, without requiring manual effort from the user.

3.2.3. Task Management

To properly associate task contexts with tasks, we required a means for a user to specify which tasks are being worked on and when that work occurs. The goal of this integration is to ensure that users work with tasks within the monitored application, such as Eclipse and Mylar, where their interaction is monitored and contributed to the task context. We implemented the Tasks UI to support task management for Eclipse. This UI supports both programmers working with bug reports, and personal tasks used by both programmers and other kinds of knowledge workers.

The Tasks UI includes the Task List view, shown in Figure 3.3. This view allows users to create, organize, edit and share the tasks on which they work. The Task List allows tasks to be created, and their description, priority, personal notes and scheduling information to be edited. Tasks worked on previously can be recalled either by browsing or querying their description or other attribute such as completion date. Since teams often use a shared repository for tasks, the Task List connects to various task repositories, which support the sharing of tasks. For example, for programmers, Mylar 0.7 integrates tasks stored in the Bugzilla, Trac, and JIRA[37] web-based repositories. These are used by programmers for working with bug reports and feature enhancements. To enable monitoring of all of a user's interaction with tasks, Mylar's task management support allows the properties and comment threads on tasks to be edited directly from within Eclipse. This facility ensures that when a programmer works with a bug report all interactions are monitored, such as interactions between the structure of the system and the task (e.g., navigating a link

---

[36] The view management and perspective management mechanisms were added after the Mylar 0.3 release used for the programmer field study (Section 4). All mechanisms were available in Mylar 0.5.

[37] http://www.atlassian.com/software/jira [verified 2006-10-02]

from an exception stack trace in a bug report to the corresponding Java element). Tasks can be organized via categories, or by using queries over a shared repository. In Figure 3.3, the queries and repository tasks are identified by the small blue server image that is overlays the bottom of their icon.

To support task context, the implementation's task management needs to enable a mechanism for the user to indicate which task is currently active. We use a "radio button" style toggle for this indicator (Figure 3.3, task with bold font[38]). This task management integration indicates to the task context model which interaction history events to associate with which task. If more than one task is activated, the composition operation associates the interaction events with each task that is active. Task activation can be streamlined further; for example, the opening of a task can automatically trigger the activation[39].

For the purpose of our validation, the task management facilities had to be of sufficient fidelity for participants to use them for their daily task management. Additional details of the task management that we implemented are available in the Mylar user documentation [36]. Monitoring interactions with the tasks themselves allows the task activity context (Section 2.4) to be used for focusing the Task List to reduce information overload when working with large numbers of tasks (Section 5.5.6).

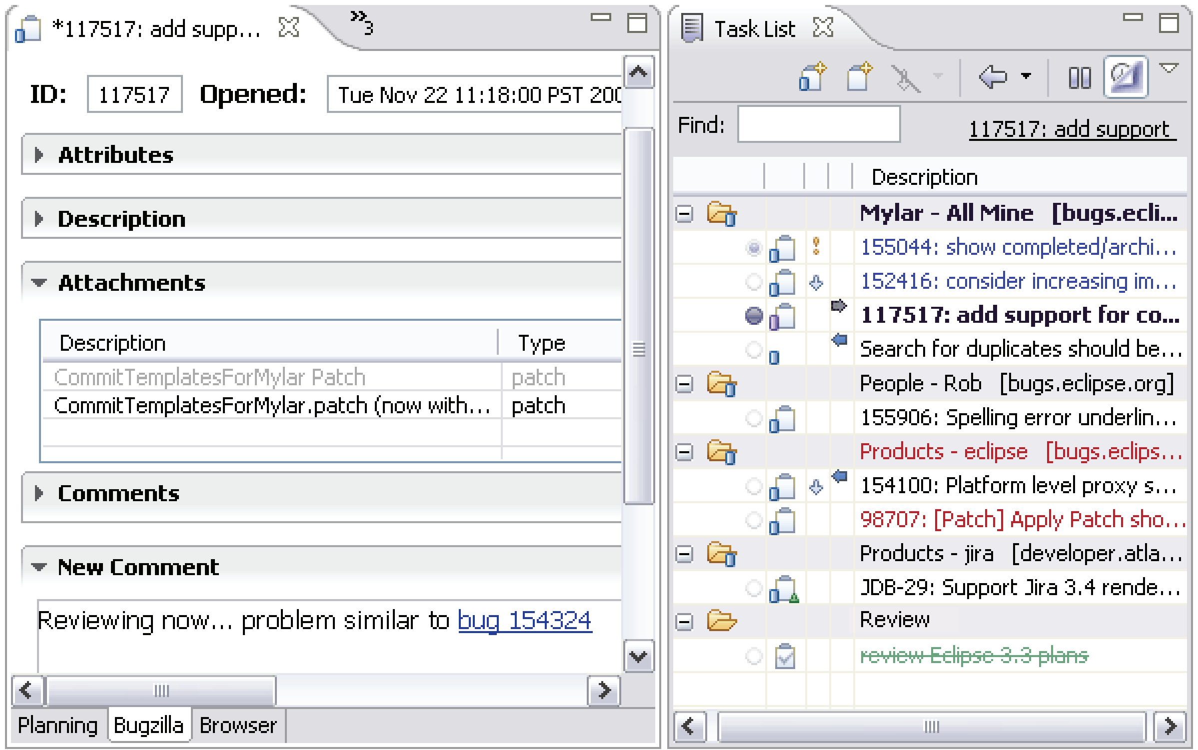

Figure 3.3: Task management facilities for activating and editing tasks

---

[38] Categories and queries are also decorated bold when they contain a task that is active.

[39] This is a user-configurable option in our implementation, turned off by default in order to match the expectation task opening with that of file opening.

3.2.4. Interaction History Storage

A storage mechanism is required to enable persistence and recall of task contexts. Since the interaction history particular to each task encapsulates all of the information that is needed to derive the task context, the task context model itself is not persisted. Instead, when a task is re-activated, the corresponding interaction history stream, stored as an XML file, is re-processed. We also store a single XML file of the same form for the task activity context. In memory, we maintain the meta-context and a single composite context that allows any number of task contexts to be loaded concurrently. As the user works, interaction events are appended to the corresponding event stream, which is periodically externalized as an XML file. If more than one context is active, the interaction events are distributed equally among the streams (Section 5.1.1). The approach of storing only the interaction insulates the storage mechanism from the implementation, algorithm, and processing method. It provides flexibility for operations on task context, as discussed in Section 2.3.

Early benchmarks indicated interaction history file sizes of 1-10MB for a full workday of interaction. However, given our field study data we estimate that programmers working full time should generate only around 1MB of interaction history information per month. The difference comes from the large redundancy in interaction histories that result from repeated interaction with the same elements. To address this redundancy, Mylar's persistence support can *collapse* the interaction history for any context. Collapse can be lossless if it uses run-length encoding, or lossy if it produces aggregate events for all interactions of one kind with a single element. On average, the latter reduces file sizes by ten times. Our remaining size differential comes from storing interaction histories as XML text files, and text compression yields another ten times file size reduction. As an example, the largest of the 1,166 task context files in the author's task context store is 204KB. The result is that even though a large amount of interaction may need to be read for very long-running tasks, the performance of activating a task is dominated by the time that it takes Eclipse to open editors, not the time that it takes to load the context.

3.3 Mylar IDE Implementation

To support investigations into the effect that an explicit task context has on programmer productivity, we needed a high-fidelity integration of task context with an IDE. Mylar IDE implements the task context model for the Eclipse IDE, enabling programmers to work in a task-focused way in every commonly used[40] part of the Eclipse IDE. We provide a thorough description of all Mylar IDE features in the 0.7 and

[40] We report Eclipse usage statistics elsewhere [52]. These results are unrelated to this thesis, but provide data that indicates the most commonly used parts of the Eclipse IDE.

later versions elsewhere [36] [37]. In this section we describe how we integrated the task-focused UI mechanism with the Eclipse IDE and overview the features that were present in the Mylar 0.3 field study release[41] (Section 4).

3.3.1. Task-focused UI for Programming

To focus the IDE's user interface on the elements relevant to completing the programming task, we project the task context onto the IDE's structure views. Decoration is supported by every structure view that shows Java elements. Interest filtering is supported by the Eclipse IDE's Package Explorer, Project Explorer, Navigator, Document Outline, Debug, Search, Members, Types, Problems, and Tasks views. Focusing these views involves adding DOI-based decoration, filtering, automatic expansion, ranking, folding, and editor management (Table 3.5).

When browsing, editing, or debugging Java programs, Mylar IDE decorates elements in any currently active task context with their interest level. This decoration is visible in the Search view (Figure 3.4 bottom right) where the gray elements are uninteresting, black elements are those with a direct interest, and bold elements are landmarks. This decoration scheme indicates the DOI of elements consistently regardless of the view in which they are displayed.

Filtering has the effect of removing the uninteresting elements from view and works consistently for views that show the elements directly, such as the Package Explorer (Figure 3.4 top-left), and those that show proxies for the elements, as in execution stacks in the Debug view (Figure 3.4 top right). Filtering is toggled using the right-most "Focus on Active Task" button visible as the rightmost icon on the toolbars of those views.

[41] Mylar 0.1, described in Section 4.2, provided only task-focused UI mechanisms and no explicit support for tasks. Although it was available with Mylar 0.3, we do not describe the working set management integration here. Working sets are user-defined groups of files, folders, and project. The facility was not used either due to a cumbersome UI or to the fact that Mylar already provides a lighter-weight version of working sets. Its role was to define a scope based on interest in a way that integrates with Eclipse's working set management. Conceptually this is identical to the change set management integration described in Section 3.3.5.

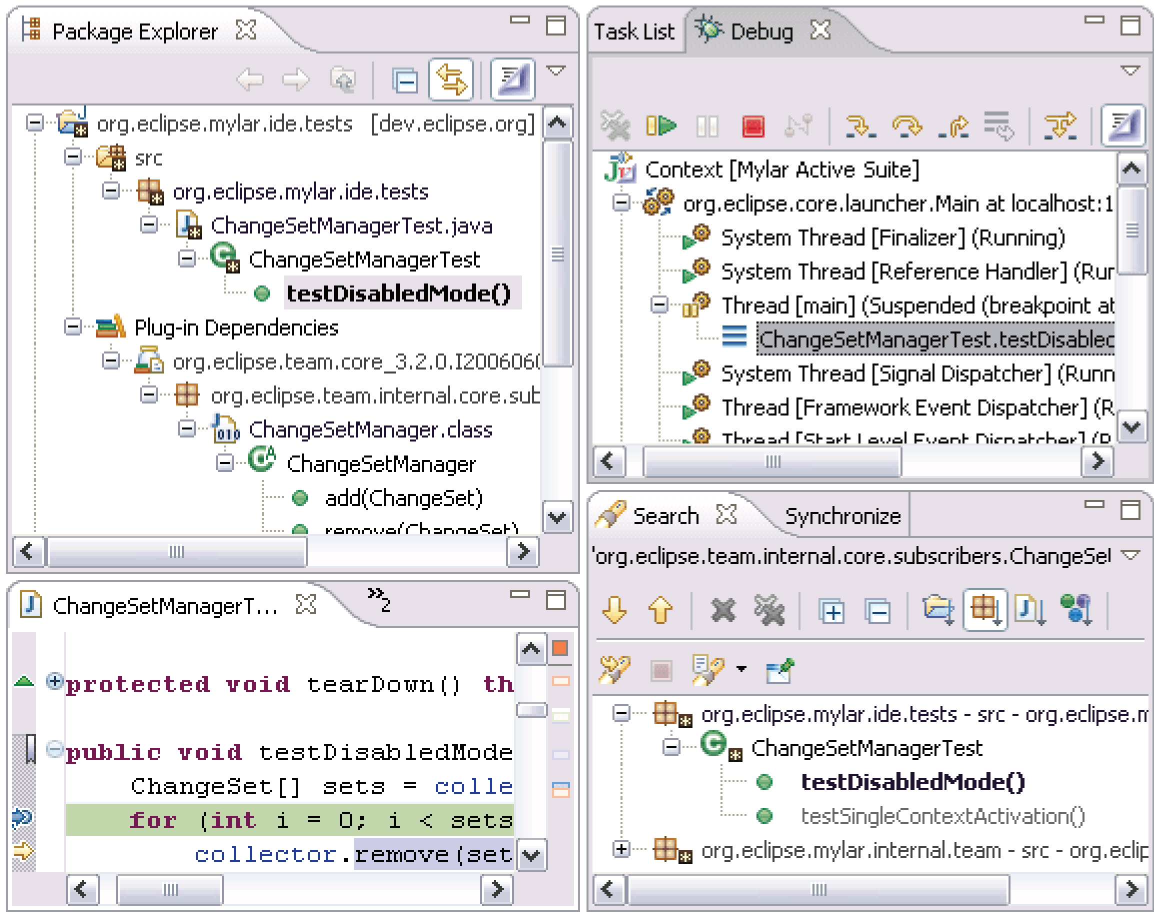

Figure 3.4: Decoration, filtering and ranking of Java elements

Interest filtering and decoration work for all of the Eclipse IDE views that are used to navigate system structure, not only for views that show Java elements. If a view shows a file, such as the Project Explorer, interest filtering works whether the file is structured and has contents for which a structure bridge is registered, or whether the file is unstructured, as in the case of the "refresh.gif" image file (Figure 3.5 top left). For structured files, the Outline view, which provides a structural view of the file being edited, filters all uninteresting XML elements from the build.xml file (Figure 3.5 bottom left). The Problems and Tasks views, which show static checker or annotation-determined items in a list (Figure 3.5 bottom right) can also be filtered to show only the items relevant to the task context.

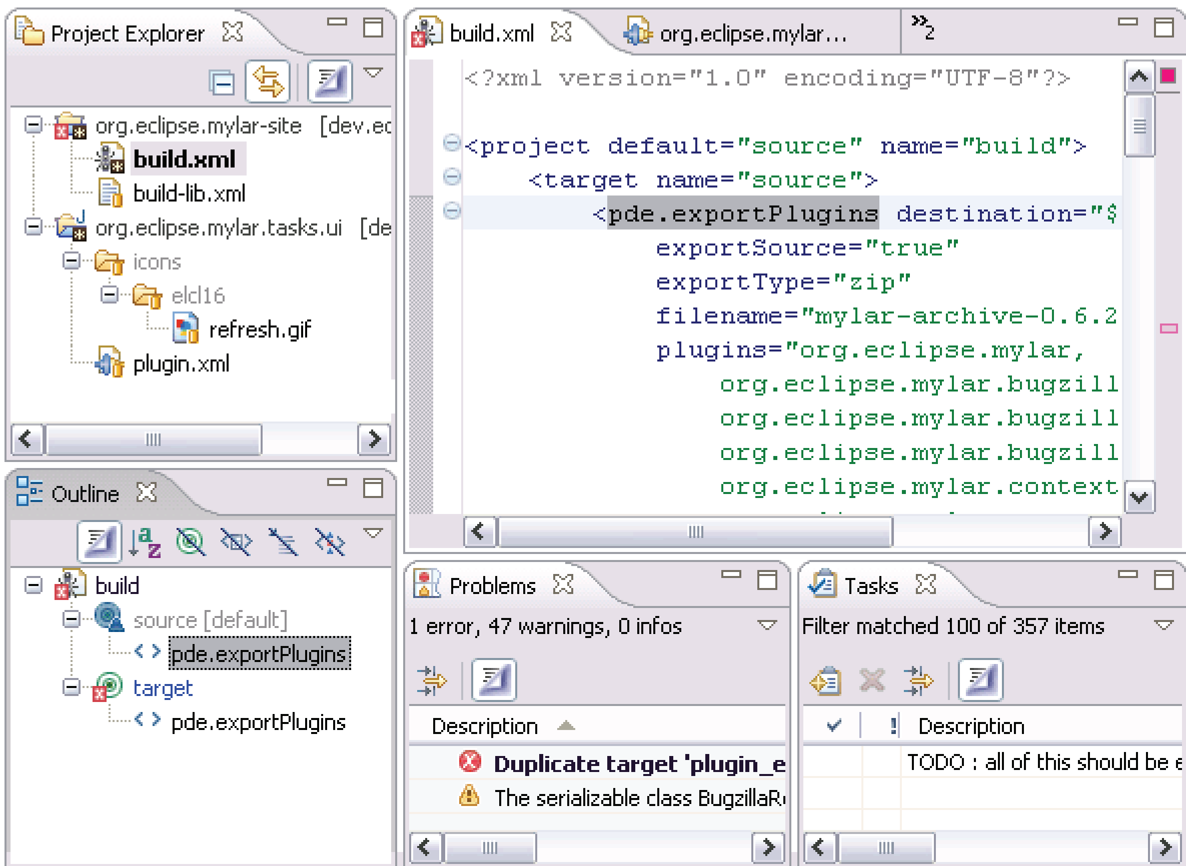

Figure 3.5: Decoration, filtering and ranking of various program artifacts

Tree views support the automatic expansion mechanism. For example, when interest filtering is on, the Package Explorer always has all of its nodes expanded to avoid the user needing to repeatedly expand and collapse nodes. Auto expansion is always enabled when the view is filtered, meaning that the user cannot manually collapse the tree. Removing this ability from the user is acceptable because decay ensures that interest-filtered views do not show a scrollbar when working on a typical task and when used on a sufficiently large display (e.g., 1000 pixels tall and higher).

The task-focused UI mechanism that corresponds to filtering and automatic expansion for editors is automatic DOI-based text editor folding. For example, when editing Java code, Mylar makes task context explicit by automatically unfolding the interesting elements and folding all uninteresting elements (Figure 3.6).

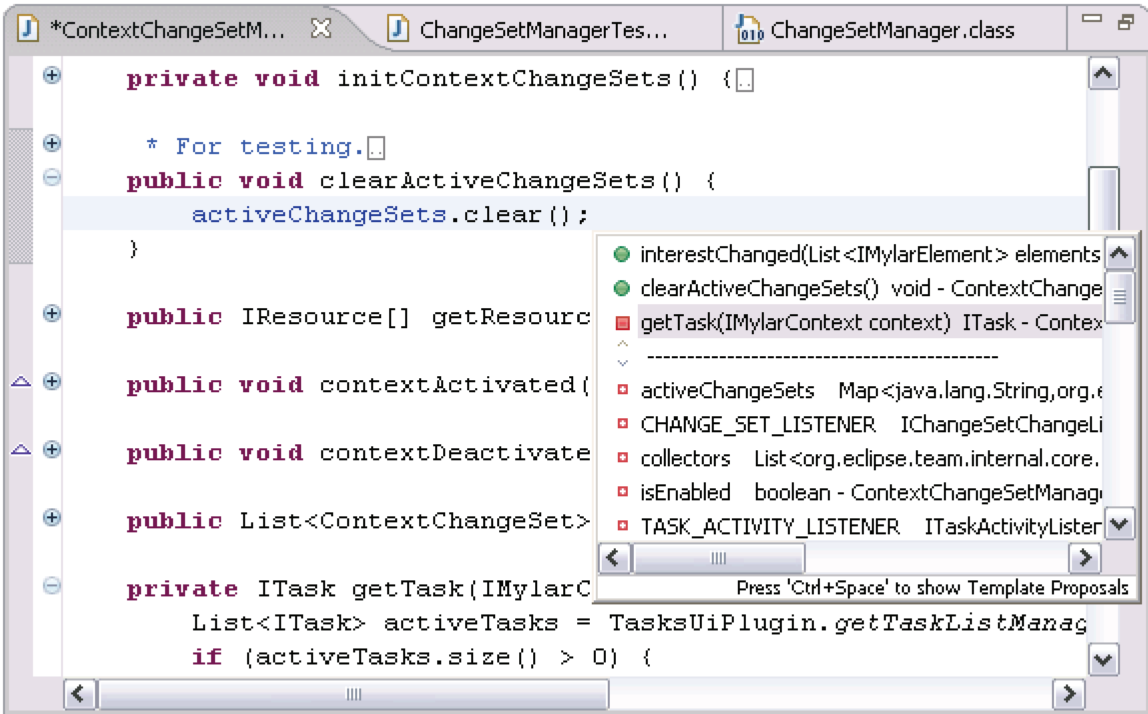

Figure 3.6: Automatic folding and content assist ranking in Java editor

Interest ranking can be applied to any list that orders elements by sorting the view on the DOI values of elements and relations. Many views in the IDE also provide their own ranking. To separate task context ranking visually from other rankings, such as those based on structural heuristics, we insert a separator. For example, Eclipse's content assist provides a ranking of suggested completions in the editor based on Java heuristics. Mylar projects DOI values onto that ranking, adds a separator and moves the elements that are in the slice of all interesting elements to the top of the list above the separator. The elements above the separator are in DOI sorted order (Figure 3.6). The elements below the separator remain in the order of the existing structural heuristics (e.g., Java scoping rules).

To support the editor management mechanism, when a task is activated, the focusing mechanisms are applied automatically to every view, open editors are closed and all of the editors for the interest files in context are opened. Elements that decay out of the slice of all interesting elements have their editors automatically closed as the programmer works. When a task is deactivated, all editors are closed.

3.3.2. Inducing Interest

The induction operation allows Mylar IDE to display elements that have not yet been interacted with in a view. For example, when a method is selected in a filtered view, propagation events cause each parent of the method to become interesting so that all of the parents, up to the project containing the method, show in the view. Another propagation that we support is the interest of errors. For example, if the programmer is working on a class, "Foo", and accidentally changes the name of the class to "Fooo", all of the classes

that use it will break. To make the programmer aware of this, interest propagates to each of the references to the class, causing it to appear in the view (Figure 3.7). Only elements with a direct interest are decorated with a black text font. Since the programmer has never directly interacted with any of the other elements, such as “baz” or “Jazz.java”, these elements appear in gray.

