An Active Help System to Improve Program Navigation

by

Petcharat Viriyakattiyaporn

B.Sc., The University of Toronto, 2007

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

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April, 2009

Abstract

Software development environments have evolved to make it easy for third parties to integrate a variety of tools into the environment. Previous research has shown that developers often do not use or know all of the tools available in development environments that they regularly use. The most common solution to alleviate this problem is to provide a means to search through passive help documents. However, this approach requires a developer to be able to express her desires in a form understood by a search engine.

To overcome this limitation, we propose using an active help system to automatically recommend to a developer tools that are appropriate to the developer's current goal. We introduce an active help system to recommend navigation tools because previous research has shown a substantial amount of effort expended by a developer on a software evolution task is related to navigation. Our approach infers which structurally related pieces of information are of interest to a developer and determines when the navigation performed by the developer between the related information is suboptimal. In suboptimal navigation situations, our approach recommends a tool that would enable more efficient navigation.

Our approach was implemented as a prototypical active help system, called Spyglass, in IBM's Rational Team Concert development environment. To investigate whether and under what conditions Spyglass helps or hinders a developer in navigation, two empirical studies were conducted: a longitudinal user study in an undergraduate software engineering course and a laboratory user study comparing Spyglass to a passive tutorial. The longitudinal study provided feedback to improve the user interface of Spyglass and provided evidence that the developers who were novice to the development environment liked the idea of having an active help system in the environment. The laboratory study confirmed that Spyglass can help developers navigate between code elements more efficiently. The developers in the laboratory study also confirmed their desire for a system like Spyglass. The laboratory study also revealed that Spyglass requires improvements to its accuracy.

Table of Contents

Abstract ii

Table of Contents iii

List of Tables vii

List of Figures ix

Acknowledgements x

Dedication xi

1 Introduction 1

2 Related Work 4
- 2.1 Active Help Systems 4
- 2.2 Active Help in Software Development Environments 6

3 Spyglass: The Approach 7
- 3.1 Domain Artifacts 7
- 3.2 User Behaviour Analysis Over Time 8
- 3.3 Inferring Navigation Source and Target 9
- 3.4 Selecting Tools to Recommend 11
- 3.5 Filtering Potential Recommendations 14

4 Spyglass: The Implementation 15
- 4.1 The Core 16
  - 4.1.1 Knowledge Base 16
  - 4.1.2 Monitoring Component 20
  - 4.1.3 Computation Component 21
- 4.2 The User Interface 22

5 Longitudinal User Study 25
5.1 Spyglass 0.9 25
5.2 Participants 27
5.3 Experimental Design 29
5.4 Experimental Procedure 30
5.4.1 Collecting Data 30
5.5 Results 32
5.6 Quantitative Results 32
5.6.1 Log File Analysis 33
5.6.2 Post-Experiment Questionnaire Analysis 38
5.7 Qualitative Results 40
5.8 Summary 41

6 Laboratory User Study 43
6.1 Participants 44
6.2 Experimental Design 45
6.3 Experimental Procedure 45
6.4 Results 48
6.4.1 Quantitative Results 53
6.4.2 Qualitative Results 64
6.5 Results Summary 66
6.6 Threats to Validity 67
6.6.1 Construct Validity 67
6.6.2 Internal Validity 67
6.6.3 External Validity 68

7 Discussion 69
7.1 Approach Improvements 69
7.2 UI Improvements 70
7.3 Study Improvements 70

8 Conclusion and Future Work 72

Bibliography 74

Appendices

A BREB Approval Certificate 77
A.1 For the Longitudinal Study 77
A.2 For the Laboratory Study 78

B Additional Materials Related to the Longitudinal Study 80
B.1 Consent Form 80
B.2 Preliminary Questionnaire 83
B.3 Post-Experiment Questionnaire 85
B.4 Spyglass Introduction 87
B.5 The First Code-Exploration Exercise 91
B.6 The Second Code-Exploration Exercise 95
B.7 Recurring Questionnaire 99
B.8 Interview Protocol 100