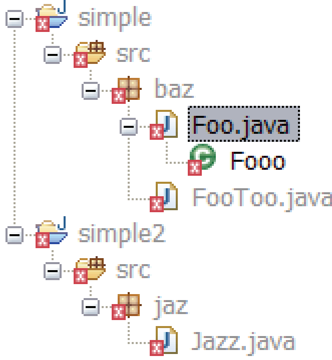

Figure 3.7: Propagated interest of errors in the Package Explorer

Another instance of interest propagation is made visible by the Active Hierarchy view (Figure 3.8). This view projects a slice that includes all Java classes with a DOI exceeding the landmark threshold, as well as the superclasses of those classes. This induction is similar to the propagation that occurs when a file is selected, but propagates along the inheritance hierarchy instead of the containment hierarchy.

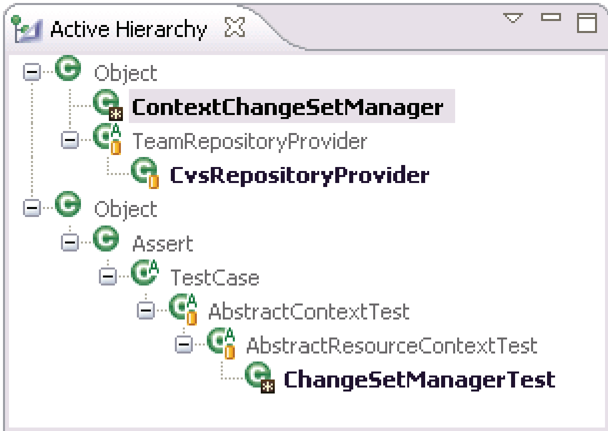

Figure 3.8: Propagated interest of superclasses in the Active Hierarchy view

3.3.3. Active Search

The Active Search facility uses the induction operation to compute the predicted interest of elements and relations, thereby expanding the task context. The input for Active Search is a slice of the active task context for elements with a DOI value over the landmark threshold. The scopes that Active Search uses are slices of the task context model; in the UI we refer to the different kinds of scopes as *degrees-of-separation*. This term is indicative of the distance from the highest interest elements. We define degrees-

of-separation by both the DOI level and by the containment relations. As a context grows, lowering the degree of separation is akin to tightening the search scope to decrease the number of results. To focus results, the Active Search view[42] uses the same ranking and decoration mechanism as the other views. Figure 3.9 shows the degree-of-separation scopes supported by Mylar IDE, each of which corresponds to a slice of the active task context.

1) Landmark elements: the most interesting elements
2) Interesting elements: all elements above the interesting threshold
3) Interesting files: all interesting files (often interesting via propagation)
4) Interesting projects: all interesting projects (usually via propagation)
5) Interesting dependencies: all interesting project dependencies (usually via propagation)

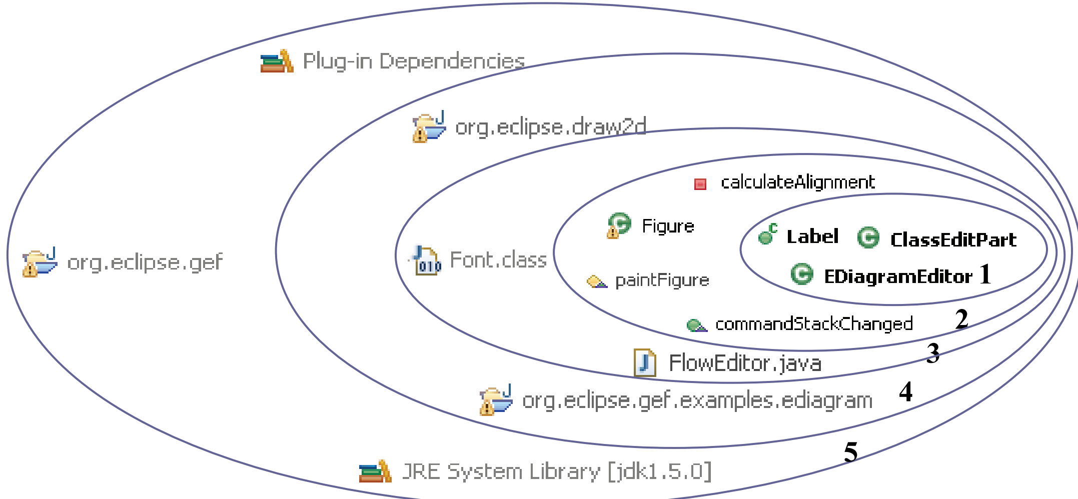

Figure 3.9: Degrees-of-separation for Active Search

The Active Search view shown in Figure 3.10 displays a projection of all landmark elements and their relations to elements that are interesting, whether the interest of those elements results purely from prediction or from direct interaction. When configuring a view to take a particular projection of the task context model, an important decision is whether or not the corresponding slice should include elements of predicted interest. As these can be high in number, we do not show elements with predicted interest in the

---

[42] Active Search is a facility that provides the induction operation to propagate predicted interest. However, unlike all the other operations we describe, which always run whenever a task is active, neither the Active Search nor the Active Hierarchy operations are enabled unless the corresponding view is active. Although the memory and performance overhead of these features is minimal, we wanted to ensure that they could be turned off on very slow machines or in the case where users encountered bugs.

standard Eclipse views, only those with direct or propagated interest. The Active Search view is the only exception as it displays a slice that includes elements with predicted interest.

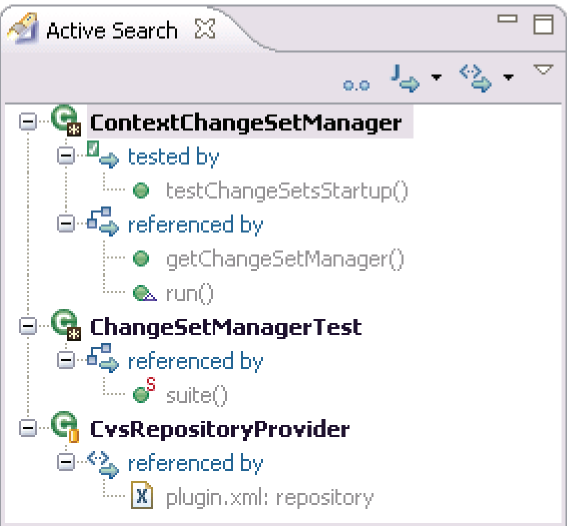

Figure 3.10: Elements and relations with predicted interest in the Active Search view

3.3.4. Context Test Suite

Eclipse provides an integrated JUnit[43] facility, used for running unit test suites. A unit test is defined by being a subclass of `junit.framework.TestCase`. When using Eclipse, the programmer will specify a test suite or collection of test suites to run. On a large system, such collections can involve a large number of slow-running tests. The context test suite uses a slice that gathers all subtypes of `junit.framework.TestCase` in the task context. Since elements of a predicted interest are part of the context, this includes all test cases interacted with directly and those that have a predicted interest because they are structurally related to elements with positive DOI. For example, when editing a method, the method can become a landmark, which causes the Active Search facility to find all unit tests for that method; these tests become a part of the task context. Running the context test suite is identical to running a regular test suite (Figure 3.11), but does not require the programmer to indicate which test types should be run.

[43] http://www.junit.org [verified 2006-10-02]

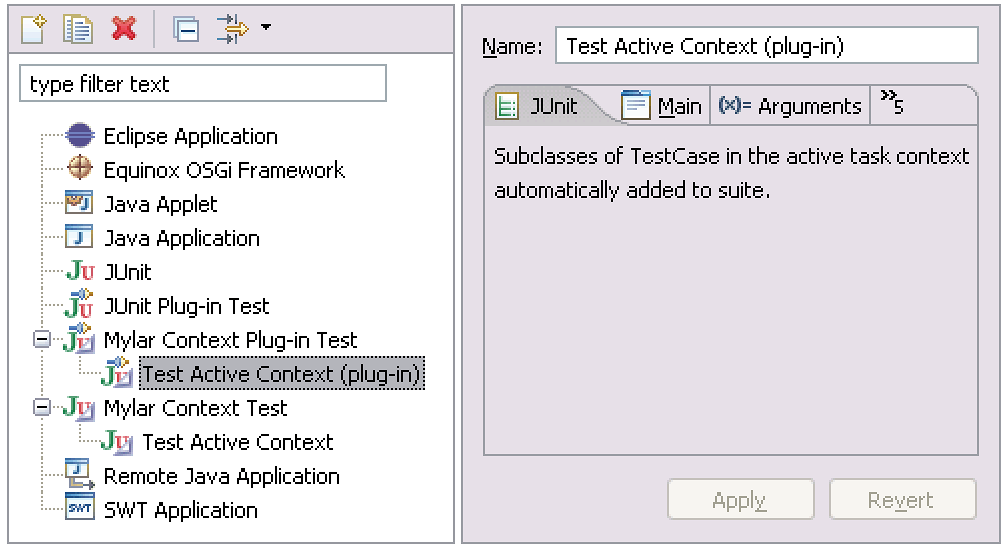

Figure 3.11: Automatic context test suite

3.3.5. Context Change Sets

In Eclipse and other IDEs, change sets are groupings of outgoing or incoming revisions to files. A programmer can set up change sets manually to perform revision operations (e.g., commits, updates) on parts of the system rather than on the whole system. Mylar IDE's automatic change set management creates a change set for each activated task and populates that change set automatically with a slice corresponding to all of the files modified as part of the activity associated with the task. These slices differ from other slices presented by the task-focused UI because they ignore decay and because they persist when a task is no longer active. For example, if outgoing changes remain when the programmer switches to a new task, the previous task's context change set will continue to contain the outgoing changes until they are committed (Figure 3.12 left). This support enables the programmer to switch between multiple tasks without committing the code modified as part of the current task before switching to another. As with other facilities that are automated by task context, the change set management removes the burden of manual change set creation from the programmer. The explicit association between change sets and tasks also supports automatic commit messages, which can then be used to navigate back to a task from a commit of the context change set, as visible in the History view (Figure 3.12 right).

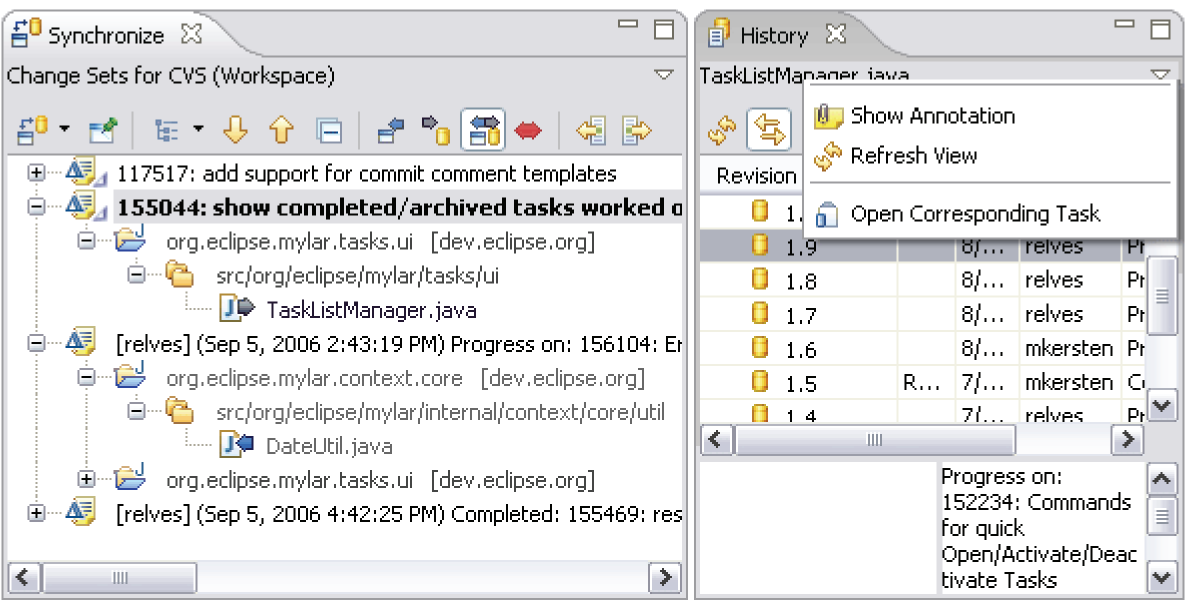

Figure 3.12: Automatic context change sets

3.4 Mylar Browser Implementation

To validate task context for the more generic knowledge work domain, we needed an implementation of Mylar that supported the resources commonly accessed by knowledge workers: files and web pages. Email is another medium used by knowledge workers for defining work items, and approaches such Taskmaster [5] have demonstrated that tasks can be made a more primary part of the email inbox. In order to make tasks the central mechanism for interaction, our approach is to enable collaborations around tasks and task context via the Task List (Section 5.3) rather than integrating task management into existing tools such as email readers.

Unlike Mylar IDE, which is integrated with the Eclipse IDE, the Mylar Browser is a standalone application composed of the parts of the Eclipse platform that support working with files and Eclipse's embedded web browser. We created a new view, called Resources, which supports the display of files, folders and web site URLs[44]. The task management and task-focused UI features were identical to those of the Mylar IDE. Unlike the Mylar IDE, which provides additional task context integration facilities such as change set management, the Mylar Browser is a simpler application that provides only task-focused UI integration. The task context model, operations, scaling factors, and thresholds, and all other parts of the Mylar frameworks were unchanged in Mylar Browser.

[44] The Mylar Browser is an Eclipse Rich Client Application that uses the file management features from the Eclipse Platform, but excludes any programmer-specific features: http://eclipse.org/rcp [verified 2006-10-02]

The key difference in the integration of task context with a browsing tool is the difference in the structure of information that knowledge workers view and create. A program has an explicit primary structure. The task context model in Mylar IDE is aware of interaction with the program's explicit primary structure down to the commonly used and fine-grained member level of granularity[45]. In contrast, the documents that knowledge workers use often have a less apparent structure.

3.4.1. Navigation Support for Files and Web Resources

To support users navigating with files and web resources, we created a Resources view (Figure 3.13). This view allows the user to link in any folder from their file system or from a network drive (e.g., the "My Documents" folder). Each folder linked is a root node in the tree. By default, the tree is in focused mode, showing only interesting resources in the current task context. An additional root node called "Web Context" contains all of the web pages in the task context.

3.4.2. Capturing Interaction with Files

Knowledge workers browse and edit a variety of files. Some of these have a well-defined structure that is understood by Mylar (e.g., XML documents); others have a structure, but the structure is not understood by Mylar Browser (e.g., Microsoft Word documents with sections); and finally some do not have a well-defined structure all (e.g., images). Mylar Browser monitors selections down to the file level of granularity. The more a file is selected the more interesting it becomes (i.e., toggling between two visible editors will increase interest identically to how it would when toggling between elements visible in a structure view in Mylar IDE). Mylar Browser also understands directory structure, and propagates interest to directories containing the file when a file is selected. If an Eclipse-based editor is available for the file, selections in the editor are also captured. If the Eclipse-based editor supports modifying the resource, and Mylar supports the editor, edit interaction events are also captured.

Since many of the documents that knowledge workers edit are Microsoft Office based and not Eclipse based, we needed to ensure that interaction with these editors was monitored sufficiently. To support this requirement, we relied on the SWT [46] toolkit's ability to embed Microsoft Office documents. Implementing a structure bridge specific to the contents within Microsoft Office or other semi-structured documents is technically possible and would result in additional selection and edit events for parts of a

---

[45] For Java, this is members such as classes and methods, for XML it is elements. The same level of granularity is commonly shown in IDEs' structure views.

[46] http://eclipse.org/swt [verified 2006-10-02]

document, such as outline section headings. We did not implement this bridge as we did not require this level of support for our validation of the Mylar Browser (Section 4.4).

Figure 3.13 shows the author's workspace with the Mylar Browser when working on this dissertation. The Task List is identical to that of Mylar IDE and shows both shared tasks (those with 'incoming arrows') and personal tasks, such as "book FSE trip". The editor pane shows an embedded Microsoft Excel document and the Resources view shows the files and pages that are part of the task context.

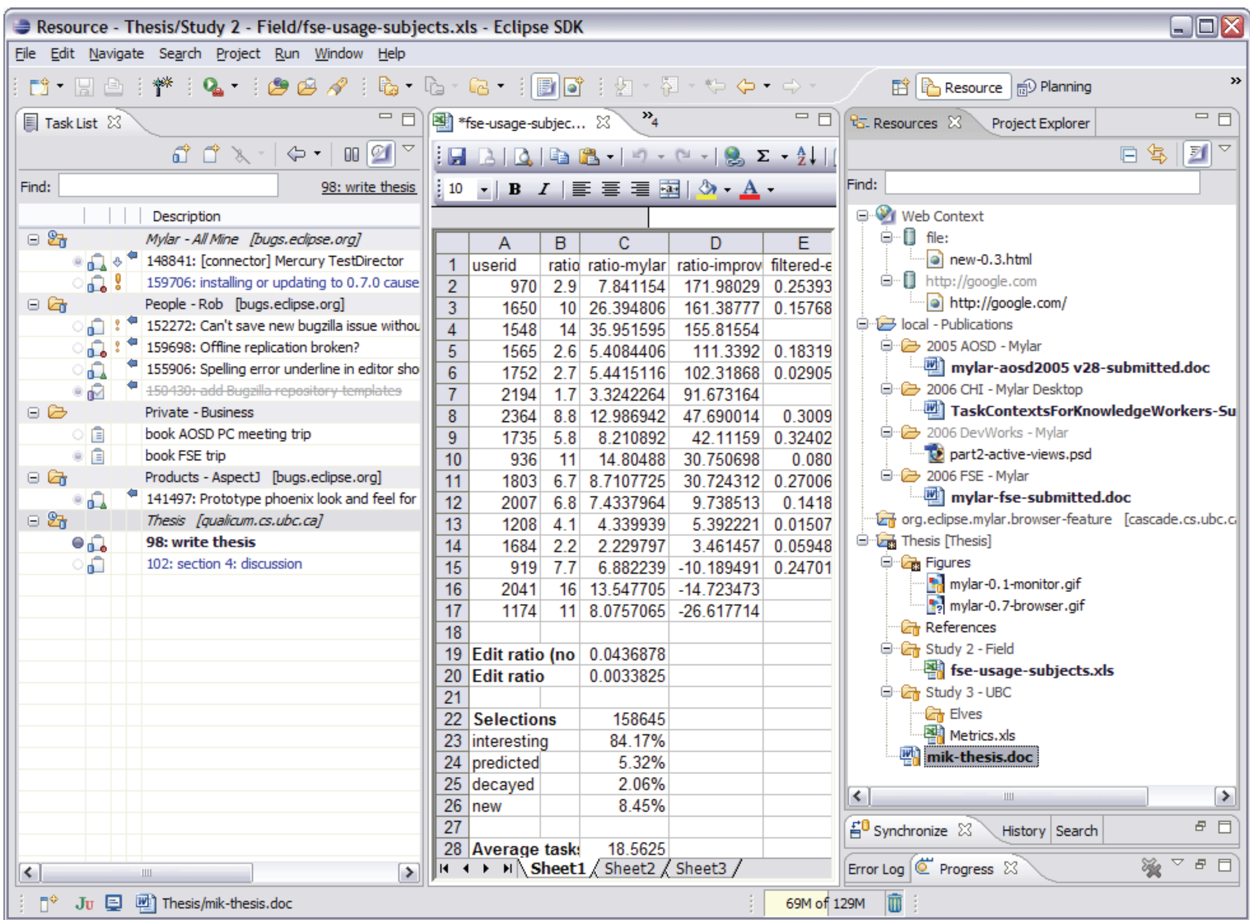

Figure 3.13: Mylar Browser showing Task List and embedded Excel document

3.4.3. Capturing Interaction with Web Resources

To support interaction with web resources, we needed a mechanism for monitoring web browsing activity. To maintain a similar level of granularity to file navigation, we chose the URL[47] as the element of interaction. The monitor was extended to capture successful retrieval of URL contents for any click

---

[47] http://www.w3.org/Addressing/URL/Overview.html [verified 2006-10-02]

made within the browser embedded within Eclipse (Internet Explorer on Windows[48] or Mozilla on Linux[49]). The only structural relations that we infer form URLs is the web site from which the relation originates. The result is that the task context capture and display used for web documents looks and feels consistent to that used for files.

Unlike its abstractions for working with resources such as files, the Eclipse Platform provides no support for handling or displaying web documents. In this respect, the integration of web resources differs from other kinds of files: when a task context is active, the Resources view shows all of the web resources in the context. When a task context is not active, no web resources can be shown because they only exist in the context[50] (i.e., are stored in the interaction history).

3.5 Implementation Summary

The primary goal of our implementation was to support the study of how the task context model supports programmers and other kinds of knowledge workers. We implemented a generic task-focused UI and task management facilities to support both kinds of users. We integrated Mylar IDE deeply with almost every part of the Eclipse IDE, to ensure that the entire user experience when programming with Eclipse could be re-focused around task context. This involved using each of the operations defined in Section 2.3. We then extracted the parts of Mylar IDE that were generic to working with files, added support for web browsing, and, without altering any of the task context model or task-focused UI, created the Mylar Browser. In the next chapter, we discuss the validation of task context via field studies performed on these two applications.

---

[48] http://microsoft.com/windows/ie [verified 2006-10-02]

[49] http://mozilla.org and http://linux.org [verified 2006-10-02]

[50] An alternative approach would have been to store all of the web navigation history in the user's workspace, and then to use a filtering mechanism similar to files. This kind of global persistence of web navigation information was beyond our validation needs. However, we did have to implement a cache for all page titles in order to ensure that context activation did not require waiting for frequent round-trips to the server.

4. Validation

To validate our thesis, we needed to test how the use of task context impacts the productivity of programmers and to test whether the task context model generalizes to other kinds of knowledge work. In this section, we describe two field studies we performed of programmers using task context for working on well-structured software systems and a third field study that involved a more generic class of knowledge workers using task context for a broader set of less structured artifacts.

Our first field study was a preliminary feasibility test of task context involving the Mylar 0.1 prototype, which added the task-focused UI and a single task context model to the Eclipse IDE. This study involved six professional programmers at IBM who were willing to try Mylar 0.1 for one week [39]. The data from this study showed that programmers voluntarily used and liked Mylar 0.1 (Section 4.2). In addition to gathering qualitative data, we also measured the impact that Mylar 0.1 had on the IBM programmers' productivity. To measure the effect on productivity, we devised a simple metric called *edit ratio*, the proportion of edits over selections. Although the number of subjects was too small to yield statistical significance, the usage data from the IBM study indicated a promising average increase in the edit ratio of the participants (Section 4.2.4). The study results also indicated that task management and a better integration of task context with Eclipse was required for the programmers to use Mylar on an ongoing basis.

Our second field study of programmers involved the use of Mylar 0.3. This version of Mylar supported task context for multiple tasks, provided a view for switching between tasks, and integrated a Bugzilla client to support programmers using Bugzilla repositories for their task management. To recruit study subjects we presented a preview version, Mylar 0.2, at the EclipseCon 2005 conference[51]. We then released a standalone Mylar Monitor (i.e., no Mylar UI components) to the 99 programmers who signed up for our study following EclipseCon. We measured the edit ratio of the programmers in the study during approximately two weeks of full-time programming. After they passed a pre-determined threshold of activity, we allowed programmers to install Mylar 0.3. We required approximately three weeks of full-time programming with Mylar to accept their usage data in the study (Section 4.4.2). At the end of the study, we had sixteen accepted subjects. Our analysis of this usage data showed a statistically significant improvement in the edit ratio of these subjects (Section 4.3.3).

[51] http://eclipse.org/mylar/publications.php [verified 2006-10-02]

The first two studies tested the task context model on programmers working with software systems, which have a clear and explicit structure. To test whether task context supports other kinds of knowledge workers who interact with less structured information, we deployed a field study of the Mylar Browser, which supports task contexts for file and web browsing activities. The Mylar Browser uses the same task context model, algorithms, and task-focused UI facilities, but does not include programmer-specific features. The study involved eight participants with a broader range of professions. All of the participants used Microsoft Windows, Microsoft Office and web browsers for their knowledge work activities. We asked them to try to use the Mylar Browser to work with these applications. The results of the third study demonstrate that the task context model generalizes to supporting file and web browsing activities (Section 4.4). The usage data indicates that when used for file and web browsing the task context tends to represent a small subset of information from the large set of information available on shared document repositories and on the internet. We conclude that our interaction-based frequency and recency based weighting was effective at showing the relevant information to knowledge workers authoring and browsing documents and web pages. We also found that a key component of our approach for these works is the decay mechanism that prevents views of task context from becoming overloaded.

4.1 Methodology

We chose to validate the task context model through field studies because validating the claims made about this model required observing the effects of the model on the performance of knowledge work. Knowledge work typically occurs in a demand and deadline-driven, collaborative, and multi-tasking environment [26]. We did not believe that we could reproduce this environment with sufficient fidelity with time-constrained and mocked-up laboratory studies. In addition, we did not have evidence that the inherently non-expert student subjects who were available to us for longer-running laboratory experiments would provide a sufficient approximation of experts working in the field. Finally, we wanted to ensure that we tested the effects of subjects' voluntary use of task context for daily work over the course of many days and weeks, not just over the time-constrained period of a laboratory study. As a result, the field studies on which we report involve monitoring subjects in their own work environment. The monitoring facilities we designed and used allowed us to interpret the activity of the workers without affecting their work process and without requiring them to send company-private information[52].

For each of the three studies described in the following sections we describe the methodology used (in the Method and Study Framework section for each study). We also describe any shortcomings or factors that could affect the validity or accuracy of our results (in the Threats section for each study).

---

[52] The Mylar Monitor uses a one-way hash function to obfuscate the handle identifiers of elements.

4.2 Study I: Programmer Feasibility Study

Our first study sought to gather data about how programmers might use task context and whether programmers liked working with an explicit task context.

4.2.1. Subjects

The participants in our study were six senior IBM Toronto Lab programmers working in Eclipse 3.0 on projects involving WebSphere[53], XDE[54] and Eclipse plug-ins. 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 the intern's data in the results we report; the results presented are for the six experienced professional developers. Before the study, subjects were given a questionnaire asking about their experiences and problems using Eclipse (Appendix A). The problems cited included a dislike of the way in which editors and files were handled, and overpopulation of tree views, such as the Package Explorer. Table 4.1 lists the answers that the subjects had to the question of "gripes that you have with Eclipse's current support for your navigation needs". The responses of subjects 1, 2, 4 and 5 are indicative of the problem of information overload and the need to scope down the amount of information presented.

Table 4.1: Subject comments related to information overload in Eclipse

Subject	Answer
1	"I wish the content in the navigator view and the package explorer view can be more condensed, e.g. reducing the vertical spaces between each package. So that the more [*sic*] information can be displayed in a view with the same size."
2	"I like to collapse the tree that I have finished looking at and only expand those that I need at the moment. It may be complicated if I have many Java projects in the workspace…. User has to filter out unwanted files explicitly."
3	"I am a simple user, does not like fancy stuff. just basic functions are fine with me."
4	"Extremely slow. For instance, unfolding or folding the tree . Navigating between open editors using keyboard. Do quick fix using keyboard so that I don't have to change to different views"

[53] http://www.ibm.com/software/info1/websphere [verified 2006-10-02]

[54] http://www.ibm.com/software/awdtools/developer/rosexde [verified 2006-10-02]

5	"I would like to have a blank view which I could drag files and bookmarks onto which would serve as a navigator. That way I don't have too be confined to packages, projects, etc… I don't like managing the expansion state of trees"
6	"Sorry, none off the top of my head."

4.2.2. Method and Study Framework

We used a diary study format to gather feedback from the subjects [57]. The programmers were asked to use the Mylar 0.1 prototype and to provide daily qualitative reports about their experiences. The study required that programmers work with plain Java code since Mylar 0.1 only provided a structure bridge for Java. We augmented the diary study format with the Mylar Monitor's quantitative measurement by recording the programmers' activity.

For the duration of the study, a researcher was co-located 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 3.1 that included the Mylar Package Explorer, Outline, and Problems List. Programmers were given the suggestion to try, but were 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 or requiring too much time from the subjects, we provided no training and only required the programmers to read 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. The total amount of time that Eclipse was active on the programmers' machines was 25.45 hours for the three days of baseline monitoring. The Mylar Monitor provides a monitoring framework that reports some low-level summary statistics (e.g., Figure 4.1 right side) as well as exporting this usage from the programmers' desktop. The Monitor also exports this information to a log file. An additional view also allowed us to inspect the subjects' single task context model during the exit interviews (e.g., Figure 4.1 left side).

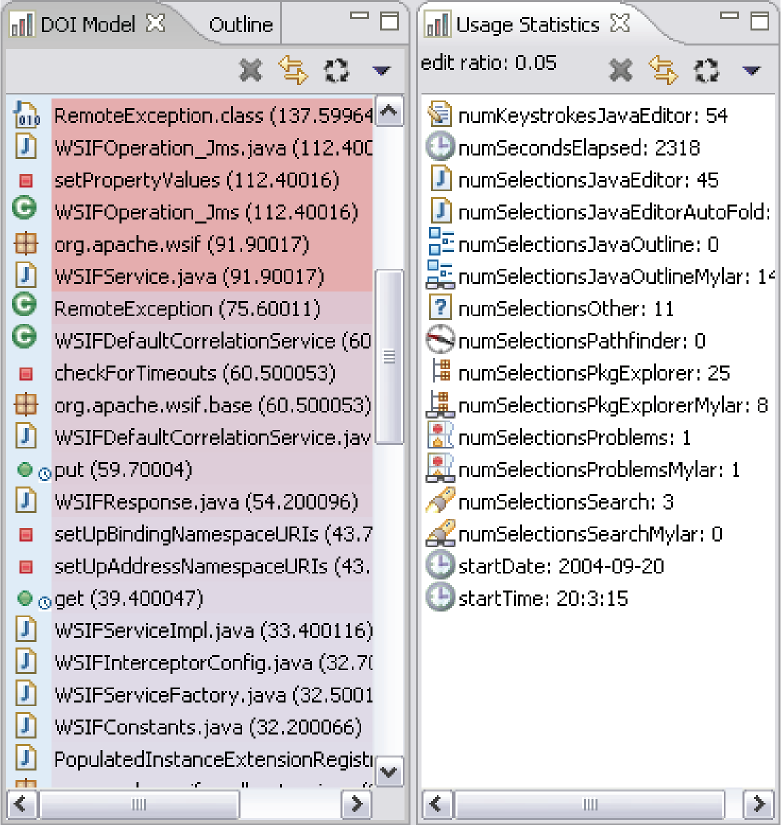

Figure 4.1: Ranking of elements, usage statistics

This study was performed using Mylar 0.1. The most notable differences between this early version of Mylar and later versions are that it supported only one task that was always active and that it did not provide task management support. In addition, the task-focused UI provided replicas of the Eclipse views rather than integrating with the existing Eclipse views and did not allow the highlighting scheme to be turned off. Figure 4.2 depicts how Mylar 0.1 presented task context for each of the three views visible in Eclipse's default Java perspective. The highlighting scheme used color to decorate elements with interest level, with deeper hue indicating a higher DOI[55]. Since Eclipse already uses highlighting to indicate the currently selected element, Mylar 0.1 used bold font to indicate the currently selected element. There are several items to note in Figure 4.2.

1) Mylar Package Explorer: Interest-based filtering is enabled, resulting in only the files and libraries relevant to the task being displayed. The number of filtered elements is indicated on the parent label. For example, the `org.apache.wsif` package in Figure 4.2 shows eight interesting files of the fourteen files in that package. The filtering mode reduces the need to manually expand and scroll the tree by

[55] The color and icon scheme metaphor in Mylar 0.1 was that of a bathygraphic map, one of the original inspirations for the task context model, where uninteresting elements were 'sunk underwater'.

actively maintaining the visibility of high-interest elements; this helps keep hierarchical relationships visible, such as the package structure. 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 problems list, but corresponding program elements are additionally displayed and used to highlight the DOI of the problem. Mylar 0.1 did not provide filtering in this view.

3) Mylar Outline: Interest filtering and highlighting shows 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.

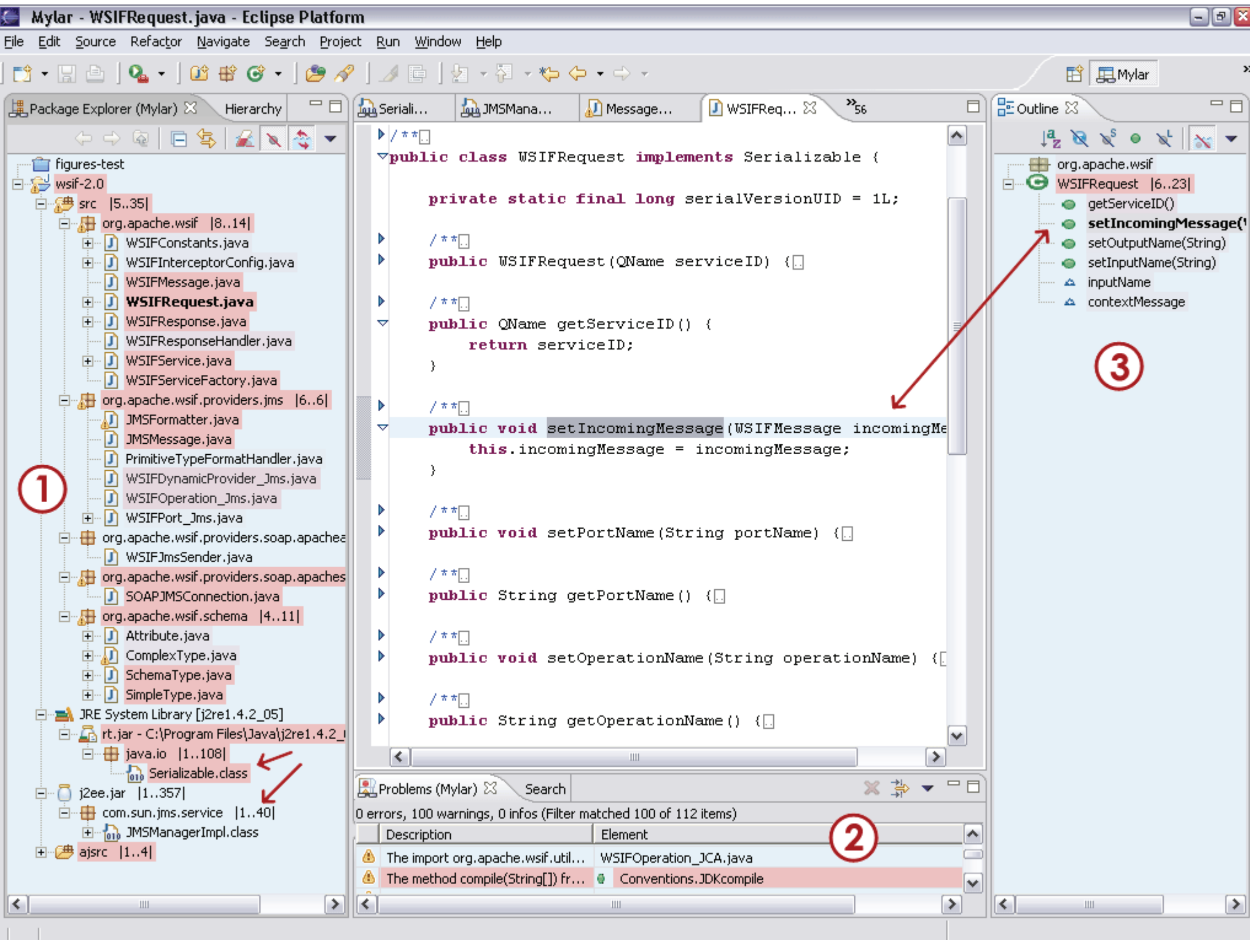

Figure 4.2: Mylar 0.1, used for IBM study

4.2.3. Results: Usage statistics

The study ran for five business days. During the study, we logged a total 56.99 hours of Eclipse usage across subjects. At the end of each day, we asked the subjects 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 in person with each of the subjects.

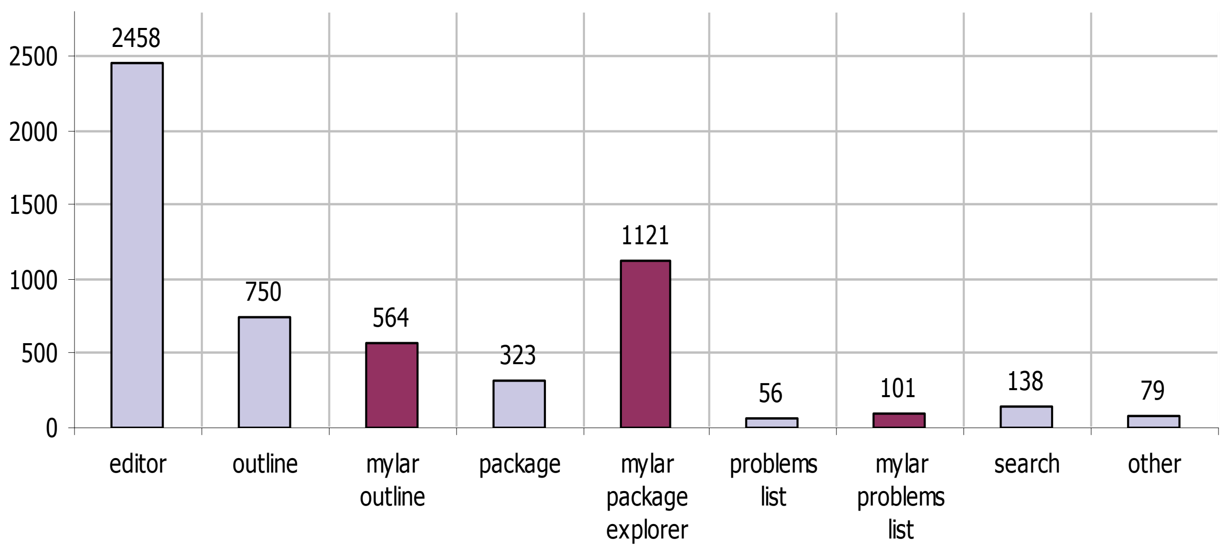

Figure 4.3: Mylar view vs. standard Eclipse view selections across subjects

For all but the Outline view, when a Mylar view was available as a replacement for a plain Eclipse view, the programmers used the Mylar view more than the plain Eclipse view (Figure 4.3). The view used most by the programmers was the Mylar Package Explorer, which is consistent with the baseline ratio of view usage where the participants used the regular Package Explorer most. 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, the subject used the Mylar Outline almost exclusively. The complete usage statistics for the week using Mylar are visible in Figure 4.3. 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.

4.2.4. Results: Edit Ratio Change

To measure any effects that an explicit task context could have on the subjects' productivity we defined the edit ratio measure. The edit ratio is the number of keystrokes in the editor over the number of structured selections made in the editor and views. 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 was encouraging. Finding a meaningful statistic of this ratio was challenging not only due to the small sample size, but also due to the short duration of the study. From the daily diary responses, we learned that several subjects 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 one half-hour in Eclipse during the baseline week). Although the sample size in this study was too small to yield any conclusive results, the 15% average edit ratio improvement across subjects was promising. Also, the single most active subject, who actively worked in Eclipse for nineteen hours during the week and accounted for 40% of the activity across both weeks, reported that she worked on similar tasks both weeks. Her edit ratio improved by 49%. During the wrap-up interview, we asked the subjects if the significant increase in the edit ratio was consistent with their impressions. All of the subjects agreed, stating that they did not need to navigate or search for elements as much as they did with the plain Eclipse views.

4.2.5. Results: Model Feedback

All of the subjects reported that the task context model accurately represented the information relevant to their work. During the wrap-up interviews, we showed them the hidden DOI Model view (Figure 4.1) 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 this view for internal debugging and inspection purposes. Some programmers were surprised by the accuracy of this view and expressed interest in using it for their programming activity.

Most of the subjects stated that the predictability of the model was important to them; for instance, 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. For example, the subjects complained that to start on a new bug report they would have to clear the model, even though that model may be needed again.

Two subjects 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; the UI for manually reducing interest 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 lead to the overpopulation.

4.2.6. Results: View Feedback

Although all of the subjects liked the task context model that the task-focused UI views exposed, there was a mixed response to the highlighting scheme: three programmers liked it, one felt neutral, and 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 change neither the color used in the highlighting nor the text annotations on the elements.

The Mylar Package Explorer view was liked the most. Subjects found the automatic filtering and auto-expansion mode in the Package Explorer 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 differed too much from a typical tree view. As one example, users could not collapse nodes containing children of high interest since the collapse function was not mapped to an interest operation on the model. Contrary to our intuition, 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 structure-based filtering mechanisms.

The Mylar Problems List was also 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 liked by all the subjects because 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 and the Content Assist popup view. All of the programmers expressed interest in using future releases of Mylar[56].

4.2.7. Threats

One key threat to the results of this study is the short amount of time that the programmers used the tool. For example, a subject may have worked on only one particular task for the duration of the week, such as testing a feature. In such a case, we would not have observed a sufficient amount of that programmer's activity to determine the effect on the programmer's productivity. To mitigate this risk, we asked the programmers about the kinds of tasks on which they worked each day and found that most worked on

[56] Since we wanted to focus development effort on incorporating study results and not on supporting the Mylar 0.1 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.

various tasks. A related threat is that, for the period of one week, the programmers were willing to try the tool and have their behavior affected by using this new tool in a way that is not representative of their normal work practice. We could not address this threat in this preliminary study but addressed it in our second field study by using a study period that was several times longer. Finally, the number of subjects was too small to gather enough data for a statistically significant test of the edit ratio change. We also addressed this threat in our follow-up study by using many more subjects.

4.3 Study II: Programmer Field Study

Our initial study exposed a single primitive DOI weighting across all of a programmer's work. From this experiment, we learned that programmers need separate contexts for the different tasks on which they work and that a simple weighting of the frequency of element selection is not sufficient. Although the first study suggested our basic approach had potential, we learned it was not ready for daily use and lacked specific evidence that it improved programmer productivity.

To answer the question of whether an explicit task context improves programmer productivity, we conducted a much larger field study. Again, we chose a field study because the time-constrained tasks performed on medium-sized systems possible in a laboratory setting are not representative of the real long-term tasks performed on large systems in industry.

4.3.1. Subjects

The target subjects for our study were industry Java programmers who used the Eclipse IDE. To solicit participation, we demonstrated the Mylar 0.2 tool at an industry conference (EclipseCon, March 2005) and advertised on a web page. Early access to Mylar was only possible by signing up for our study through a web form. Ninety-nine individuals signed up for the study over the eight months between the announcement and the conclusion of the study. As visible in Table 4.2, the majority of these individuals were industry programmers, about half of them worked in organizations with more than fifty people and most identified their industry sector as software manufacturing [52].