C Additional Materials Related to the Laboratory Study 101
C.1 Consent Form 101
C.2 Preliminary Questionnaire 104
C.3 Post-Experiment Questionnaire for the Spyglass Group 106
C.4 Post-Experiment Questionnaire for the Tutorial Group 109
C.5 JFreeChart Introduction Document 111
C.6 Eclipse-Based Tools Tutorial for the Spyglass Group 114
C.7 Eclipse-Based Tools Tutorial for the Tutorial Group 117
C.8 Spyglass Introduction 120
C.9 Training Tasks 122
C.10 Tasks for the Programming Session 123
C.11 List of Base Code Elements for Task Completion 127
C.11.1 Task 1 127
C.11.2 Task 2 127
C.12 Marking Scheme for Task Completion 129
C.12.1 Task 1: Fix Pie Chart Labels 129
C.12.2 Task 2: Fix Bar Chart Labels 130
C.13 Criteria to Determine Relevant Recommendation 132
C.14 Interview Protocol 134
C.15 Critical Value Table for the Mann-Whitney U Test 135
C.16 Completion Scores of All Participants 136
C.17 Unrelated Code Elements Visited by All Participants 137

C.18 Target Tools Usage Frequency of All Participants 138

List of Tables

5.1 Programming Experience of Participants in Longitudinal Study 27
5.2 Software Development Experience of Participants in Longitudinal Study 28
5.3 Usage of Development Environment of Participants in Longitudinal Study 28
5.4 Expertise of Participants in Longitudinal Study 29
5.5 Interactions with Spyglass During the First Code-Exploration Phase . 34
5.6 Interactions with Target Tools During the First Code-Exploration Phase 35
5.7 Interactions with Spyglass During the Second Code-Exploration Phase 35
5.8 Interactions with Target Tools During the Second Code-Exploration Phase 36
5.9 Interactions with Spyglass During the Implementation Phase 37
5.10 Interactions with Target Tools During the Implementation Phase . . . 38
5.11 Usefulness of Recommendations and Their Rationales 38
5.12 Usefulness of Target Tools 39
5.13 User-Friendliness of Spyglass 40
5.14 User Perceptions of Spyglass 0.9 41

6.1 Programming Experience of Participants in Lab Study 44
6.2 Experience with Eclipse Environment of Participants in Lab Study . . 45
6.3 Example of Marking Scheme for Task 2 Completion 54
6.4 Completion Scores from Task 2 Used for Testing Hypothesis 1 55
6.5 Descriptive Statistics of Completion Scores Used for Testing Hypothesis 1 55
6.6 Number of Unrelated Code Elements Visited in Task 2 Used for Testing Hypothesis 2 57
6.7 Descriptive Statistics of Number of Unrelated Code Elements Used for Testing Hypothesis 2 57
6.8 Target Tools Usage in Task 2 Used for Testing Hypothesis 3 59
6.9 Descriptive Statistics of Tools Usage Frequency in Task 2 59

6.10 Descriptive Statistics of Density of Tools Usage in Task 2 59
6.11 Usefulness of Target Tools Used for Testing Hypothesis 4 60
6.12 Usefulness of Recommendations and Their Rationales 62
6.13 Disturbance of Popup and Sound Notification 64

C.1 List of the Base Code Elements for Task 1 Completion 127
C.2 List of the Base Code Elements for Task 2 Completion 128
C.3 Completion Scores of All Participants in Task 2 136
C.4 Descriptive Statistics of All Completion Scores in Task 2 136
C.5 Number of Unrelated Code Elements Visited by All Participants in Task 2 137
C.6 Descriptive Statistics of Unrelated Code Elements Visited by All in Task 2 137
C.7 Target Tools Usage of All Participants in Task 2 138
C.8 Descriptive Statistics of Target Tools Usage Frequency for All Participants in Task 2 139

List of Figures

3.1 Inferring Source and Target Artifact for a Navigation 10
3.2 A Sample Call Graph 11
3.3 Tool Output Representation Can Differ 12