Table 4.2: Demographics of the ninety-nine participants

Job	%
Application developer	65
One individual	19
Academic	13

Application architect	12
Manager/CIO/CTO	4
Other	6
Organization Size	
Fewer than 50 employees	32
50 to 500 employees	26
More than 500 employees	23
Sector	
Software manufacturing	48
Academic	19
Financial/retail	13
Communications/networking	7
Government	5
Other	8

To study whether and how Mylar affects programmer productivity, we needed to be able to compare activity during a subject's baseline period, in which they used Eclipse in their normal configuration, with their treatment period in which Mylar was also installed into their Eclipse. For instance, if a subject was mostly coding during the baseline period and mostly testing during the treatment period, the activity in the two periods would not be comparable. Since we were interested in comparing activity as a participant worked on multiple tasks, we also needed to ensure that both periods were long enough to encompass typical tasks.

Based on these goals, we defined criteria for a participant to be included in our analysis. The first was to ensure an appropriate amount of programming by setting thresholds on the amount of editing; only after a participant had reached a certain number of edit events was he moved to the treatment period. The second was to ensure that the effects of learning to use Mylar did not overly bias the usage data. To meet these criteria, our threshold of acceptance of a participant was triple the number of events (i.e., 3000 events)

needed to move from the baseline to treatment periods (i.e., 1000 events[57]). As the Active Search and Active Hierarchy features required the use of additional views, we did not allow subjects to install this feature until they had crossed the 1500 event threshold in the treatment period, in order to avoid creating too steep of an initial learning curve.

Subjects thus progressed through the study as follows. For the first 1000 events, participants were monitored only. After they crossed this initial threshold, they were given an option to download the tool. After they downloaded the tool and used it for another 1500 events, they were given the option to install the additional views. After that, if they accumulated 1500 or more events, their data was accepted for analysis. We refer to individuals accepted for analysis as subjects, and use the term participants for the entire set of individuals who registered. We standardized on the number of events rather than the time spent programming to account for variations in the rate at which different programmers work.

Of the ninety-nine initial participants, sixteen met the criteria to be considered subjects. This one-in-six ratio is indicative of the challenge we had in recruiting subjects: industry developers typically have little time to try out new tools unless they perceive an immediate and concrete benefit. The minimum two-week delay in getting the Mylar UI was one contribution to the drop-off, as was the need to use Mylar and continue participating in the study for several weeks to reach the 3000 edit event acceptance criteria. Participant feedback indicated that those who dropped out did not program as much during this period, did not use Bugzilla (the only task repository Mylar 0.3 supported), or stopped using the tool after they encountered a bug or incompatibility with another Eclipse plug-in they were using.

4.3.2. Method and Study Framework

We designed our field study to measure the effects of our tool within subjects. Participants joining our study were asked to install a subset of the tool, called the Mylar Monitor, which captured and stored their interaction history without affecting how they normally work with Eclipse. The monitor was extended with a module that would periodically prompt the participant to upload their interaction history as an XML file to a server at UBC, along with exception logs and feedback. To ensure anonymity, each participant was assigned a unique identifier. To ensure privacy, any part of the interaction history referring to the elements in the system on which the participant worked, such as Java type names, was obfuscated using a one-way hash function. We refer to this period of a participant's involvement in the study as their baseline period.

After the participant reached a certain threshold of work, which we chose to be 1000 edit events over no less than two weeks, the participant was prompted as to whether they wanted to install the Mylar task

[57] 1000 edit events corresponded to approximately 1-3 weeks of full-time programming based on trials of individuals in our lab.

context and task task-focused UI features. Installing these moved a participant into the treatment phase of the study. As before, the Mylar Monitor would continue to prompt the participant to upload their interaction history to a server at UBC. A participant was also notified when there were updates available for Mylar, including both feature additions and bug fixes. We ran the study for four months, July 6 to October 28, 2005 using Mylar 0.3. The task context model, scaling factors, and UI thresholds were frozen for the duration of the study (and for subsequent releases up to the latest Mylar 1.0).

To analyze participants' interaction histories, we created a reporting framework that allowed us to play back interaction to reproduce usage patterns and gather statistics. We used this framework to analyze the effect of Mylar on what programmers did and how they did it. We discuss the tuning of the scaling factors and thresholds for the study in Section 5.1.4.

4.3.3. Results: Analysis and Edit Ratio

Our focus for this field study was to measure the effect of task context on programmer productivity. We approximate productivity by comparing the amount of code editing that programmers do with the amount of browsing, navigating, and searching. To capture this behavior, we refined the edit ratio metric from the preliminary study to use an abstract notion of edits instead of keystrokes, but kept the same equation (i.e., *#edits* / *#selections*). The abstract edits corresponded to a sequence of characters before a pause in a text editor (e.g., a word written), and to manipulations in a graphical editor, such as changing a table value in the visual representation of Eclipse's plug-in editor, which results in an automated textual change of the underlying file. Edit ratio treats interaction consistently across different kinds of artifacts, whether the artifact is source code, another kind of file, or a binary library. Edit ratio also allows us to measure programming behavior independently of the task management features that Mylar provides since only interaction with program structure affects the ratio. This is important because edit ratio assumes that only edits of the system resources contribute to our productivity metric and not edits of tasks (e.g., comments on bug reports). For this study, we ignored interaction events that resulted from editing and selecting the tasks themselves since our goal was to validate how well task context represents the artifacts relevant to the task. The next study reports on interaction with tasks (Section 4.4).

Table 4.3 shows the edit ratios for each of the subject's baseline and treatment periods and highlights percentage change in the ratio. To determine whether there was statistical significance in the changes of edit ratios we normalized the edit ratios across subjects by taking the log of each and performed a paired t-test. The result is statistically significant with $p = 0.003$, indicating that the use of our Mylar tool improves edit ratio. Given that our choice of acceptance criteria for a participant to be considered a subject in the study was pre-defined and somewhat arbitrary, we also wanted to see if there was stability in this result for different acceptance criteria. We thus analyzed the edit ratios of programmers with a range of both lower and higher thresholds of baseline/treatment edit event cut-offs. Statistical significance

of the t-test ($p < 0.05$) holds until lowering the threshold results in inclusion of subjects whose usage data indicates that they did not use Mylar for their daily work (i.e., they are far below our acceptance criteria), and until raising the threshold is increased to the point where only six of the sixteen subjects remain.

Table 4.3: Field study data and percentage improvement

	edit ratio			filtered selections			activity	
id	baseline	treatment	delta	explorer	outline	problems	hours	tasks
3	2.9	7.8	172.0%	25%	7%	0%	91.3	61
8	10.1	26.4	161.4%	16%	0%	0%	71.3	30
6	14.1	36.0	155.8%	0%	0%	41%	64.7	23
7	2.6	5.4	111.3%	18%	5%	3%	44.4	54
12	2.7	5.4	102.3%	3%	0%	0%	24.4	5
15	1.7	3.3	91.7%	0%	0%	0%	25.3	3
16	8.8	13.0	47.7%	30%	14%	0%	35.1	7
10	5.8	8.2	42.1%	32%	22%	40%	11.3	6
2	11.3	14.8	30.8%	8%	1%	0%	27.5	11
9	6.7	8.7	30.7%	27%	0%	0%	43.4	12
13	6.8	7.4	9.7%	14%	3%	0%	48.5	4
5	4.1	4.3	5.4%	2%	3%	0%	6.5	12
11	2.2	2.2	3.5%	6%	0%	6%	12.4	7
1	7.7	6.9	-10.2%	25%	5%	0%	62.5	52
14	15.9	13.5	-14.7%	0%	0%	0%	66.2	9
4	11.0	8.1	-26.6%	0%	0%	0%	17.1	1

4.3.4. Results: Qualitative analysis

Our main hypothesis is that task context improves programmer productivity by making it easier for programmers to find the information that they need to complete a task, whether that is done by focusing a view in the user interface or by running only the tests relevant to changes made during the task. The edit

ratio analysis described in the previous section provides evidence that, for at least one measure, task context improves programmer productivity. In this section, we further analyze the content of the task contexts created by the programmers to determine whether the contexts were capturing the appropriate information. We consider the following questions: How accurately did the model capture the context of programmers' tasks? Did the programmers create and perform multiple tasks to which they returned? How much and in which views was filtering used?

Accuracy

Across the sixteen subjects, we observed three notable trends in the selection of elements: 84.2% of the selection events were of elements in the model with a positive DOI (i.e., the elements were visible in a filtered view); 5.3% of the selections were of elements that had only a propagated or predicted interest (i.e., not previously selected or edited, but visible in either a filtered view, Active Search, or Active Hierarchy); and 2.1% of the selections were of elements with a negative DOI (i.e., the elements had decayed out of visibility in a filtered view).

The first trend is indicative that programmers work on only a subset of the system artifacts and provides evidence to confirm that a task context does capture the majority of the elements often used as part of a task. The number of propagated and predicted element selections is slightly lower than expected due to our decision to not allow subjects to install the Active Search view until they had used Mylar for half of the treatment period's threshold, specifically 1500 events. We delayed the introduction of this view to avoid an overly steep initial learning curve. Once it was introduced, Active Search was not used as much as the other facilities and was used repeatedly by only five users. Qualitative feedback indicated several reasons for lack of use: performance problems, not having screen real estate available for another new view, and the view reporting too many matches[58].

The number of decayed selections indicates that the decay scaling factor may have been set too high. In contrast, usage data of the "Make Less Interesting" action indicates that at times too many elements were being shown, since two of the more active users frequently used this action (225 times for subject 3, 210 for subject 7, none for all others). This tension between data indicating that in some cases too much was shown, and in others too little, highlights the difficulty of providing one set of scaling factors for all tasks and all users (see Section 6.2.1).

---

[58] The degree-of-separation value needed to be decreased manually as the number of interesting elements within the default degree-of-separation (interesting files) grew large. The number of landmarks growing too large was not a problem because the landmark threshold was tuned so that no matter how long-running a task was there would not be a large number of landmarks.

Task Activity

We designed our study around measuring the effects of task contexts, and unfortunately did not include sufficiently rich monitoring of the meta-context to determine when the subjects recalled a specific previously worked-on task. However, we do know from the task activity how often subjects switched tasks (Table 4.3). Although Mylar is designed around facilitating work with multiple tasks, it can be used with one active, often long-running task (i.e., subject four, whose usage data indicates he or she worked with the same task active across eight Eclipse sessions). We are encouraged by the fact that most subjects used the task switching facilities regularly and those with the largest improvement in edit ratio used them heavily. The hours column in Table 4.3 is an indication of the total time a subject worked with some task active, approximated by issuing a timeout event when no interaction events had been observed for three minutes.

View Filtering Usage

Whenever a task was active in the treatment period, a task context was being formed and the UI of the IDE would show which elements were interesting through interest decoration. To inform and guide the effectiveness of UI mechanisms by which we project the interest model onto the IDE, we also analyzed usage trends related to the view filtering and predicted interest facilities. The percentages of selections made with the view in filtered mode are visible in Table 4.3 (for Package Explorer, Outline, and Problems views); the relatively frequent use of filtered selections is encouraging. Unfiltered selections result from either no task being active or the task being active but the view being in unfiltered mode. When a view is in unfiltered mode, many more selections are typically required to find the same information than when filtered; this causes the percentage of selections in filtered mode to appear lower than a subject might actually perceive.

4.3.5. Threats

One threat to the accuracy of the study results is that the subjects are not representative of typical industry programmers. The incentive to participate in the study was gaining access to a preview release of Mylar, and as such, this selection process was likely biased to early adopters of new programming technologies. Our study results must be viewed in terms of this potential weakness. Another threat is that we had no control over the tasks performed by subjects between baseline and treatment periods so their activity may have varied widely. This threat is addressed by the large amount of both baseline and treatment interaction we had for each subject. If programmers had worked on a single task across the baseline and treatment periods, changes in the edit ratio across the lifecycle of a single task could have been a problem. However, we have evidence that most subjects switched tasks multiple times during a workday (on average 2.3 tasks switches per active hour). Finally, bugs in interaction history creation, parsing, and

analysis could skew results. Our bootstrapping, testing, and ongoing use of the Mylar Monitor framework by ourselves and others is continuing to harden it against such errors.

An objective and generic measure of industry programmers' productivity is difficult as it depends on how a developer works (i.e., their process), what they work on (i.e., their domain) and how quality is measured in that domain. Although a definitive measure of productivity is elusive, edit ratio provides us with a measure of effort spent writing code versus effort spent looking for the information needed to write code. Since programmers chose to use the tool voluntarily, their choice to continue using it is also a positive indicator that the edit ratio metric approximates programmer productivity.

4.4 Study III: Knowledge Worker Field Study

To test whether the task context model supports more generic knowledge work, we adopted the same user-monitoring framework used for the programmer study. In particular, we wanted to investigate whether the model could accurately reflect the information needed to complete a task when the interaction was occurring with less structured information than in the programming domain. The goal of this study was not to determine whether task contexts make knowledge workers more productive, since the productivity of generic knowledge workers cannot be as easily approximated as that of programmers, whose productivity is typically defined by how much code they produce (Section 4.2.4). Instead, our goal was to gather usage data on how knowledge workers use task context. We also wanted to learn whether or not knowledge workers would explicitly mark their task boundaries voluntarily using the task management facilities, how accurately task context represented the file and web documents with which they work, and how necessary the support for decaying uninteresting items is for supporting file and web browsing activities.

We refer to the configuration of Mylar used for this part of the study as the Mylar Browser (Section 3.3). Rather than performing ad-hoc tuning of the task context model for knowledge work, we wanted to gather usage data to test how well this model worked for less well-structured data. For this reason, we left the scaling factors and all other implementation aspects of task context identical to those offered in the programmer study. The only substantial addition to the implementation of the tool is that it provided an additional structure bridge to support interaction with web documents. We also removed all of the programmer-specific features from the tool (e.g., Java development tools, Mylar Java Structure and UI bridges).

4.4.1. Subjects

The participants that we targeted for this study were knowledge workers within and related to our university that were not programming as part of their work activities. We advertised the study to approximately two dozen individuals who had heard of the programming tool. In all, eight individuals installed Mylar Browser, used it, and responded to our follow-ups for gathering usage data and feedback. The job descriptions, activities, and desktop applications other than the common ones listed below are summarized in Table 4.4.

Table 4.4: Accepted subjects in knowledge worker study

	Job Description	Activities	Desktop Apps	Web Apps[59]
S1	Technology transfer officer	Technology assessment, literature reviews, patent search, market analysis	patent search	patent search, marketing search
S2	M.Sc. student in Biology	Literature/database search, grant proposals	gene search	gene search, spreadsheet
S3	Secondary School Teacher	Lesson planning, Financial planning	finance	finance management
S4	Instructor	Teaching, managing administrative assistants	n/a	calendaring
S5	CEO	Manage operations of small company	n/a	contact management
S6	Project coordinator	Project management, hiring, reporting	web authoring	finance, student management
S7	Student services coordinator	Coordinating undergrad programs, student advising	n/a	student management
S8	Communications coordinator	Organize publicity and publish documents	graphic design, web authoring	n/a

Most of the individuals used the Firefox/Mozilla web browsers. Some used Mozilla and Thunderbird[60] for email, others used Microsoft Outlook[61]. All used Microsoft Office 2003 with the exception of S3 who

[59] All reported using Google as the primary search engine, http://google.com [verified 2006-12-21]

[60] http:// mozilla.com/thunderbird [verified 2006-10-02]

used the Microsoft Office 2007 beta. The commonly used Microsoft Office applications were, in order, Word, Excel, and PowerPoint.

4.4.2. Method and Study Framework

Participants were asked to use the tool as they saw fit for their daily work. Each was given a thirty-minute tutorial on how to use the tool by one of the researchers. Questions that arose about the usage of the tool during the study were answered by email and in person. Participants were provided bug fix updates of the tool. Since participants joined the study over a period of six weeks, the period over which we collected data for each participant varied. Subjects S2, S4, and S5 signed up in the last two weeks of the study. Multiple subjects had vacation days over the study period, or were out of the office, so we report days actively using the Mylar Browser in Table 4.5. Over the study period, we collected subjects' usage data. At the end of the study, we conducted exit interviews.

4.4.3. Results: Task Activity

The active workdays in Table 4.5 indicate the number of days in which subjects activated tasks. Subjects sometimes worked on tasks without activating them, for instance, if the task represented a work item that did not correspond to files or web pages. The broad spectrum in the number of active workdays is in part due to some subjects signing up at the tail end of the study period (S2, S4, and S5), and, in one case, due to a lack of use of the tool (S7). Although there is a broad spectrum in how many workdays subjects activated tasks, the table indicates that subjects will voluntarily indicate which tasks they are working on when using Mylar Browser. We believe this occurs because the automatic creation of task context supported by the Mylar Browser provides users with an incentive to activate tasks. The tasks/day column indicates how many unique tasks were activated that day. The activations/day column is higher in all but the S7 case, indicating that most subjects returned to tasks on which they had worked earlier that day. This indicates that subjects used task contexts for the purpose of restoring context when multitasking.

---

[61] http://microsoft.com/outlook [verified 2006-10-02]

Table 4.5: Task context activation

	Active Workdays	Total tasks activated	Tasks/day	Activations/day
S1	25	26	5.0	7.0
S2	5	5	2.8	9.0
S3	8	22	5.6	11.0
S4	4	9	2.75	3.8
S5	4	5	2.5	6.8
S6	26	41	1.8	2.2
S7	1	1	1	1
S8	14	29	3.4	3.8

Mylar's task management facilities do not impose any process for how tasks are defined. For this study, we also did not provide subjects with guidelines on how tasks should be defined. All subjects used personal tasks, not shared tasks (Section 3.2.3). Table 4.6 reports the kinds of work items that subjects managed with Mylar's tasks and the items for which they did not choose to use Mylar Browser.

Table 4.6: Work items reported by subjects to be associated with tasks

	Used Mylar for	Did not use for
S1	most work items	very short tasks, phone calls, meetings
S2	scholarship applications	literature/database search
S3	some work items, web research	working in quicken, phone calls
S4	most work items, personal tasks	quick email messages, new documents first without task then made them
S5	some work items, personal tasks	tasks defined in email
S6	most work items	task defined in email, meetings
S7	one work item	paper tasks, web application forms
S8	some work items	desktop publishing, web authoring

4.4.4. Results: Task Context and File Structure

To determine how the information visible in the task context corresponded to the underlying information that knowledge workers access, we devised several methods of measuring the contents of the subjects' task contexts. Our goal was to understand how effective our approach was at gathering and showing the relevant information compared to other common methods of organizing information, such as hierarchical filing or the use of consistent naming or tagging approaches to facilitate query-based retrieval. If workers were already organizing their information in a way to make relevant information to a task easy to find with existing mechanisms, we also wanted to know how much irrelevant information the Resources view, which showed files and web documents relevant to the task, was hiding with its interest filtering facility. This filtering facility was always on by default. The methods presented in this section consider only files. For web documents, since the amount of information available online dwarfs what a subject accesses in a typical task, the majority of available information is always hidden.

The degree to which Mylar Browser helps find and filter files depends on the size and structure of the file systems being used. Four subjects (S1, S4, S6 and S8) accessed shared document repositories that contained at least tens of thousands of files. Three subjects (S2, S3, and S5) accessed local folders that contained at least hundreds of files. Subject S7 did not use the file browsing feature. The mechanism by which we measured these metrics differed from all the other quantitative measurements we made because they required measuring the subjects' file systems and the Mylar Monitor preserves file system privacy. As a result, we performed the measurement on a limited sample due to subjects' time constraints. We did not take any measurements for subject S5 because that subject's file system was not available to us. For measurement, we chose the last two tasks that the subject had activated and that contained files in the context.

We defined four metrics to determine the relationship between the subset of the information in the task context and the information that would normally be shown through structure-centric views, such as through a tree view of directory structure or as the results of a query over all files tagged via naming conventions.

1) Average path length. This metric captures the average path length of all interesting files (i.e., files in the context with a positive DOI) to the root folder. The higher the number of nested folders that are used to file relevant information to the task, the higher the value of the hierarchy metric. The higher the value of this metric, the more folders the user would need to click and open in order to get to the file(s) of interest when resuming a task, unless they manually created and managed explicit links to folders, as subjects S1 and S5 had done prior to using Mylar Browser.

2) Average directory density. This metric captures the average ratio of interesting files in every directory within the context to all other files within that directory. The lower this number, the

more irrelevant information being hidden by the task context. For example, S6 frequently used shared directories that contained hundreds of Microsoft Office documents. However, S6 typically required only a few of those documents for her tasks.

3) Scattering[62]. This metric captures the average distance of an interesting file to a common parent directory. The higher the value of the scattering metric, the larger the number of places in the hierarchy that contain files of interest. For example, if the subject were working on a file a/b/c/foo.doc, and a second file in a/d/bar.doc, the scattering measurement would be (3+2)/2) = 2.5. A scattering of 0 means all files were in the same directory.

4) Tagging ratio. This metric captures a ratio of the number of interesting files that share a common tag in the name of the file to the number of all interesting files in a context. For example, "Case1" might be the tag used in two files, such as "Case1-Intro.doc" and "Case1-Data.xls".

Table 4.7: Task context content structure measurements

	Avg. path length	Avg. directory density	Scattering	Tagging ratio
S1	5.5	0.1	1.5	0.3
S2	2.7	0.2	1.1	0.9
S3	3	0.1	1.4	0.4
S4	2	0.3	0	0
S5	n/a	n/a	n/a	n/a
S6	1	0.01	1	0.4
S7	n/a	n/a	n/a	n/a
S8	n/a	n/a	n/a	n/a

As discussed above, the subjects with "n/a" values did not use the file browsing feature sufficiently to produce data for the tasks we sampled or we were unable to measure the contents of their file systems. For the other subjects, Table 4.7 indicates that the path metric was highest for S1, who worked on the largest network file system shared across dozens of employees, whereas the others had relatively short

[62] We use the term "scattering" from AOP, where it connotes the amount of structure not cleanly modularized within the dominant decomposition. In the case of a file system, the dominate decomposition is the directory structure.

paths to interesting files[63]. S1 used the richest hierarchical structure and tended to organize all files along with tasks. Since the subjects were able to map any file system folder into the Mylar Browser and since we performed the measurements within the Mylar Browser, in each case the path length is shorter than it would have been if accessed directly via the Windows file system. For example, S6 mapped only the few parts of the shared network drive that contained documents relevant to her into the Mylar Browser, using this as a shortcut to avoid needing to browse through additional path segments (e.g., omitting the first drive letter and two directories in "Z:\Documents\Planning\New Project" and mapping "New Project" directly into Mylar Browser).

All of the subjects' task contexts exhibited a low density over the file system structure. The low density measurement demonstrates a key benefit of task context; the vast majority of files within a directory were uninteresting and the task-focused UI showed only the few interesting ones. Scattering was present in every case other than S6, who mapped the parts of the file system containing her files, resulting in a flat structure with hundreds of files in each directory (with only a 0.01 density).

Tagging was done by S1, S2, S3, and S4. The ratio of tagged files was 40% or less, indicating that tagging was not a reliable way to identify all files relevant to a task.

4.4.5. Results: Task Context Contents

In addition to gathering data about the relation of the context to the structure, we also wanted to understand what was contained in the task contexts. Table 4.8 summarizes the task context contents. This data includes all files with which interaction occurred, even if the file had decayed out of the Resources view. The unit of measurement for the two size columns are elements; for example, on average S1's contexts referred to 69 web pages The total number of tasks for each subject is lower than that the total listed in Table 4.5 because subjects periodically created tasks that had neither a web or file context. For instance, some created a task for the purpose of an appointment and used the personal note feature of tasks (Section 3.2.3) to capture information relevant to the appointment.
This data indicates that both web and file contexts had a non-trivial size. Some task contexts accumulated many documents, as in the case of S1, who for long-running tasks accumulated 239 files in one context and 317 web pages in another. It also indicates that subjects used task context for working with files and for browsing, sometimes simultaneously. Six out of the eight subjects worked with task contexts that had both web and file elements, and for S1 and S5 most task contexts had both a web and file elements. This

[63] S1 had been creating shortcuts to each directory he worked with frequently on his desktop in order to avoid having to browse to the directories of interest each time. S1 had dozens of such shortcuts on his Windows desktop.

indicates that task context model's ability to uniformly represent interaction with various kinds of artifacts is useful, since many tasks involved both file and web resources.

Table 4.8: Task context content size measurements

	Web contexts			File contexts			Tasks with both
	Tasks	Avg. size	Max size	Tasks	Avg. size	Max size	
S1	7	69.1	239	16	42.7	317	6
S2	4	5.3	8	5	4.0	14	2
S3	9	17.8	96	2	3.0	4	1
S4	2	3.0	4	1	3.0	3	0
S5	5	32.2	125	4	68.8	225	4
S6	5	5.2	12	2	2.0	2	1
S7	1	5.0	5	0	0	0	0
S8	9	4.9	13	3	4.7	9	2

4.4.6. Results: Task Context Model Accuracy

To determine how accurately the task context model represented what users were interested in, we gathered data on the landmark, interesting, and decayed elements within the model, considering work done across all tasks (Table 4.9). This data indicates that in most cases more web pages and files were being filtered from the UI than were visible. This leads us to assume that the task context's ability to decay low DOI elements out of view is key to accurately representing the elements relevant to the task. Since some subjects had very long-running tasks (e.g., S1), our DOI ranking can change substantially over the course of the task. If not for the frequency and recency based DOI, the Resources view would likely have been so populated with elements that it would have not provided the task-focused UI benefits of being able to select an element of interest without scrolling or expanding a tree structure to find it. Coupled with the general feedback that the task context represented what subjects worked on (Section 4.4.7), we conclude that capturing an interaction-based weighting, and only showing the most relevant elements based on that weighting, provides an automated mechanism for focusing knowledge workers on their task and for facilitating multi-tasking.

Table 4.9: Proportions of task context model slices (direct interest only)

	Web (%)			Files (%)		
	Landmark	Interesting	Decayed	Landmark	Interesting	Decayed
S1	1	10	89	3	8	89
S2	0	73	27	5	50	45
S3	1	30	69	45	55	0
S4	0	100	0	0	75	25
S5	2	13	85	2	19	79
S6	4	96	0	0	1	0
S7	20	60	20	0	0	0
S8	3	52	48	3	34	63

4.4.7. Results: Feedback

With both a large amount of interaction recorded and a high occurrence of decay, the question that remained was how accurately the visible elements represented what was relevant to the subjects' tasks. To judge this aspect, we asked each of the subjects about the accuracy of the task context model that was presented in the Resources view. Table 4.10 summarizes their assessments.

Table 4.10: Feedback on how much information relevant to the task was displayed

	Too much	Too little	About right	Comments
S1		X		Some web documents disappear from context too quickly after reactivating context
S2			X	Page redirects sometimes pollute the web context
S3	X			Initial navigation puts too much uninteresting information in the context
S4			X	Sometimes too much was shown initially in web context
S5	X	X		Sometimes too much shown initially, not always a reflection of how time is spent

S6			X	Since it is not the default browser some relevant web context missing
S7			X	Did not use enough
S8	X			Initially web context tracks too many wrong site visits

All of the subjects were very positive about the idea of context being tracked automatically as they worked, expressing that the task context represented what they worked on and that they liked seeing only the files and web pages relevant to their task. However, when asked how accurately the task context model portrayed the files and web pages in which they were interested, almost all recalled cases where either too many or too few elements were filtered away. Common feedback indicated that web contexts often get overpopulated at the beginning of a task when nothing has decayed but many links have been clicked in search of something, especially when performing a web search. Some reported that, in many cases, only the last pages browsed were highly relevant. Also, some subjects reported that interaction with web documents could cause files to prematurely disappear from the context. Finally, we asked subjects whether they liked using the tool and asked if they would continue using Mylar Browser. We are encouraged that all stated they would continue using it (Table 4.11).

Table 4.11: General feedback

	Like	Dislike	Did not use enough	Will continue using
S1	X			X
S2			X	X
S3	X			X
S4	X			X
S5	X			X
S6	X			X
S7			X	X
S8	X			X

We conclude that task context can be integrated into knowledge work applications. If provided with the benefit of the task context, knowledge workers will voluntarily indicate on which task they are working. Although the tuning of the scaling factors and thresholds could have been more precise, feedback from

subjects indicated that integration with other applications (e.g., email, better Windows integration) was far more important to the subjects than task context model tuning improvements.

Based on the low task activation usage and task context model contents of the S7 subject, whose occupation involved the highest proportion of data entry, we hypothesize that the task context model may be more useful for knowledge workers needing to access a large amount of information than for those spending most of their workday in a single data entry application.

In terms of task context model implementation, the key lesson we learned is that differences between structure bridge implementation and monitoring can have a major impact on how task contexts evolve when they contain aggregations of different kinds of resources. For example, when a context had a lot of web interaction, interesting files decay out of the context too quickly. This situation happened because the Mylar Browser monitored more interaction with web pages (every click) than with Microsoft Office documents. Extending monitoring facilities to include scrolling or time-based interaction events for semi-structured resources could improve the accuracy. However, we believe that the existing scenario, where different structure bridges provide different levels of structural awareness, will be common. We discuss approaches of addressing this issue, such as normalizing or scaling the relative amount of interaction, in Section 5.1.3.

4.4.8. Threats

The number of subjects in our study was small, and these subjects may not be representative of any interesting class of knowledge workers. Given that the subjects fulfill a variety of roles, the results may generalize to an interesting population, although further study with a better sampling across a wider population is warranted. The generalizability of our results is also limited because the amount of data collected for four of the eight the subjects corresponded to one workweek or less. Another threat to our results is that the subjects may have only used the Mylar Browser for tasks well suited to the tool. We do not know what percentage of their daily tasks for which they used the tool. The heavy use of the tool by some participants suggests that it does fit some work patterns; for example, one participant (S1) even stopped using MS Outlook task management in favor of the task-focused desktop. Future studies should consider which tasks our task context model supports and which it does not and why.

4.5 Summary

Our user studies have validated that an explicit task context makes programmers more productive. We have shown with statistical significance that productivity, as defined by the amount of time spent navigating versus editing, increases for programmers when they work with explicit task contexts. We also demonstrated that task context generalizes to file and web browsing activities. As a result, we believe this is a promising technology for reducing information overload in the knowledge work domain.

5. Discussion

In this chapter, we discuss potential improvements to the task context model to better support both programming and other kinds of knowledge work. Many of these potential improvements are based on experiences from the field studies and on the feedback from the user community of Mylar IDE[64]. In this chapter, we review ways to improve the accuracy of the task context model (Section 5.1) and the task-focused UI (Section 5.2). We then describe and discuss support in Mylar IDE for task context collaboration (Section 5.3). We also present methods of visualizing task context (Section 5.4) and conclude this chapter with a discussion of future work (Section 5.5).

5.1 Improving Task Support and Context Accuracy

5.1.1. Working on Multiple Tasks Concurrently

Our preliminary study data indicated that programmers needed support for working on multiple tasks concurrently. We interpreted this input as programmers needing to have multiple tasks active and implemented support for multiple active tasks in Mylar 0.2-0.3. This implementation distributed interaction events among all active tasks. The fact that no Mylar IDE users complained when we eventually removed this feature has indicated that our interpretation was wrong. Although our existing user base needs support for working with many tasks concurrently, for the common cases, they do not need support for working with tasks in parallel. Use of this feature also makes the interaction with task activation considerably more complex. As a result, we removed the capability of activating multiple tasks in Mylar 0.4, and instead we have focused on making switching between tasks easier.

5.1.2. Related Tasks

Our model treats a task as an independent atomic unit. In practice, tasks are often related. Consider a programmer working on fixing a bug. The programmer creates and activates a task for the bug. As work progresses on the bug, the programmer identifies and begins work on a related bug before the first bug can be resolved. With our current model, the programmer has two choices: deactivate the first task and

---

[64] In the 11 months since the conclusion of the programmer field study (Section 4), Mylar IDE users have filed 260 enhancement requests and 928 integration and bug reports: http:// eclipse.org/mylar/bugs.php [verified 2006-10-12]

recreate the context when starting work on the bug, or do both tasks under the context of the first. Both choices are problematic, and while the latter is easier, it causes a potential loss of context as the programmer cannot return to the context for just the second bug. We believe that addressing this problem will require extending the model to support schemas for tasks (e.g., subtasks, sequences) and allowing the programmer to have one or more related task contexts active by means of context composition. For example, if a user activates a task with three subtasks, each of the contexts of the subtasks could be activated simultaneously, so that the parent task includes all of the contexts of the subtasks. This technique would also support pre-populating the context of a new task, for example, with the composite context of a parent task. Pre-population is a commonly requested feature. This feature requires careful UI design to avoid the complexities that arose in our previous attempts to support multiple simultaneous active tasks.

5.1.3. Task Context Lifecycle

Our task context model is oblivious to the lifecycle of a task. We use the same scaling factors and apply the same algorithms for operations whether a task is near its start and has a sparse context, or near its completion and has a rich context. Making the model sensitive to a task's lifecycle could further improve accuracy. For example, at the beginning of a task, it may be beneficial to have a slower rate of decay, and suggestions for related structure could come from a broader degree-of-separation when the task context is small. Near the end of a task, the core set of information in the context has stabilized and the context contains more information. The size of the task context could be used to adapt the DOI function, scaling factors and degrees of separation, helping tailor the contents of the model to the task's lifecycle.

5.1.4. Tuning the Scaling Factors

A concern we had prior to starting the field study was that poorly tuned scaling factors could prevent the task context model from capturing the information programmers needed and that scaling factors might need to be personalized for different tasks types, programmers, and display resolutions. We decided not to expose a mechanism for a programmer to change the scaling factors because we believed that the problem of information overload was so severe for large system development that an approximate tuning would suffice. We chose an order of magnitude value for each scaling factor and used it in our daily programming with Mylar. This resulted in only slight variations within those orders of magnitude being set for the Mylar 0.3 field study release (Section3.1.3).

We did not focus our validation on the settings or tunings of these parameters because they only play a minor role in the overall effect of adding task context into an application. For example, if we turned up the decay several times higher, task context would still help, but would show fewer of the relevant

elements. If we turned it down several times lower, the user would have to scroll more but would still see the information relevant to the task.

Although we expected to change the scaling factors and thresholds based on feedback from the study participants, the values continue to work and remain unchanged through to the 1.0 release. As a result, we believe that a substantial improvement may require a more sophisticated tuning approach that adapts to properties such as the task's lifecycle, the type of task and programming domains, and the user's profile. Further study is necessary to determine how varying and adapting scaling factors affects the accuracy and precision of the task context model.

5.2 Improving the Task-Focused UI

5.2.1. Decorating Elements to Indicate Interest Level

A notable change we made between Mylar 0.1 and 0.3 was to remove the forced highlighting scheme (Figure 4.2) as a default. In Mylar 0.7, the highlighting was removed entirely from the Mylar distribution. We had initially thought that using a background color to indicate how interesting an element was would help distinguish elements that were most interesting, or that were about to decay out of the view. We supported both a discrete and a continuous highlighting scheme and allowed the assignment of highlighters per-task. The user study data we collected did not show much use of the highlighting scheme. Feedback from users also indicated that they found the highlighters to be visually distracting. Since neither we, nor the user community, expressed a need to be able to distinguish beyond the three levels of interest (e.g., uninteresting, interesting, landmark), we removed this facility.

One of the reasons we thought that the foreground highlighting would be necessary was to allow multiple active tasks (Section 5.1.1) to be distinguished visually. Although that support is gone from the programming parts of the UI, the highlighting idea may be useful for supporting other activities that involve multiple contexts being active, such as code reviews (e.g., inspect all contexts for a release) and context comparison (e.g., compare remote context with local workspace's context).

5.2.2. Exposing the Interest Thresholds

An alternate approach to creating smarter threshold levels and scaling factors is to expose these to the user for manual tuning. We prototyped this support in the UI and occasionally still use it for debugging purposes. We have chosen not to expose it to avoid complicating the UI with manual tuning facilities. We believe that this approach could be useful in some situations, for instance when viewing large aggregations of contexts (e.g., for code review).

An example UI to allow direct manipulation of thresholds is shown in Figure 5.1. This context slider enables the programmer to directly manipulate the thresholds for encoded interest visibility and predicted interest visibility. The former can control the interest filtering threshold for elements with a direct interest (e.g., allowing more task context to be shown on larger displays). The latter can be used to define how much predicted context is made visible to the user (e.g., more can be shown when exploring, less when working on a well-defined task). Sliding the left slider to the left will show more of the past interaction history, while sliding the right slider to the right will show more of the potential future interaction history. Sliding them both to the centre will result in only the current selections being visible.

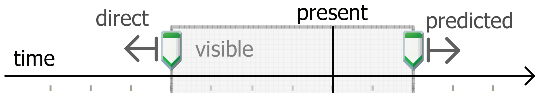

Figure 5.1: Mockup of threshold control UI

5.3 Collaborating with Task Context

Two or more programmers often end up working on the same task. This kind of collaboration can occur in at least three different ways: programmers may huddle around the same computer, they may pass the task back and forth with one programmer making progress and then delegating it to another, or they may work on the task at different times and in different places, such as when one programmer revisits a task completed previously by a colleague.

Mylar helps programmers in each of these scenarios. When multiple programmers work on a task simultaneously at one computer, the focus provided by the task context can make it easier for the programmers to discuss the software and for the non-driving programmers to follow the actions of the driving programmer on the screen.

Mylar provides assistance for the other two scenarios by facilitating the sharing of task context. We have experimented with two means of sharing task context. Most commonly, we attach the context used in solving a bug to the task describing the bug in a shared task repository. Second, a context for a task can be passed through email. We use the former approach in the Mylar project, since we promote transparency in the development process by making our task contexts openly available.

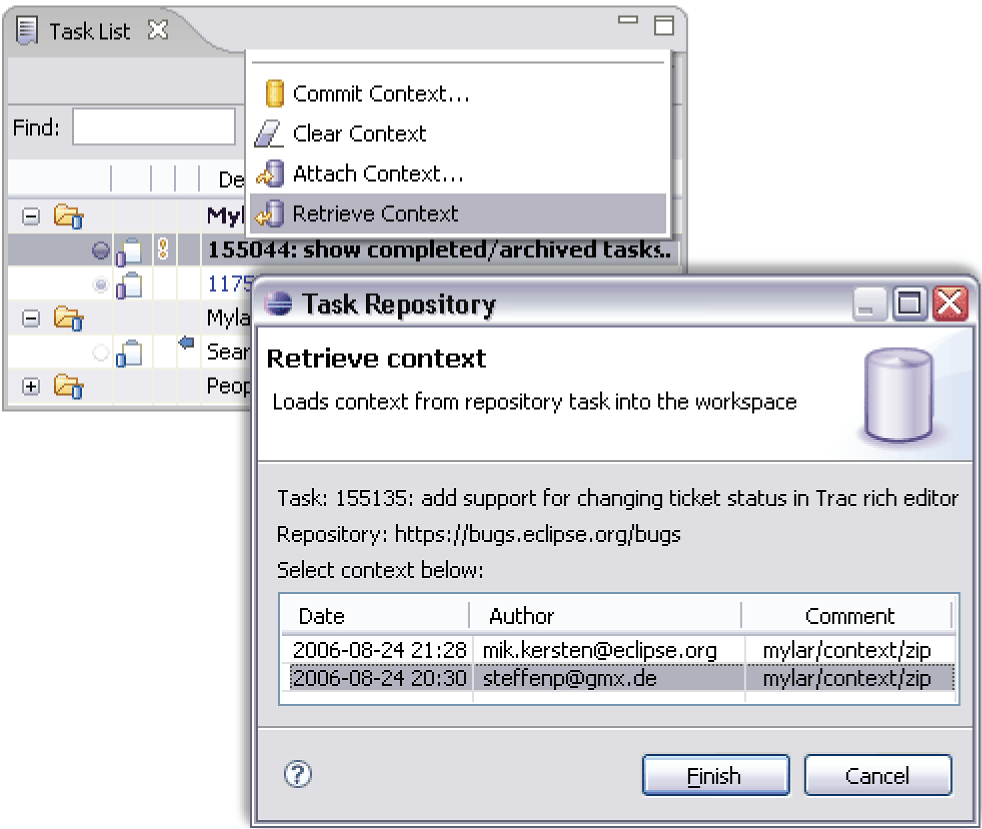

Figure 5.2: Context sharing initiated from the Task List

A programmer who wishes to work on a task unfinished by another programmer or who wishes to see how a bug was resolved, can import the context for that task into his or her workspace (Figure 5.2). Just as one programmer can switch between tasks, a programmer importing a task context can switch to that context thereby accessing just that subset of the software system that the original programmer had considered in completing the task. When contexts for tasks are stored in a shared repository, such as the Bugzilla repository used for the Mylar development itself, the repository becomes a richer source of knowledge about how to complete problems. Currently, the Mylar source repository has contexts attached for 253 tasks (as of September 14, 2006). These shared contexts have made it much easier for Mylar's developers to work on reopened bug reports and to delegate partially completed tasks. The Mylar open source project also has a policy of requiring task contexts to be submitted as patches. This policy has made it easy to apply dozens of patches contributed each month.

Mylar also helps focus communication between multiple programmers by providing an Eclipse-based user interface for working with repository tasks via the Mylar Task List. In particular, Mylar supports the use of queries to watch for changes in particular categories of tasks (Section 3.2.3). For example, a query may be added to watch all updates made to a task by a particular colleague. When the colleague adds a comment to a task, the programmer will see the task appear under the query in the Mylar Task List and

will see an incoming arrow to represent the changes that have been made. When the programmer opens the task, the editor for the task highlights the incoming changes and unfolds all new comments.

Another way to add more collaborative support for working with task context is to provide recommendations. Mylar provides an experimental form of recommendation called Active Search, which for resources of landmark interest, automatically runs and displays reference-based searches. For example, the Active Search may display the callers of a particular method that have a high-degree of interest. By knowing the context of a task, when a programmer begins work on a new bug report, it is possible to similarly recommend previous bugs completed in the past and to provide rich support in suggesting which parts of the system may be relevant to solving the problem [51]. These recommendations provide focus when they are sufficiently accurate; inaccurate recommendations would reduce the focus of the programmer.

At times, it would also be useful for programmers to know which parts of the system are being worked on in parallel by other team members. One way of presenting this information is through decorations to resources in the IDE [10]. This decoration can be overwhelming when applied to all resources in the system. By knowing the current task contexts of other programmers, it may be possible to scope the collaboration information presented to provide more focus. For example, Figure 5.3 is a mockup UI that could be used to show a programmer that the bug she is fixing is also active in a team members' task context (via the bug icon on the left of Figure 5.3). Clicking this icon would then bring up a dialog with the bug details that could show the overlap between the two programmers' task contexts. The presence of overlap could serve as a warning to prevent a conflict that could result from their simultaneous editing of the same code, or as a prompt to share expertise.