4.1 Spyglass Client Architecture 15
4.2 The Open Type Hierarchy Tool Output 17
4.3 The Callers Returned by the Open Call Hierarchy Tool 18
4.4 The Callees Returned by the Open Call Hierarchy Tool 18
4.5 The Work Item Editor Links Box 19
4.6 Spyglass Popup 23
4.7 Spyglass User Interface 24

5.1 Spyglass 0.9 User Interface 27
5.2 Log Files Format 31

6.1 Recommendations and Use of Target Tools by Spyglass Group in Task 1 49
6.2 Recommendations and Use of Target Tools by Sub-Group of Tutorial Group in Task 1 50
6.3 Recommendations and Use of Target Tools by Spyglass Group in Task 2 51
6.4 Recommendations and Use of Target Tools by Sub-Group of Tutorial Group in Task 2 52
6.5 Usefulness of Target Tools 61

Acknowledgements

I would like to thank my supervisor, Dr. Gail Murphy, for her guidance, collaboration, and support during my graduate study at UBC. I feel fortunate to be her graduate student and to be able to learn from her. I would also like to thank Dr. Eric Wohlstadter for being the second reader for my thesis, despite the short notice.

I would like to thank Dr. David Shepherd for inspiring me to work on the research in this area and Dr. Emerson Murphy-Hill for helping me proofread this thesis. Also, I would like to thank my friends and colleagues at UBC who volunteered to participate in my pilot and laboratory studies.

I would like to thank the members of the Software Practices Lab who gave valueable advice regarding my research or software engineering in general, whenever I asked, especially Seonah Lee, Sarah Rastkar, and Kaitlin Duck Sherwood.

Finally, I would like to thank all my friends at St. John's College, especially Jie Yu, Amy Wei You, Zoreh Jabbari, Yumi Endo, Susana Zoghbi, and Alejandra Fosado Garcia, for giving me advice regarding my academic courses, my research, and my future career and listening to my complaints. Thank you for your valuable friendship and support throughout the years I stayed in Vancouver.

Dedication

I would like to dedicate this thesis to my Mom, Grandpa Sutjarit, and Grandma Yupin. Thank you Mom for raising me to be independent and strong, for giving me the best education you could, and for your support during the tough times. Thank you Grandpa Sutjarit and Grandma Yupin for taking care of me and my family. All these years, you gave me courage to do my best. Without you all, this achievement would not be as meaningful.

Chapter 1

Introduction

The software development environments commonly used by software developers have evolved to include extension mechanisms that make it easy to support new tools and different artifacts; for example, a web repository (Eclipse Plugin Central[1]) that lists all extensions for the Eclipse[2] development environment currently lists over 1000 available extensions ranging from support for providing diagrammatic views of code (e.g., Creole[3]) to support for editing bug reports (e.g., Mylyn[4]). Unfortunately, the developers who use these environments are not as adept at evolving as the environment themselves. As more and more tools are added, the developers are typically unable to keep up with the tools (or functionality) available to them [5]. One option to make developers aware of the functionality available is to make all tools visible in the development environment's user interface. With this approach, a developer can quickly be overwhelmed by long menus, a multitude of wizards, and so on. Alternatively, mechanisms can be added into the environment that make it possible for a developer to look up needed functionality; for example, a developer may be provided with a means of searching through help documentation. A problem with this approach is that a developer may not know how to express desired functionality appropriately to find a tool that is available, leading to a gap between the developer's presumed knowledge of available tools and the tools that are actually available [6].

To bridge this kind of gap in domains other than software development, various kinds of active help systems have been proposed (e.g., [8, 12, 14]). An active help system is a help system that acts as a knowledgeable colleague or assistant who watches over the user while he or she is using the underlying system and who introduces useful functionality that the user does not appear to know about [7].

We have been investigating an approach to creating an active help system for a software development environment to recommend appropriate tools to a developer as the developer works. Our initial focus has been on actively recommending tools to

[1] Eclipse Plugin Central:`http://www.eclipseplugincentral.com/`, verified April 16, 2009
[2] `http://www.eclipse.org/`, verified April 16, 2009
[3] Creole: `http://www.thechiselgroup.org/creole`, verified April 16, 2009
[4] Eclipse Mylyn Project: `http://www.eclipse.org/mylyn/`, verified April 16, 2009

help with navigation between structurally related pieces of information available in the development environment. We will refer to these pieces of information as *domain artifacts*. We chose to focus on navigation difficulties faced by developers because of reports that developers spend substantial effort in understanding existing code that they have to maintain [13] and that not all developers perform these understanding activities efficiently [19].