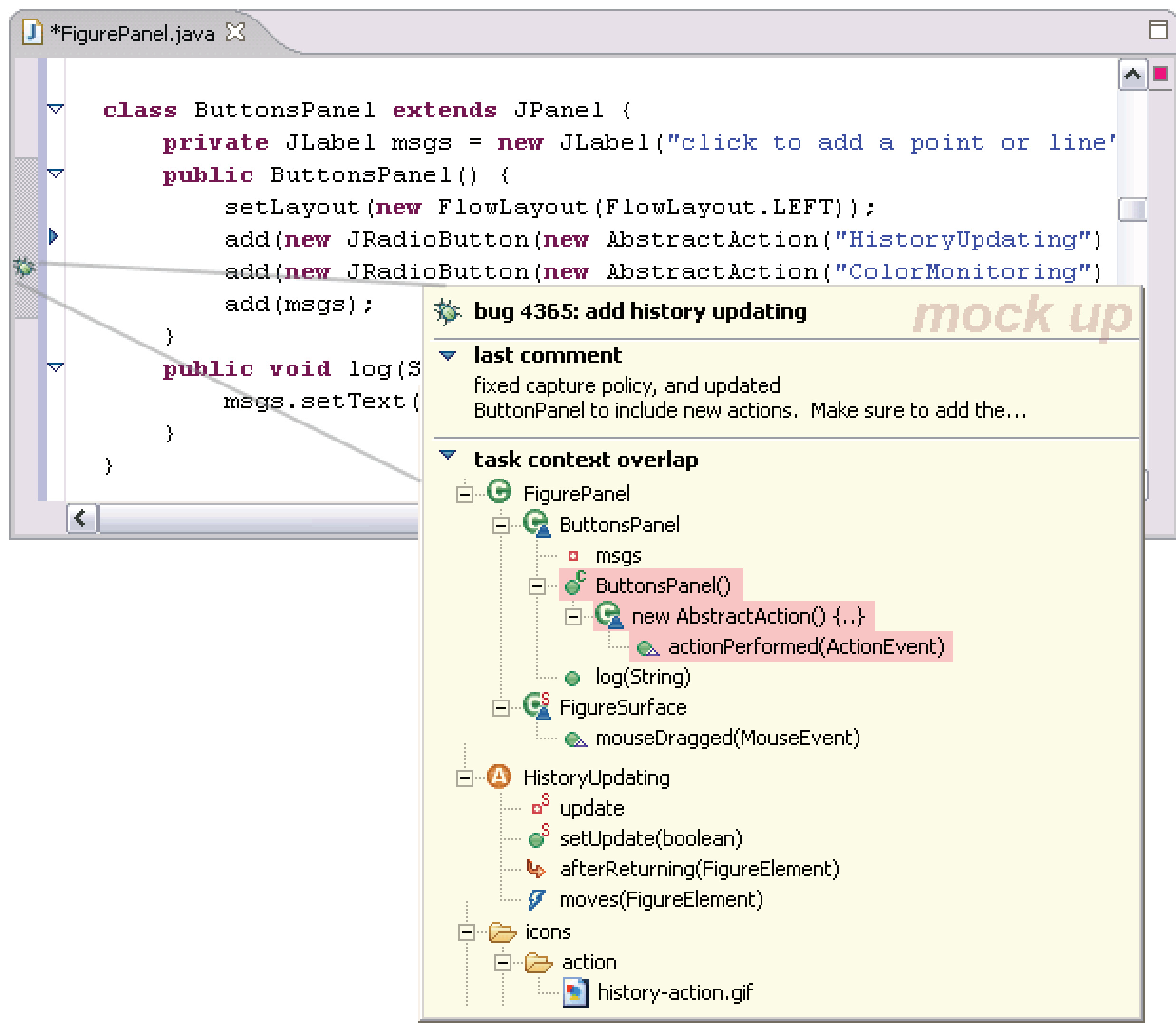

Figure 5.3: Mockup of overlay showing overlap between two team members' contexts

5.4 Visualizing Task Context

Our implementation and validation focused on displaying task context within the existing display facilities offered by modern IDEs. For the field studies, we targeted a 1280x1024 screen resolution. Even at this resolution, we received feedback that programmers did not have enough screen estate to show a view such as the Active Hierarchy even if they found it useful[65]. In the knowledge worker study, the situation was worse because some subjects had 1024x768 screen resolutions and were accustomed to a full-screen browser.

---

[65] The Mylar 0.7 documentation recommends not using Active Search or Active Hierarchy with screen resolutions lower than 1920x1200, a typical maximum resolution available on today's developer workstations.

However, larger screen resolutions are becoming more common. We have experimented with several graphical views of task context. We have found these visualizations of task context useful for our own internal experimentation.

Our first attempt at depicting interaction history was called Pathfinder, which used a grid layout where interesting files were arranged horizontally according to the package in which they were contained. Rather than sorting files in descending interest, this view placed the most interesting files in the middle, and sorted less interesting files in descending order above and below those. The intent was to allow a ridge of the most interesting paths through the package navigation to form, as visible in Figure 5.4. In addition to the background highlighting, the border of the file node was decorated to indicate with decreasing luminosity the place that this file had in the interaction history in order to make it easier to know where the "Back to previous file" navigation would lead. The problem with this view was that when many files within a single package were selected it required either too much vertical space or forced scrolling. We discarded this visualization and instead tried to show the same structure using the existing Package Explorer view. From this start, we came up with the idea of filtering existing tree views to focus on the task context.

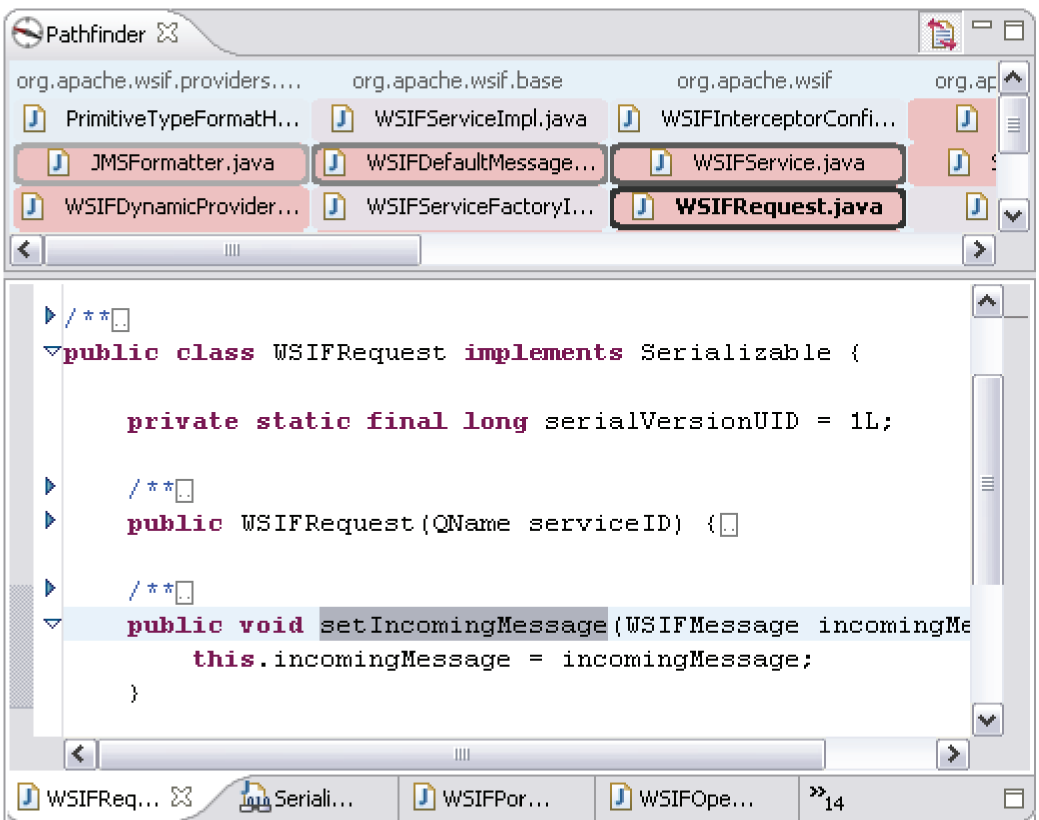

Figure 5.4: Table layout showing interesting files

Unlike the Pathfinder view, the visualizations we describe below are actively in use for the purpose of debugging or inspecting the task context model and for the purpose of continued experimentation with

task context visualization techniques. Those that refer to the "Sandbox" in their view title indicate that they are in the developer part of the Mylar open source repository[66].

5.4.1. Spring Graph of Task Context

For the purpose of debugging and inspecting the task context model, we use a view called the Context Tree (Figure 5.5). This view shows all of the elements and relations in the task context. It also shows, through annotations, the DOI level of the elements and relations. This view is similar to Active Search, but takes the slice of all elements and relations in the model, regardless of DOI. The problem with this view is that it is difficult to get an overall sense of the relations in the model. For example, Figure 5.5 shows a cycle in the task context graph that causes the tree view to expand indefinitely.

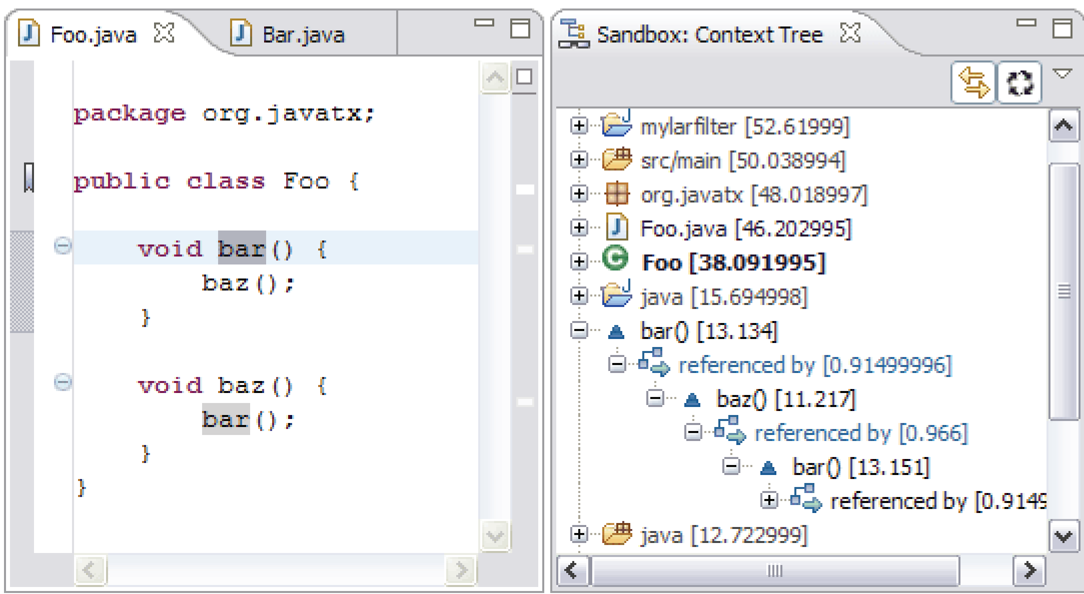

Figure 5.5: Task context tree view showing relations of elements

To address this issue, we created a force-directed or spring [25] layout of task context that uses the DOI of each relation as a weighting for the layout algorithm, as show in Figure 5.6. A higher the value of the weight corresponds to a greater the tension on the spring, resulting in the closer the proximity of the corresponding elements. This view explicitly displays the task context model's interest groupings and propagations that we discussed in Section 2.2. A key property of this visualization of task context is that it exhibits high stability as interaction is processed. Thanks to the frequency and recency weighting, task context tends to contain a small number of landmark elements with high interest, as well as high interest relations connecting those elements. This results in a high spring tension between those elements. Only small local disruptions occur to the layout when a new element is added.

---

[66] http://eclipse.org/mylar [verified 2006-10-02]

Our spring layout can also be adapted to other visualizations that show elements and relations, such as Unified Modeling Language (UML) static structure diagrams. Although this layout requires considerably more screen estate than the tree views we currently rely on, we believe that it may be an effective task context display mechanism once larger screen resolutions become more common.

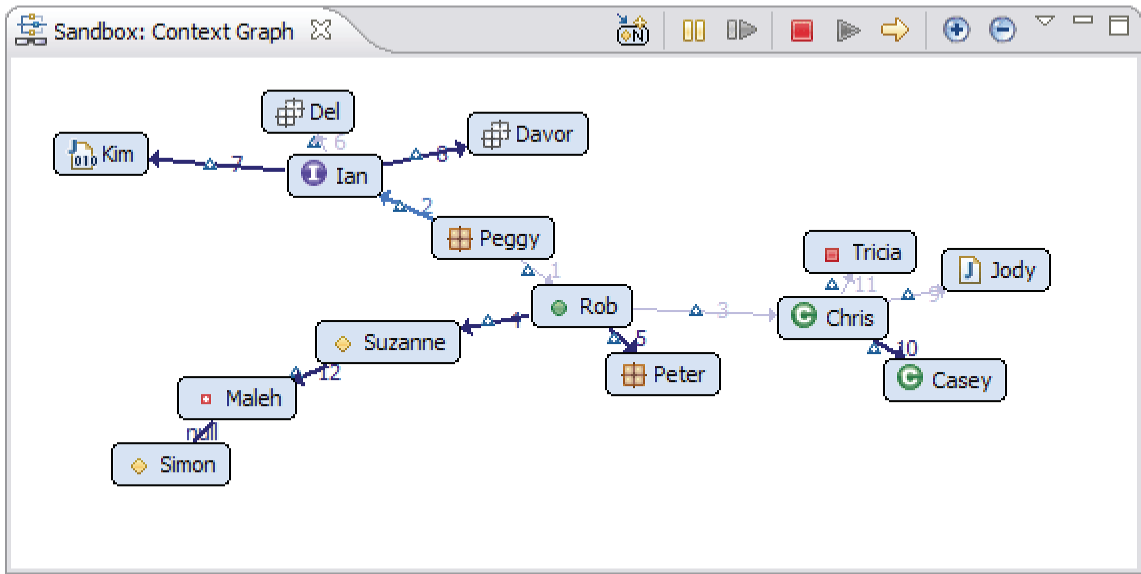

Figure 5.6: Force-directed layout of task context

5.4.2. Seesoft Visualization of Task Context

The Seesoft layout provides a view of system structure that can be used to highlight changes [22]. We wanted to test how well task context supported this layout. Figure 5.7 shows a Seesoft visualization of task context[67] where the vertical bars correspond to the slice of interesting files and the highlighted regions correspond to the slice of interesting elements within those files (e.g., Java methods). Our own experience using this view indicates that its information density is likely to be too low for most programming activities. However, we do make it available for experimental use in the Sandbox. We believe that such a view could be useful for other activities such as reviewing code to find "hot spots" that could, for example, be candidates for refactoring.

[67] For this we use Visualiser component provided by the http://eclipse.org/ajdt project [verified 2006-10-06]

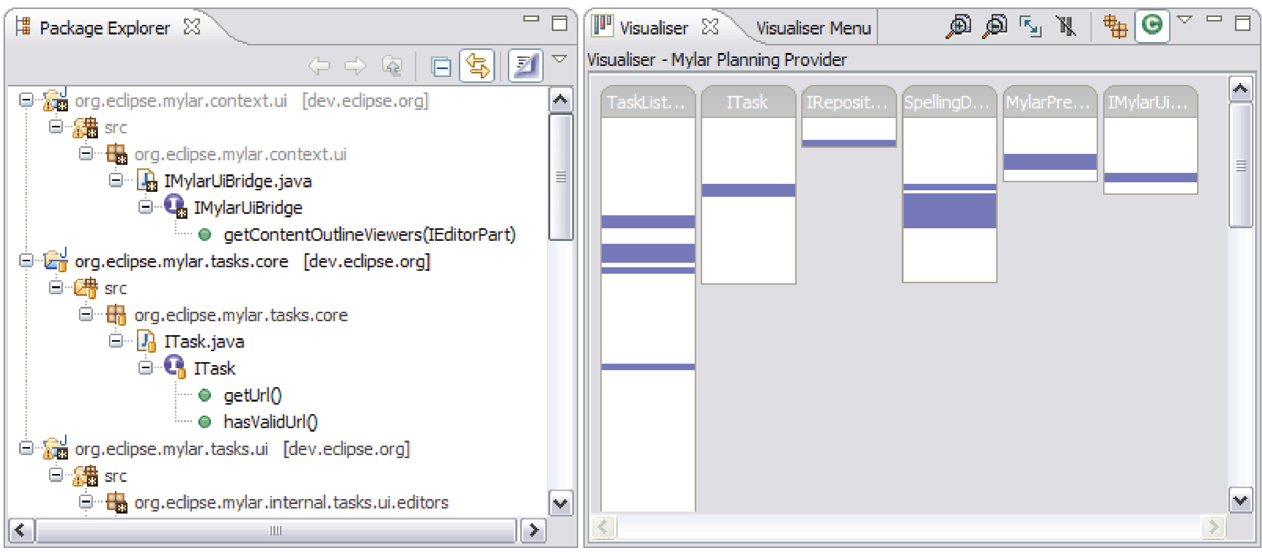

Figure 5.7: Seesoft visualization of task context

5.4.3. Interaction History Chart

In addition to prototyping views to improve how task context is displayed to the user, we also created visualizations to improve our understanding of the massive amounts of usage data that came from our field studies. One example of such a visualization is shown in Figure 5.8, where the task context algorithm was run on a user's data to simulate what would be visible in their Eclipse if he had been working with a single task active for the entire duration. This visualization ignores decay and depicts a few elements that gradually gain landmark interest, and the large number elements with low interest. We have used these visualizations for to look for trends such as frequency of landmarks and the range of their interest, and believe that such visualizations have potential for future work in identifying patterns in interaction histories.

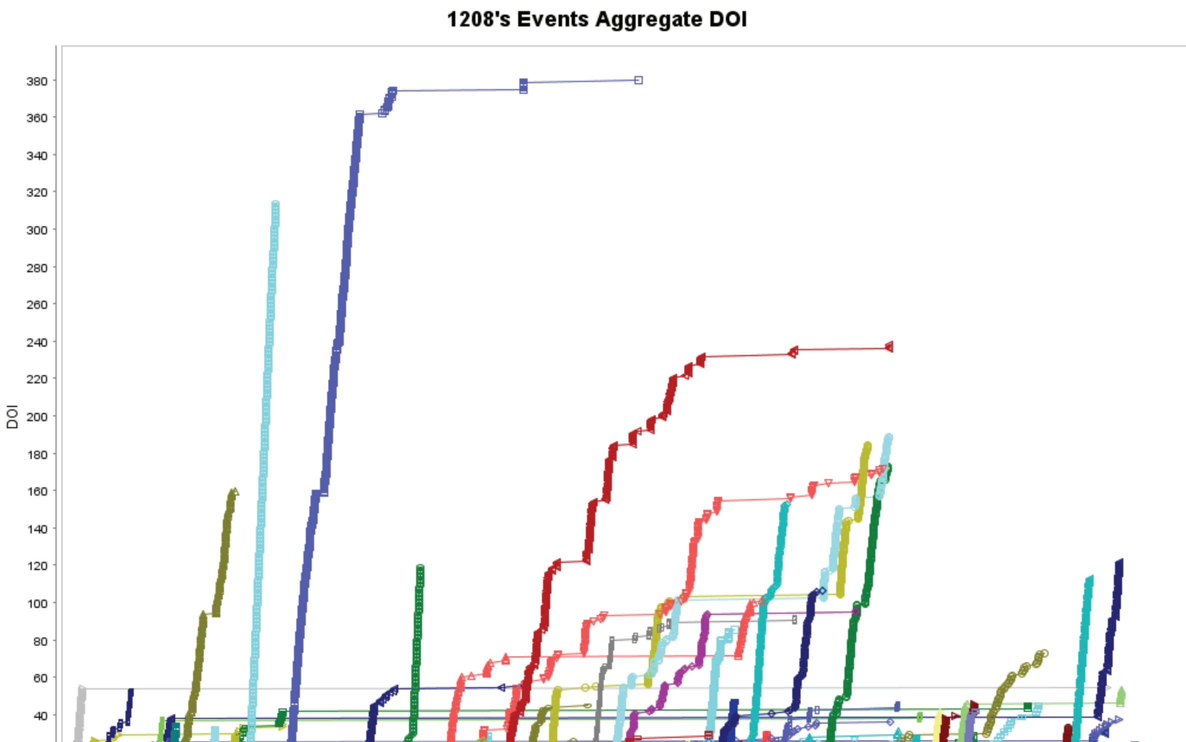

Figure 5.8: Visualization of interaction history with decay ignored

5.5 Future Work

We have defined a model of task context and validated an instance of the model for a development environment and for a file and web browsing application. Our implementation is one particular point in the design space of using interaction to scope down the information presented in an application. In this section, we discuss alternative approaches and future work that could improve task context accuracy and applicability.

5.5.1. Transforming and displaying interaction history

Alternate implementations of task context could use different schemas for representing interaction, different algorithms for transforming interaction history into a task context and different data structures and operations for modeling task context. As one example, a simplified model could ignore interaction with generic relations as elements in the model and make the common containment relation (i.e., hierarchy) a first-class part of the model. Another could make relations more prominent, if the domain made relations a direct target of interaction (e.g., an interactive visualization tool that allows direct selections of the edges between elements).

We have defined a set of task-focused UI mechanisms, suitable for use with Eclipse and similar UI frameworks. Different task-focused UI mechanisms could be used to display task context. For example, in addition to sorting or using force-directed layout to focus a view, groupings or visualizations that cluster elements with related interest values could show additional properties of the task context model. Navigation patterns present in the interaction history could also be overlaid on task-focused UI displays in order to further highlight elements interacted with recently (e.g., building on the simplistic shading in the Pathfinder visualization described in Section 5.4).