Our prototype, called Spyglass, determines suboptimal navigation by watching the information over which a developer navigates and comparing the distance between domain artifacts that are structurally related to optimal distances available if particular tools were used. Spyglass then recommends tools that could have shortened the navigation path that the developer has taken.

For an example, suppose a developer is working in a development environment with Spyglass available. The developer navigates from artifact A to artifact B, which are structurally related, by visiting several, say five, other artifacts on the way from A to B. Spyglass determines that there is a tool that could have helped the developer navigate from artifact A to artifact B directly and recommends that tool to the developer, along with a reason why the tool could help.

We hypothesize that these tool recommendations can make developers aware of the tools and how to use them. Indirectly, we expect such recommendations can help developers work more efficiently. We conducted two studies to investigate this overall hypothesis: a longitudinal study in an undergraduate software engineering course and a laboratory study comparing Spyglass to a passive tutorial.

Through the longitudinal study, we primarily learned about difficulties with the user interface of Spyglass, such as notification of recommendations not being sufficiently visible [25]. We incorporated this feedback into the version of Spyglass we used in the laboratory study. Through the laboratory study, we found that Spyglass helped improve a developer's navigation through domain artifacts (in this case, Java code elements) and was as effective for making the developer aware of new useful tools and how to use them as a tutorial document. We also confirmed through the laboratory study that the user interface of Spyglass was user-friendly, providing recommendations that were noticeable but not obtrusive to a developer. From the study, we also learned that Spyglass could benefit from improvements in its precision in giving recommendations.

This thesis makes three contributions. The first contribution is an approach for constructing an active help system for an integrated development environment that introduces navigational tools, based on a developer's navigation behaviours and the system's knowledge of structural relationships between those visited artifacts. The second contribution is a longitudinal evaluation that provided feedback about a suitable user

interface for Spyglass. The third contribution is a laboratory evaluation that provides a comparison of benefits and limitations of Spyglass to a passive help system.

Chapter 2 reviews previous efforts in active help systems, especially as they have been applied to aid software development. Chapter 3 describes the approach used in Spyglass. Chapter 4 describes the implementation of the current version of Spyglass in detail and how it has evolved from the first version. The two studies we have performed and their results are described in Chapter 5 and Chapter 6, respectively. Chapter 7 discusses the approach and the laboratory study. The last chapter concludes the thesis and discusses potential future work for this research direction.

Chapter 2

Related Work

Many approaches have been developed to help users find appropriate functionality in High Functionality Applications (HFA [6]). These approaches include passive help systems in the form of help documents or question answering systems (e.g., [8, 12, 28]); active help systems that offer guidance or advice to a user in real-time based on what the user is doing (e.g., [2, 8, 12, 14]); multi-layer user interfaces that show a user only a set of tools based on their levels of expertise (e.g [3, 22]); adaptive user interfaces that automatically manipulate the user interaces based on the user's usage history (e.g., Microsoft SMART MENUS, [17, 21, 24]); and adaptable user interfaces that give a user full control to configure her own digital workspace (e.g., [9, 16, 23]).

Of these approaches, only passive and active help systems are able to help a user discover new tools appropriate to a task-at-hand. All of the other approaches simplify the user interface, but either hide the tools from the user or do not help the user find new tools.

Passive help systems require a user to have some knowledge about the underlying HFA so that she can appropriately query for desired knowledge. Active help systems continually guess whether a user needs help and what kind of help is needed, and then, offer the help without an explicit request from the user. Although active help systems are more obtrusive than passive help systems, if an active help system can guess correctly and can offer appropriate help at the right time in the right way, it can improve the efficiency of the user at work [14]. We focus on the use of active help to aid a software developer to use appropriate tools as we are addressing a problem in software development environments where a developer is unaware of existing functionality that may help her at work. Thus, we focus the remainder of our survey on related work on these two areas.

2.1 Active Help Systems

An active help system (AHS) offers a user assistance regarding how to use functionality available in an application without being asked explicitly for the assistance by the user.

One way to categorize active help systems is by whether or not the system incorporates an explicit model of the user. Typically, a user model comprises information such as user preferences, attitudes, proficiency, interaction history, and/or user classification (e.g., novice, intermediate, expert) [15].

Examples of common active help systems that do not rely on explicit user models are error messages that are displayed when a user invokes a command provided by the application incorrectly or confirmation messages that prompt a user to confirm that she is not executing a command by mistake (e.g., exiting before saving).

An AHS that relies on an explicit user model gives advice based on such information as a user's profile and her interaction history. An example of such a system is the Microsoft Office Assistant or Clippy [12]. This type of AHS can be tricky to design since it must mimic a knowledgeable human colleague or assistant in trying to account for differences between individuals. Similar to Clippy, Spyglass is an AHS that relies on an explicit user model and operates for an environment in which users can perform hundreds of possible actions and have hundreds of possible goals. Unlike Clippy, Spyglass addresses a fraction of these actions and goals, namely those involved in navigation.

When designing an AHS, a critical choice is the technique that will be used to infer a user's goals as this information is used to determine appropriate functionality to recommend. Widely used techniques include using finite-state automata [8], Bayesian networks [12], and parsing and matching techniques [14]. Our choice to limit the goals of which Spyglass must be aware to navigation allows us to use a simple technique for goals determination. Spyglass infers that a user has a navigation goal by looking for predefined patterns of behaviours in a monitored interaction stream that are signs that a user is not navigating. Examples of such patterns of behaviours are executing run/debug/edit commands or not selecting any domain artifact for at least 30 minutes. Activity other than these patterns is assumed to be navigation.

After inferring the user's goals, an AHS needs some metric(s) for determining whether the user is performing suboptimal behaviours to achieve the goals and whether the AHS should make a recommendation. Some examples of metrics that have been used are number of keystrokes in executing a command [8] and predefined patterns that are "clues" that the user may need help [12, 14]. As a metric, Spyglass uses the number of domain artifacts (e.g., Java code elements) visited while navigating from an identified source domain artifact to another structurally related domain artifact.

2.2 Active Help in Software Development Environments

Fischer et al. integrated an active help system, called Activist, into an Emacs-like editor [7]. Activist could recognize twenty possible goals from a user's behaviours, using finite-state automata. After inferring the user's goals, such as "deletion of the left part of the current word" or "positioning to the end of the line", Activist would determine whether the user had used optimal commands or minimal keystrokes to achieve those goals. If not, Activist would decide whether to interrupt by outputing a new help message in a dialog window based on how many times the suboptimal behaviours had happened or how many times the user had ignored its suggestions for the same plans. The evaluation of Activist was limited to a self-performed case study; there was no formal evaluation to assess its benefits for users at work, nor to assess its acceptance by users.

More recent work has considered an active help system for UNIX, called RESCUER [26]. RESCUER relied on the knowledge of UNIX operators and model of files stored. It used Human Plausible Reasoning Theory to infer what commands related to file manipulation a user should have typed, given what the user did type. The work was accompanied by a comparative evaluation between hypotheses generated by the RESCUER regarding the user's beliefs and hypotheses generated by human experts on the same user sessions.

In comparison to these systems, we consider a different user goal, program navigation, introduce a different metric for determining sub-optimal behaviours, and present the results of more extensive user studies.

Chapter 3

Spyglass: The Approach

We chose to focus our active help system on aiding navigation because many developers spend substantial effort in navigating across pieces of information to understand existing code that they have to maintain and not all developers perform these understanding activities efficiently [13, 19]. While data is available for developers navigating across source code elements, similar navigation happens for other phases of development, such as software developers who triage bugs [1]. We will refer to the pieces of information whose models or relationships are stored in the development environment as *domain artifacts*. Our premise is that the effort spent by developers in navigating can be eased if developers use more appropriate tools for performing various kinds of navigation. As a simple example, developers sometimes navigate a call chain between methods by searching at each link of the call chain for the next method in the chain. A more efficient way of looking at the call chain in many development environment is available through a view (window) that shows the entire call chain from a particular point in a chain.