5.5.2. Extending Support to Other Domains

Our implementation for task context targeted an IDE and a file and web browsing tool. Task context could also be applied to other application domains. Extending task context to domains that have a well-defined notion of task and structure is straightforward, but more work is needed to determine how the task context model could be extended to support domains where tasks and structure are less well-defined. In a domain where tasks are less structured, entire activities, such as listening to music, could correspond to a task context. Supporting such domains would also involve exploring integration with artifacts that are typically considered unstructured, such as images, audio, or video files. What would be required to integrate with such artifacts is a notion of identity of the elements in those artifacts. For example, the identity of the elements in an audio file could be time ranges. Alternatively, the identity of elements in a video file could come from the metadata for that video. Once such a notion of identity is defined, the interaction monitoring and structure bridge frameworks could be used to focus interaction with these kinds of resources.

5.5.3. Improving Interaction with Filtered Views

Knowledge workers tend to use a variety of displays ranging from mobile devices to high pixel count desktop or wall-mounted displays. All of the task-focused UI mechanisms that we have defined have used fixed threshold values. These thresholds determine which elements are landmarks, and when elements decay out of a view. Consider a programmer working on a task on his 1024x768 laptop. Given our current tunings, it is likely that this programmer will see a scrollbar in their Package Explorer and will need to do some scrolling to find landmarks of interest. Conversely, a programmer working on a 2048x1536 display is likely to see a considerable amount of unused space in their Package Explorer. The thresholds for tree views could be adapted to the available vertical space in the views. As the context grows, the threshold for an element to be interesting could be gradually raised from $-\infty$ in order to fill the visible area of the view without adding a scrollbar. Coupled with the automatic view expansion (Section 3.2.2) this technique could be used to ensure the guaranteed visibility [50] of landmark elements. The tradeoff with

this technique is that it would set the threshold per-view, which would come at the cost of different views showing different slices of the context.

5.5.4. Unified Interaction, Context, and State Model

Task contexts provide a full history of the interaction with a task. When considered along with a representation of the changing state of information, such as a version-aware document repository, task contexts could enable "rewinding" both interaction and state concurrently, enabling a worker to easily return to a previous state in their work history. Our usage metrics indicate that programmers generate roughly 1-10 Megabytes of interaction information per month. This means that it could be feasible to maintain a unified model of all of the interaction all users have and link this with the state changes, without adding substantial overload to document repositories, which are often measured in terms of Gigabytes. Such a unified interaction and document storage facility could support rewinding the state of a system across one or more tasks. Our current model has no support for undoing interaction, and supporting this might require the ability to fork interaction histories.

A related challenge is handling resources that change beyond of the reach of the interaction monitoring framework. For example, currently it is possible for the task context model to be updated by all the changes in a software system as long as the source code repository tracks those changes (e.g., by storing the refactoring information that details how elements are renamed). For this to be feasible for web pages, the task context model would need to be notified of changes to the structure of the internet, or of a particular intranet of interest.

5.5.5. Cascading Contexts

We have ignored the schemas that knowledge workers use to define their tasks, such as grouping tasks by project or structuring them into subtasks, sequences, or dependencies. Since the task repositories with which Mylar integrates support such task structuring, Mylar IDE users have made frequent requests to allow tasks contexts to contain other tasks and their contexts. For example, if task A has subtasks B and C, the context of task A could be the composite context of task B and C. Mylar can be extended to support this kind of flexible cascading of task contexts using the composition operation. Support for cascading contexts could have a profound effect on the representation of the task context model. For example, instead of storing individual contexts and requiring composition of those contexts, a task context could be identified by a set of spans of an interaction history and relations to other such spans. Activating a task context could then involve "swapping in" the corresponding segments of interaction. A simplistic version of this is approximated by our current implementation, which always maintains a composite context of the activated tasks in main memory. However, due to disk space and memory constraints we discard repeated

events with the same element corresponding to the same task context's interaction history (Section 3.2.4). More availability of main memory and disk space could remove this restriction allowing the model to have access to all previous interaction, enabling facilities such as interaction pattern detection and the ability to rewinding interaction across one or more tasks.

5.5.6. Focusing the Workweek

Once task management becomes an integral part of the work process, it is possible for the Mylar Task List to become inundated with tasks. For example, the author's task list currently contains 2394 tasks, 677 of which are incomplete. In realizing this, we initially became concerned with the profound irony of having moved the information overload problem from the structure views to the task management view. However, we then realized that we could apply task context to tasks (Section 2.4). We use the same operations and similar task-focused UI mechanisms to filter tasks that are not relevant to a slice of the context for the current workweek (Figure 5.9). Tasks that are not scheduled for this week are uninteresting and filtered. The slice of tasks scheduled for the current workday is highlighted in blue, and the slice of those overdue is highlighted in red. Tasks of a high interest, such as those with incoming comments, are always visible (blue arrows in Figure 5.9). The Mylar bug database indicates that a substantial number of users work in this "Focus on workweek" mode in order to manage the large number of tasks in their Mylar Task list.

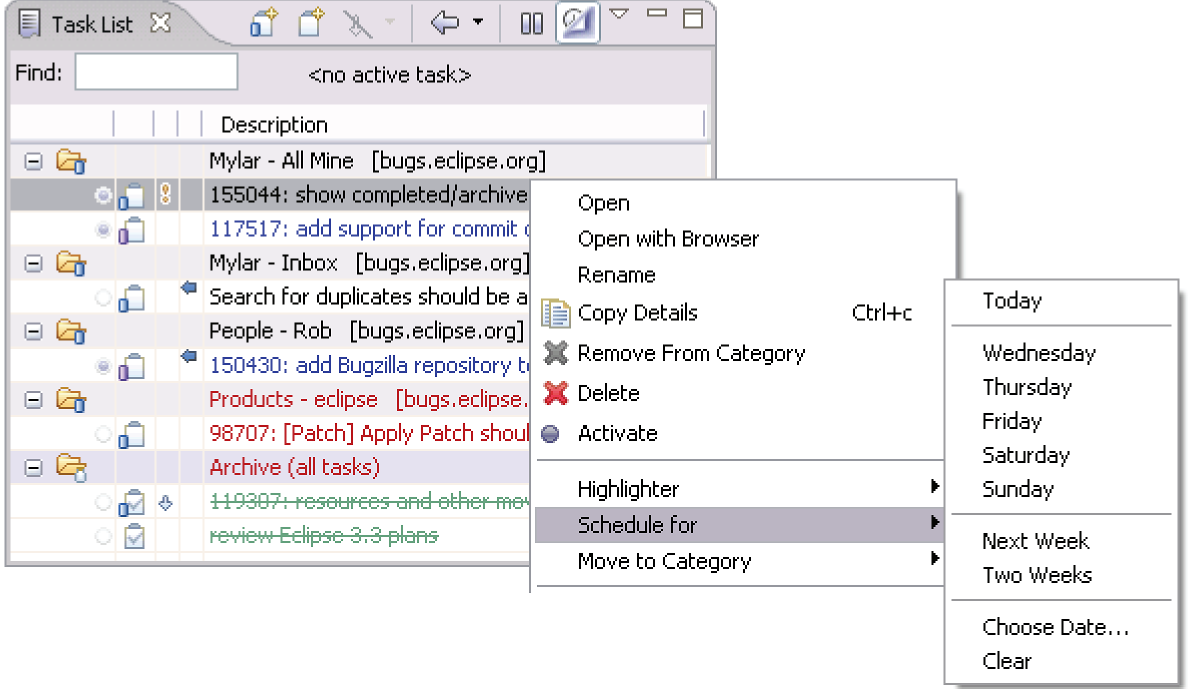

Figure 5.9: Focusing the work week

The key difference with the way that the task activity context is projected is that it uses a binary notion of decay: either a task's recency places it in the current workweek or it does not. It also uses a discrete notion of recency: a task is active this week, active today, active now. Although this has the benefit of predictability, it misses a considerable part of the benefit of our interaction-based weighting and forces the user to constantly manage their task list to ensure that not too much is scheduled for any given day or week, and that tasks with incoming comments are read so as not to pollute the view. We believe that converging these mechanisms with those used for the artifacts themselves could automate some of the manual task list management currently imposed on the user; this is left to future work.

6. Related Work

Task contexts capture information that is relevant to a knowledge worker's tasks. In this chapter, we build on our summary of other approaches (Section 1.3) and compare task contexts to related work that can identify and represent task information, including query and recommender tools (Section 6.1). Some of these approaches include mechanisms that take into account a worker's interaction with information to implicitly create a context (Section 6.2). We also compare our approach to task management tools (Section 6.3) and information focusing user interfaces (Section 6.4). Our comparisons include approaches that target programmers specifically and approaches that target a more general class of knowledge workers.

6.1 Mechanisms for Identifying Relevant Structure

When performing a task that involves understanding a subset of a large information system, hierarchical decompositions alone are often insufficient [42]. In this section, we describe query and concern management tools that have made it possible for programmers to find information related to their task. We also describe how the task context model builds on this previous work by providing a novel data structure that makes interaction explicit while also representing the elements and relations of the underlying system that correspond to the interaction. We describe how our application of task context to focus a user interface goes beyond the models currently being used by search engines and collaborative filters in order to show the user the most relevant task-specific information.

6.1.1. Query and Concern Management Tools for Programmers

A programmer can use traditional query tools, such as grep or a program database [67], to find individual points in the system structure in which he is interested. The idea of "slices" took the idea of a query further, recognizing that a collection of related points in the code is often the desired result. For instance, Weiser's original notion of a slice identified all code affecting (or affected by) a particular variable at a particular point [70]. Many now use the term "concern" to describe a collection of such points. Concern management tools support a more flexible definition of slices by including points in the system based on modularity properties, external specifications, or annotations in the code. For example, JQuery [33] provides a comprehensive query language and UI for expressing, storing, and viewing slices based on modularity. Concern Graphs [59] and CAT [28] use an external specification that represents the key structure of code contributing to a concern. The AOIG tool extends search heuristics beyond the system

structure to include natural language content in related artifacts such as comments and annotations in the code [60].

All of these approaches put the burden of accurately formulating queries or annotations on the programmer. In contrast to the structure-centric approach of most query, slice, and concern tools, task contexts are an interaction-centric approach. NaCIN is similar in constructing a Concern Graph from a programmer's navigation activity [48]. Whereas NaCIN's model is a lazily-created subgraph of system structure, task context builds on this idea and instead makes the primary abstractions in the model be based on interaction, preserving interaction events in order to determine usage-based weightings. Our approach is also novel in enabling interaction-based weightings of both the elements and the relations accessed, representing indirect interactions, and supporting operations on task context such as inducing the interest of elements that have not yet been the target of interaction.

6.1.2. Search Engines and Collaborative Filtering

The most notable difference in the information that knowledge workers access compared to programmers is that the information accessed by a general knowledge worker tends to contain more unstructured information (e.g., in the form of Microsoft Word documents or image files). Query tools that use filenames and text indices often return a very large number of hits when queries are made on the little structure that is available. Similar to programming tools, approaches for improving the relevance ranking of query results have focused on adding additional heuristics based on the structure of the data, such as Google's Page Rank algorithm that weighs results based on hyperlink structure [8]. The Squeal tool [62] has demonstrated that additional structure can be mined directly from web document themselves, for example by mining sections within pages. The Semantic Web roadmap [6] promises to make more of the structure embedded in web pages explicit.

An alternative to relying on a well-defined structure is to make it easy for users to add their own metadata by providing affordances for creating and sharing tags, as social bookmarking tools such as Flickr's photo tagging or del.ic.ious' bookmark tagging [27]. Such tools facilitate the sharing of metadata to increase the accuracy of the hits returned by search engines or recommender systems. Collaborative filter recommenders have also been demonstrated in the IDE domain by tagSEA [63].

These search and tagging approaches typically burden the knowledge worker with accurately tagging elements and formulating queries, whereas task context determines the relevant structure automatically from monitoring interaction. Systems that employ user activity to support collaborative filtering are a notable exception. For example, the amazon.com item-to-item recommender system uses an implicit notion of activity [46], the purchasing of a product, to inform the recommender. Although task context builds on this idea of weighting via interaction, it makes both interaction and the underlying structural

elements and relations a part of the model. This enables task context to be extended to any domain structure that can be represented as elements and relations and supports uniformly weighting interaction with different domain structures. Other key differentiators are the explicit representation of users' tasks, and the operations that task context supports, such as inducing interest on structurally related elements.

6.2 Using Interaction to Create a Context

In contrast to structural approaches for creating a context, several tools have employed a user's interaction with an application to formulate the set of relevant elements. Such interaction-based contexts have been used for focusing displays on the relevant information (Section 6.2.1) as well as improving search results (Section 6.2.3). We also present alternative approaches to mining user interaction (Section 6.2.4)

6.2.1. Displaying Usage-Based Context

Many IDE tools show the programmer structural context for the currently selected element, starting perhaps with Interlisp's Masterscope [67]. Providing a richer context than the currently selected element can be accomplished by monitoring and recording the user's interaction. For example, in the document editing domain, the Edit and Read Wear tool was one of the first to do this by highlighting editing and selection patterns that a user had performed on a document to indicate areas of high interest and change [31]. Mylar 0.1 [39] and Wear-Based Filtering [16] (developed independently in parallel) expanded this to the programming domain by using interaction frequency to highlight the elements of interest in the IDE UI and filter away uninteresting elements. Both Mylar 0.1 and Wear-Based Filtering had an overly simplistic model of interest decay, did not represent weightings of relations, and lacked the ability to perform operations such as interest predictions and propagations.

Team Tracks used interaction with program elements gathered across multiple uses by multiple users to drive a recommender that can suggest to other team members which program elements may be of interest [15]. Parnin uses an alternative to Mylar's user interaction-based DOI function [53] for the similar purpose of providing recommendations in a team environment, and extends interaction monitoring to include the queries that the user has formulated (e.g., via searching for all references to a particular method). A recent Focus+Context approach to UML visualization, in which "A class is displayed at a particular level of detail using a degree of interest (DOI) function based on the frequency of access to a particular class and its distance from the current object in focus", also captured interaction to focus the user interface [32].

The key shortcoming of these approaches is that they do not have an explicit representation of programmers' tasks. This results in information overload on long-running tasks and the context loss problems that we identified during the preliminary study of Mylar 0.1, which also lacked such a facility

(Section 4.2). In addition to making tasks explicit, the task context model is novel in capturing the interaction with the tasks themselves, enabling the display of task activity (Section 5.5.6).

6.2.2. Ranking Search Results

Recent approaches to improving information retrieval accuracy for applications, such as web search engines, have been “personalizing” search results [54], for example by incorporating interaction data into their ranking algorithms [66] [2]. Each of these employs user click-through data to improve the ranking of search results, for example, ranking web pages the users have previously visited higher than those that they have not. A key difference with task context is of the ability to weigh the various kinds of interaction with aggregate kinds of structured and semi-structured data. Task context also explicitly represents tasks, making it possible to express task-specific weightings of interaction that support interest decay, and enabling operations that treat structure-based weightings separately from interaction-based weightings. For example, in contrast to the search tools cited above, which blend interaction-based rankings and other heuristics to create a single ranked list of results, task context makes it possible to separate structure-based rankings from interaction-based ones. This separation enables distinguishing task-specific interaction-based rankings from global structure and heuristic-based rankings. For example, the content assist dialog rankings in Figure 6.1 below make this separation of interaction- and structure-based ranking clear with a visual separator. Maintaining interaction-based rankings separate from structure-based rankings enables switching the interaction-based rankings with each task, while ensuring that the heuristic rankings remain a predictable view of system structure.

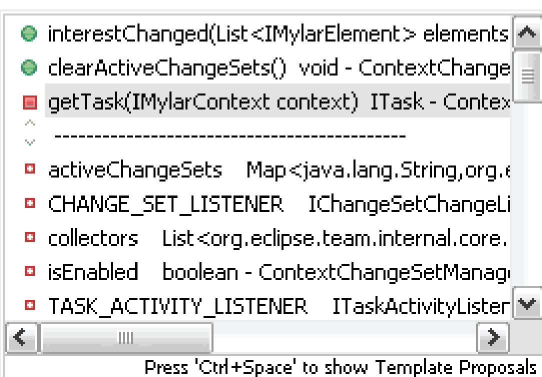

Figure 6.1: Separation of interaction-specific rankings from structure-based rankings

6.2.3. Implicit Search and Usage-Based Scoping

Implicit or automatic query facilities can provide a mechanism for driving search without the explicit need for the user to invoke the search. Such facilities can use detailed information about the user’s environment. For example, just-in-time information retrieval agents use keyword occurrences within

currently-selected documents to drive an implicit search and can recommend relevant web pages [56]. As implemented by our Active Search (Section 3.3.3), task contexts provide two additional facilities to such mechanisms. First, by virtue of defining a set of the most interesting elements in a particular context, a slice of the context, such as all landmarks, can be used as an input to implicit search. In the Active Search view, this provides stability, since the set of landmarks changes much less frequently than the currently selected element. Second, a broader slice of the context can be used to provide a usage-based search scope that is task-specific.

6.2.4. Mining User Interaction

Hilbert provided a framework for monitoring application events to determine which information was the target of interactions. He also identified the challenge of determining which parts of the interaction were relevant when monitoring is done at a fine-grained level of detail [30]. Mylar addresses this problem by providing weightings based on interaction frequency and recency. Machine learning or user modeling can also be used to determine the relevant parts of a user's interaction history. Such learning approaches have been implemented for the purpose of monitoring web-based document navigation [71]. One approach used mouse movements to determine interest in the relevant area of a web-based document [11]. More recent work has extended this mouse tracking approach to create a model from additional interaction such as tracking UI commands, scrolling, and the time spent on particular pages [29]. TaskTracer has additionally used the time that a page is open, the execution copy commands, and the occurrences of words in the contents of the page to determine relevance [44]. These above approaches employ Bayesian networks or other classifier and machine learning techniques to predict the relevance of documents and other information to the user [72].

Although Mylar has rich facilities for monitoring user interaction, a driving design goal is to make the transformation from interaction to the model predictable to the user. Additional study is needed to determine whether predictability can be preserved when employing finer-grained and more sophisticated monitoring mechanisms such as cursor position, time spent on a document, or the visible areas of scrollable pane, as has been done by the above approaches. In addition, our choice not to use sophisticated and statistical learning models for creating a task context was deliberate. We wanted to ensure that the task context model did not exhibit the failures of adaptive and adaptable menus [23], whose lack of predictability made them difficult to use. However, there are other areas of where adaptability could potentially enhance task context model accuracy. Additional testing will be needed to determine if machine learning approaches can provide improved accuracy while maintaining predictability.

6.3 Task Management Tools

Some of the foundations on managing the context of documents in a task-centric way come from the Placeless and Presto projects from Xerox PARC [17], which supported a flexible configuration of documents and provided the “Vista” display of tagged. This supported organizing and browsing files by user-defined categories and topics, making it possible to avoid navigating through deep hierarchies. This approach is now available in the Microsoft Windows Vista [68] operating system. However, these approaches require users to manually categorize their files rather than building up context implicitly. Additional early work in supporting task models and task switching focused on the definition of explicit structure for the tasks (e.g., [9]). Other systems have taken the approach of extracting the definitions and contents of tasks from relevant information sources, such as email (e.g., [5]).

Our approach is most similar to those that make the association of information with a task and the determination of the tasks themselves as unobtrusive as possible (e.g., UMEA [34] and TaskTracer [18] [64]. UMEA and TaskTracer both monitor the user’s interaction and create a listing of the elements interacted with as part of each task, in the case of TaskTracer, and as part of each project, in the case of UMEA. In these systems, the resources associated with a task increases with the amount of interaction unless a user takes explicit action to delete information from the task. Various mechanisms for determining relevance of a piece of information with a task are discussed in this earlier work, but little empirical evidence of experience with these mechanisms is provided. Our work differs in providing DOI values for each piece of information accessed and in maintaining structural links between information traversed, based on the frequency and recency of the interaction during the task. This makes it possible to present a display of task context that does not grow indefinitely on long-running tasks, since the least relevant elements are decayed out of view automatically.

The only other work we are aware of in trimming the documents associated with a task automatically is by Lettkeman et al. [44]. They tested, within the TaskTracer environment, whether machine learning could predict which web pages accessed as part of a task were likely to be revisited. Their work did not consider file interactions as part of tasks; however, we have shown these interactions to be useful in our field study of knowledge workers. In the future, it would be interesting to compare the precision of a decay feature based on interaction with a statistical approach.

---

[68] http://microsoft.com/windowsvista/

6.4 Information Focusing User Interfaces

The idea of using a DOI function to control which parts of a large set of structured data should be displayed to the user originated with Focus+Context and fisheye views [45]. In this section, we discuss how our work relates to information visualization in general and to related efforts in software visualizations.

6.4.1. Information Visualization

One goal of task context is to improve the density of IDE and information browsing views. However, whereas Card's DOI [9] is purely a function of the tree structure used to determine which parts of the structure to display, the DOI function for task context is based on a user's interaction history and supports a graph structure. Task context extends the notion of DOI beyond navigation by supporting the encoding of the interest of relationships and interaction-based manipulation of DOI values. By virtue of providing a DOI function that defines the elements of high interest, task context is related to Focus+Context views, such as those used for navigating large hierarchies [43]. Our DOI function provides weightings based on interaction with structure, not on a weighting that is purely defined by structural properties alone, as is the case with the above approaches. Task context employs a general interaction-based DOI that can provide an alternative input to such Focus+Context visualization, as discussed in Section 5.4.1.

Force-directed graph drawing [21] [25] provides some of the basis for how interest is induced along relationships since task context treats the transitive effects of the induction operation similar to the way force-directed graphs compute layout. The use of force-directed techniques has been demonstrated for rendering diagrams of software structure with a UML representation [20]. The notion of weighted relationships, used in task context by the DOI level of relations, is also used in the VisualThesaurus tool [24]. However, similarly to the other existing DOI work, the underlying model that these existing approaches take is a function of the structure and not of the interaction history.

Part of the original inspiration for task context was to make navigating software structure as intuitive as it is to navigate physical topographies[69]. As such, concepts from wayfinding and cognitive maps are represented in Mylar's UI mechanisms. The ideas of landmarks and paths originate from Kevin Lynch's work on urban planning [47]. Lynch's work has been adapted to navigating information by Vinson, who also used the ideas of landmarks and paths for navigating virtual information spaces [69]. Vinson discovered that paths are an emergent phenomenon and arise from the development of spatial knowledge

[69] Our original term for task context was "taskscape" in order to connote this topographical property. Since we have de-coupled the model from its visual representation, we now only use the term taskscape for topographical visualizations of task context.

for virtual environments in a very similar way to how they emerge for real environments. This implicit representation and emergence of paths is also present in the task context model (Section 2.2). Moonen extended the idea of wayfinding in cities to exploring software systems, but his work focused on not on interaction history but on relating software structure to the concept of landmarks, nodes, paths, districts, and edges [49].

6.4.2. Software Visualization

In addition to the generally applicable information visualization approaches described above, some software visualization tools have used related Focus+Context ideas to reduce information overload in displays of large software systems. The Argo/UML visualization tool has a notion of mapping "to do" items to files, but these all fall short of a task representation as they only provide program element specific tags [65]. A structure-based notion of DOI has been used for distortion-based techniques of visualizing system structure in terms of data flow and entity-relationship diagrams [68]. These tools manifest the same limitations of structure-centric approaches as discussed above. The work on Instability Visualization and Analysis [7] bears some similarity to our own because the visualization of version control information takes the shape of a surface, and because this surface is constructed with a force-directed mechanism that is based on the structure of the revisions. This use of revision history provides a very coarse form of interaction that is not suitable for an interactive tool (e.g., an element may not populate the context until it is committed to revision control). Our task context model provides a finer-grained representation of interaction.

7. Conclusion

Information overload and loss of context are a bottleneck on the productivity of knowledge workers. Current tools make it easy to browse and query the structure of the information that knowledge workers need to access. However, given the complexity of today's information systems, knowledge workers end up spending an inordinate amount of time looking for the information relevant to the task-at-hand. As they are constantly multi-tasking, they are also burdened with repeatedly creating and recreating the context they need to get work done.

Our approach is to move the burden of finding and identifying the relevant information from the user to a software tool. We leverage users' episodic memory by having them indicate and recall the tasks that form their units of work. In support of this approach, we have created a model of task context, defined user interface and interaction mechanisms for integrating this model with knowledge work applications, and implemented it for an IDE and for a file and web browsing tool. Through a field study of professional programmers, we have demonstrated with statistical significance that an explicit task context makes the programmer class of knowledge workers more productive. Through a study of knowledge workers from a cross-section of professions, we have demonstrated that the model generalizes to supporting interaction with less well-structured information systems and that, when provided with the benefits of an explicit task context, knowledge workers will voluntarily indicate the tasks on which they work.

This thesis makes the following contributions to the field of software engineering, and the broader field of knowledge work.

First, we provide a generic task context model that represents interaction with any structured or semi-structured data that can be represented as elements or relations. The elements and relations in this model are weighted based on the frequency and recency of interaction with those elements and relations. We demonstrate that this weighting is key to reducing information overload and that capturing context per-task is key to reducing loss of context when multi-tasking.

Second, we provide operations on task context that support composing and slicing task contexts in order to integrate the model with views and tools that display system structure. We provide the induction operation for growing the model to encompass structurally related elements and to support implicit search facilities. We also provide operations that support direct manipulation of the interest levels in the model. We demonstrate with our task-focused UI mechanisms that these operations support making task context explicit in an IDE and in a file and web browsing application.

Third, we provide a specific instantiation of the model for Java, XML, generic file and web document structure, as well as tasks themselves, and provide an architecture that supports integrating the model with various other kinds of domain structure. Our model can be extended to other kinds of domains and application platforms, and has already been implemented for a different application platform by another research group[70].

Fourth, we provide a monitoring and reporting framework that can be used for studying knowledge work, as has been demonstrated by the other research groups that have reused this framework[71].

We have deployed our implementation of task context for programmers widely; tens of thousands of programmers now use Mylar IDE daily, presumably because it gives them productivity benefits as we saw in our field study. In the future, we hope to see widespread adoption of this technology for facilitating knowledge work across a broad spectrum of domains. We also hope that capturing knowledge in the form of task contexts will provide long-term productivity and knowledge sharing benefits to both individuals and organizations.

---

[70] See footnote 8.

[71] http://wiki.eclipse.org/index.php/Mylar_Related_Research_Projects [verified 2006-10-02]

Bibliography

1. Merriam-Webster's collegiate dictionary, Merriam-Webster, Springfield, MA, 2003.

2. Agichtein, E., Brill, E., Dumais, S. and Ragno, R. Learning user interaction models for predicting web search result preferences. *Proceedings of the 29th Annual International ACM Conference on Research & Development on Information Retrieval*, ACM Press, Seattle, Washington, USA, 2006, 3-10.

3. Backus, J.W. Automatic programming: properties and performance of FORTRAN systems I and II. *Proceedings of the Symposium on the Mechanisation of Thought Processes*, ACM Press, Los Angeles, California, USA, 1958, 165-180.

4. Bellotti, V., Dalal, B., Good, N., Flynn, P., Bobrow, D.G. and Ducheneaut, N. What a to-do: studies of task management towards the design of a personal task list manager. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, Vienna, Austria, 2004, 735-742.

5. Bellotti, V., Ducheneaut, N., Howard, M. and Smith, I. Taking email to task: the design and evaluation of a task management centered email tool. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, Ft. Lauderdale, Florida, USA, 2003, 345-352.

6. Berners-Lee, T. Semantic Web Road Map, http://www.w3.org/DesignIssues/Semantic, 1998.

7. Bevan, J. and Whitehead, E.J. Identification of software instabilities. *Proceedings of the 10TH Working Conference on Reverse Engineering*, IEEE, Victoria, Canada, 2003, 134-144.

8. Brin, S. and Page, L. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, *30* (1-7). 107-117.

9. Card, S.K. and Jr., A.H. A multiple, virtual-workspace interface to support user task switching. *Proceedings of the SIGCHI/GI conference on Human factors in computing systems and graphics interface*, ACM Press, Toronto, Canada, 1987, 53-59.

10. Cheng, L.-T., Hupfer, S., Ross, S. and Patterson, J. Jazzing up Eclipse with collaborative tools. *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, ACM Press, Anaheim, California, USA, 2003, 45-49.

11. Claypool, M., Le, P., Wased, M. and Brown, D. Implicit interest indicators. *Proceedings of the 6th international conference on Intelligent user interfaces*, ACM Press, Santa Fe, New Mexico, USA, 2001, 33-40.

12. d'Entremont, T. and Storey, M.-A. Using a Degree-of-Interest Model for Adaptive Visualizations in Protégé. *9th Intl. Protégé Conference*, Stanford, California, USA, 2006.

13. Dahl, O.-J., Myhrhaug, B. and Nygaard, K. Some features of the SIMULA 67 language. *Proceedings of the second conference on Applications of simulations*, Winter Simulation Conference, New York, New York, USA, 1968, 29-31.

14. de Alwis, B. and Murphy, G.C. Using visual momentum to explain disorientation in the Eclipse IDE. *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, Brighton, UK, 2006, 51-54.

15. DeLine, R., Czerwinski, M. and Robertson, G. Easing Program Comprehension by Sharing Navigation Data. *Proceedings of the Visual Languages and Human-Centric Computing*, Dallas, Texas, USA, 2005, 241-248.

16. DeLine, R., Khella, A., Czerwinski, M. and Robertson, G., Towards understanding programs through wear-based filtering. *Proceedings of the 2005 ACM symposium on Software visualization*, (St. Louis, Missouri, USA, 2005), ACM Press, 183-192.

17. Dourish, P., Edwards, K.W., LaMarca, A. and Salisbury, M., Using properties for uniform interaction in the Presto document system. *Proceedings of the 12th annual ACM symposium on User interface software and technology*, (Asheville, North Carolina, USA, 1999), ACM Press, 55-64.

18. Dragunov, A.N., Dietterich, T.G., Johnsrude, K., McLaughlin, M., Li, L. and Herlocker, J.L., TaskTracer: a desktop environment to support multi-tasking knowledge workers. *Proceedings of the 10th international conference on Intelligent user interfaces*, (San Diego, California, USA, 2005), ACM Press, 75-82.

19. Drucker, P.F. *Managing the knowledge worker*. Modern Office Procedures, 1979.

20. Dwyer, T., Three dimensional UML using force directed layout. *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation - Volume 9*, (Sydney, Australia, 2001), Australian Computer Society, Inc., 77-85.

21. Eades, P. A Heuristic for Graph Drawing. *Congressus Numerantium*, *42*. 149-160.

22. Eick, S.G., Steffen, J.L. and Sumner, E.E. Seesoft--A tool for visualizing line oriented software statistics. *Proceedings of the SIGCHI conference on Human factors in computing systems*, *18* (11). 957-968.

23. Findlater, L. and McGrenere, J., A comparison of static, adaptive, and adaptable menus. *Proceedings of the SIGCHI conference on Human factors in computing systems*, (Vienna, Austria, 2004), ACM Press, 89-96.

24. Fowler, R.H., Fowler, W.A.L. and Wilson, B.A., Integrating query thesaurus, and documents through a common visual representation. *Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*, (Chicago, Illinois, USA, 1991), ACM Press, 142-151.

25. Fruchterman, T.M.J. and Reingold, E.M. Graph Drawing by Force-directed Placement. *Software - Practice and Experience*, *21* (11). 1129-1164.

26. Gonzales, V.M. and Mark, G. "Constant, constant, multi-tasking craziness": managing multiple working spheres. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, Vienna, Austria, 2004, 113-120.

27. Hammond, T., Hannay, T., Lund, B. and Scott, J. Social bookmarking tools (I). *D-Lib Magazine*, *11* (4). 1082-9873.

28. Harrison, W., Ossher, H., Tarr, P., Kruskal, V. and Tip, F. CAT: A Toolkit for Assembling Concerns, IBM, Yorktown Heights, New York, USA, 2002.

29. Hijikata, Y., Implicit user profiling for on demand relevance feedback. *Proceedings of the 9th international conference on Intelligent user interface*, (Funchal, Madeira, Portugal, 2004), ACM Press, 198-205.

30. Hilbert, D.M. and Redmiles, D.F. Separating the wheat from the chaff in Internet-mediated user feedback expectation-driven event monitoring. *ACM SIGGROUP Bulletin*, *20* (1). 35-40.

31. Hill, W.C., Hollan, J.D., Wroblewski, D. and McCandless, T. Edit wear and read wear. *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM Press, Monterey, California, USA, 1992, 3-9.

32. Jacobs, T. and Musial, B., Interactive visual debugging with UML. *Proceedings of the 2003 ACM symposium on Software visualization*, (San Diego, California, USA, 2003), ACM Press, 115-122.

33. Janzen, D. and Volder, K.D., Navigating and querying code without getting lost. *Proceedings of the 2nd international conference on Aspect-oriented software development*, (Boston, Massachusetts, USA, 2003), ACM Press, 178-187.

34. Kaptelinin, V., UMEA: translating interaction histories into project contexts. *Proceedings of the SIGCHI conference on Human factors in computing systems*, (Ft. Lauderdale, Florida, USA, 2003), ACM Press, 353-360

35. Kersten, M. Lessons learned building tool support for AspectJ. *AOSD Conference Industry Track*, Bonn, Germany, 2006.

36. Kersten, M. Task-focused programming with Mylar, Part 1 *Streamline your work with integrated task management for Eclipse*, IBM DeveloperWorks, 2006.

37. Kersten, M. Task-focused programming with Mylar, Part 2 *Enhance your productivity with automatic context management for Eclipse*, IBM DeveloperWorks, 2006.

38. Kersten, M., Elves, R. and Murphy, G.C. WYSIWYN: Using Task Focus to Ease Collaboration *Submitted to CHI 2007*, 2006.

39. Kersten, M. and Murphy, G.C. Mylar: a degree-of-interest model for IDEs. *Proceedings of the 4th international conference on Aspect-oriented software development*, ACM Press, Chicago, Illinois, USA, 2005, 159-168.

40. Kersten, M. and Murphy, G.C. Task Contexts for Knowledge Workers: A Field Study. *Submitted to CSCW Workshop on Supporting the Social Side of Large-Scale Software Development*, Banff, Canada, 2006.

41. Kersten, M. and Murphy, G.C. Using Task Context to Improve Programmer Productivity. *To appear in proceedings of the 14th ACM SIGSOFT Symposium on Foundations of Software Engineering* ACM Press, Portland, OR, USA, 2006.

42. Kiczales, G., et al, Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming*, (Jyväskylä, Finland, 1997), Springer-Verlag, 220-242.

43. Lamping, J., Rao, R. and Pirolli, P., A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. *Proceedings of the SIGCHI conference on Human factors in computing systems*, (Denver, Colorado, USA, 1995), ACM Press/Addison-Wesley Publishing Co., 401-408.

44. Lettkeman, A.T., Stumpf, S., Irvine, J. and Herlocker, J. Predicting Task-Specific Webpages for Revisiting. *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, Boston, Massachusetts, USA, 2006.

45. Leung, Y.K. and Apperley, M.D. A review and taxonomy of distortion-oriented presentation techniques. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *1* (2). 126-160.

46. Linden, G., Smith, B. and York, J. Amazon. com recommendations: item-to-item collaborative filtering. *Internet Computing, IEEE*, *7* (1). 76-80.

47. Lynch, K. *The Image of the City*. MIT Press, Cambridge, USA, 1960.

48. Majid, I. and Robillard, M.P., NaCIN: an Eclipse plug-in for program navigation-based concern inference. *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, (San Diego, California, USA, 2005), ACM Press, 70-74.

49. Moonen, L., Exploring software systems. *International Conference on Software Maintenance (ICSM)*, (Amsterdam, The Netherlands, 2003), 276-280.

50. Munzner, T., Francois Guimbretiere, Tasiran, S., Zhang, L. and Zhou, Y. TreeJuxtaposer: scalable tree comparison using Focus+Context with guaranteed visibility. *ACM Transactions on Graphics (TOG)*, *22* (3). 453-462.

51. Murphy, G., Kersten, M., Robillard, M. and Cubranic, D., The Emergent Structure of Development Tasks. *Proceedings of the European Conference on Object-Oriented Programming*, (Glasgow, Scotland, 2005), 33-48.

52. Murphy, G.C., Kersten, M. and Findlater, L. How are Java software developers using the Eclipse IDE? *IEEE Software*, *23* (4). 76-83.

53. Parnin, C. and Gorg, C. Building Usage Contexts during Program Comprehension. *Proceedings of the 14th IEEE International Conference on Program Comprehension*, IEEE, Athens, Greece, 2006.

54. Pitkow, J., Schutze, H., Cass, T., Cooley, R., Turnbull, D., Edmonds, A., Adar, E. and Breuel, T. Personalized search. *Communications of the ACM*, *45* (9). 50-55.

55. Plotnik, R. *Introduction to Psychology*. Wadsworth Publishing Company, 2004.

56. Rhodes, B.J. and Maes, P. Just-in-time information retrieval agents. *IBM Systems Journal*, 2000, 685-704.

57. Rieman, J. A field study of exploratory learning strategies. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *3* (3). 189-218.

58. Robillard, M.P., Automatic Generation of Suggestions for Program Investigation. *Proceedings of the Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, (Lisbon, Portugal, 2005), 11-20.

59. Robillard, M.P. and Murphy, G.C., Concern graphs: finding and describing concerns using structural program dependencies. *Proceedings of the 24th International Conference on Software Engineering*, (Orlando, Florida, 2002), ACM Press, 406-416.

60. Shepherd, D., Pollock, L. and Vijay-Shanker, K., Towards supporting on-demand virtual remodularization using program graphs. *Proceedings of the 5th international conference on Aspect-oriented software development*, (Bonn, Germany, 2006), ACM Press, 3-14.

61. Snowden, J.S. Semantic-Episodic Memory Interactions in Semantic Dementia: Implications for Retrograde Memory Function. *Cognitive Neuropsychology*, *13* (8). 1101-1139.

62. Spertus, E. and Stein, L.A. Squeal: a structured query language for the Web. *Computer Networks*, *33* (1-6). 95-103.

63. Storey, M.-A., Cheng, L.-T., Bull, I. and Rigby, P., Waypointing and social tagging to support program navigation. *Proceedings of CHI '06 extended abstracts on Human factors in computing systems*, (Montréal, Canada, 2006), ACM Press, 1367-1372.

64. Stumpf, S., Bao, X., Dragunov, A., Dietterich, T.G., Herlocker, J., Johnsrude, K., Li, L. and Shen, J.Q. Predicting User Tasks: I Know What You're Doing. *20th National Conference on Artificial Intelligence (AAAI)*, Pittsburgh, USA, 2005.

65. Taylor, R. and Redmiles, D. Argo/UML. *ACM SIGSOFT Software Engineering Notes*, ACM Press, 2000, 97.

66. Teevan, J., Dumais, S.T. and Horvitz, E., Personalizing search via automated analysis of interests and activities. *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, (Salvador, Brazil, 2005), ACM Press, 449-456.

67. Teitelman, W. and Masinter, L. The Interlisp programming environment. *IEEE Computer*, *14*. 25-34.

68. Turetken, O., Schuff, D., Sharda, R. and Ow, T.T. Supporting systems analysis and design through fisheye views. *Communications of the ACM*, ACM Press, 2004, 72-77.

69. Vinson, N.G., Design guidelines for landmarks to support navigation in virtual environments. *Proceedings of the SIGCHI conference on Human factors in computing systems: the CHI is the limit*, (Pittsburgh, Pennsylvania, United States, 1999), ACM Press, 278-285.

70. Weiser, M. Programmers use slices when debugging. *Communications of the ACM*, *25* (7). 446-452.

71. Widyantoro, D.H., Ioerger, T.R. and Yen, J., An adaptive algorithm for learning changes in user interests. *Proceedings of the eighth international conference on Information and knowledge management*, (Kansas City, Missouri, USA, 1999), ACM Press, 405-412.

72. Wolverton, M. Task-based information management. *ACM Computing Surveys (CSUR)*, *31* (2es). 10-10.

Appendix A User Studies

Ethics Certifications

The University of British Columbia
Office of Research Services and Administration
Behavioural Research Ethics Board

Certificate of Approval

PRINCIPAL INVESTIGATOR	DEPARTMENT	NUMBER
Murphy, G.C.	Computer Science	**B05-0348**

INSTITUTION(S) WHERE RESEARCH WILL BE CARRIED OUT

CO-INVESTIGATORS:
Kersten, Mik, Computer Science

SPONSORING AGENCIES
IBM Canada Limited

TITLE:
Reducing Information Overload for Software Developers through Active Search

APPROVAL DATE	TERM (YEARS)	DOCUMENTS INCLUDED IN THIS APPROVAL:
MAY 24 2005	1	May 16, 2005, Cover letter / Consent form / April 12, 2005, Advertisement / Questionnaires

CERTIFICATION:

The protocol describing the above-named project has been reviewed by the Committee and the experimental procedures were found to be acceptable on ethical grounds for research involving human subjects.

Approval of the Behavioural Research Ethics Board by one of the following:
Dr. James Frankish, Chair,
Dr. Cay Holbrook, Associate Chair,
Dr. Susan Rowley, Associate Chair

This Certificate of Approval is valid for the above term provided there is no change in the experimental procedures

The University of British Columbia
Office of Research Services and Administration
Behavioural Research Ethics Board

Certificate of Approval

PRINCIPAL INVESTIGATOR	DEPARTMENT	NUMBER
Murphy, G.C.	Computer Science	**B05-0348**

INSTITUTION(S) WHERE RESEARCH WILL BE CARRIED OUT
CO-INVESTIGATORS:
Kersten, Mik, Computer Science
SPONSORING AGENCIES
IBM Canada Limited
TITLE:
Reducing Information Overload for Software Developers through Active Search

APPROVAL RENEWED DATE	TERM (YEARS)	AMENDMENT:	AMENDMENT APPROVED:
MAY 1 1 2006	1	Apr. 28, 2006, Subjects / Recruitment method / Procedures / Study location / Recruitment letter / Consent form / Interviews	MAY 1 1 2006

CERTIFICATION:

The request for continuing review and amendment of the above-named project has been reviewed and the procedures were found to be acceptable on ethical grounds for research involving human subjects.

Approved on behalf of the Behavioural Research Ethics Board
by one of the following:
Dr. Peter Suedfeld, Chair,
Dr. Susan Rowley, Associate Chair
Dr. Jim Rupert, Associate Chair
Dr. Arminee Kazanjian, Associate Chair

This Certificate of Approval is valid for the above term provided there is no change in the experimental procedures

Study I Questionnaire

Subjects of Study 1 (Section 4.2) answered the following questions before starting the study.

- How much development experience do you have? How much Java development?
- Briefly describe your development responsibilities.
- Would you describe yourself as an Eclipse beginner, intermediate, or expert user? How much of your day do you spend in Eclipse?
- What do you like about Eclipse's navigation mechanisms and tree views? What don't you like?
- Do you find that views such as the Package Explorer and document outline contain too much information? Is it cumbersome to manage the large trees?
- For browsing class members do you use the Document Outline, the Package Explorer, both, or something else?
- What do you think of Eclipse's current support for presenting and managing open editors?
- Do you use the navigation history commands? Are you ever unclear about where the back or forward command will take you?
- Do you use the editor's folding feature? If so, does it cause you problems when code you were expecting to see is folded away?
- Please list anything else particular about your navigation style, and any gripes that you have with Eclipse's current support for your navigation needs.