Our approach uses an active help system, which we refer to as Spyglass, that attempts to recommend a more appropriate tool for navigation by watching a developer's behaviour, inferring when a navigation is happening and between what domain artifacts the developer is navigating, and determining if the navigation is suboptimal in terms of visiting more artifacts than necessary. If the navigation is suboptimal, Spyglass recommends a tool that could have helped the developer navigate more optimally, for the purpose of future navigation.

Spyglass also incorporates developer feedback in making a recommendation. A developer can rate a recommended tool and the tool that has been rated poorly will not be recommended to the same developer again in the future.

3.1 Domain Artifacts

Recent software development environments provide access to many different kinds of information. Aside from knowledge about basic code elements (e.g., fields, classes,

methods), a development environment may also maintain knowledge about bug reports, work assignments, design documents, and program revisions. We use the term domain artifacts to refer to any piece of information available through the environment. Domain artifacts may be related to each other in many ways. For example, a class may be a parent of another class (inheritance), a method may call another method (call), a program revision may be associated with a work assignment (reference), a work assignment may be a parent of another work assignment (inheritance), a work assignment may be explicitly related to another work assignment (reference), or a class may have been changed, and thus, may be associated with a program revision (reference). Most software development environments provide models through which a system can programmatically access data about domain artifacts and the relationships between them.

3.2 User Behaviour Analysis Over Time

Spyglass monitors a developer's actions within the development environment over time and makes a recommendation when it detects a suboptimal navigation. The types of actions that Spyglass can recognize includes editing, domain artifacts selection, commands execution, and changes of the layout of the environment. Based on the assumption that recent events contribute more to a developer's current goals and that selection of domain artifacts may imply navigation, Spyglass only keeps track of recent events in which the developer selects domain artifacts, which we will refer to as *selection events*. More specifically, it tracks up to seven most recent selection events of domain artifacts in what is called the *window of recent activities*. Consecutive events in this window are no more than 30 minutes apart from each other. Spyglass only takes into account the events stored in this window when it makes a recommendation.

We chose seven as the maximum size of the window of recent activities based on the usage data collected from developers performing programming tasks, both in field and in the lab, in our previous studies [18]. We found that, on average, developers selected Java code elements three times with standard deviation of four before editing the source code.

The window of recent activities is updated continually. As a developer makes selections, events are added into the window of recent activities until it is full. If, at any point, the developer interacts with the environment in a way that is not recognized as navigation, the window of recent activities is restarted, meaning the window is cleared of events.

The window of recent activities needs to restart when:

- the developer performs an editing event,
- the developer starts a run or debug session, or
- the developer has not selected any domain artifact for at least 30 minutes (i.e., the latest selection event happens more than 30 minutes later than the last selection event in the window of recent activities).

At the point when the window of recent activities is full or needs to restart, Spyglass will start going through a process of making a recommendation, including inferring the source and the target artifact, selecting appropriate tools, and filtering potential recommendations. Simultaneously, Spyglass will continue updating the window of recent activities to catch up with the developer's activities. If the window of recent activities is full, it will be slided by one event to the right–the first event in the window of recent activities will be removed and the new event will be added to the end of the sequence. If the window of recent activities needs to restart, it will be wiped out, and a new selection event will be added.

3.3 Inferring Navigation Source and Target

To determine if a developer is navigating efficiently across domain artifacts in the development environment, Spyglass needs to infer the source and target of navigation. It does so by trying to find domain artifacts that are structurally related among the ones that the developer has visited within the window of recent activities. It assumes that the developer is more interested in artifacts that are related to each other than the ones that are not.

For any given kind of domain artifact, Spyglass uses facilities in the development environment to access (or recreate) a graph that represents a certain type of structural relationship (e.g., call, inheritance, reference) for that artfiact. For instance, Spyglass can access the graph that represents the call hierarchy of a given method. We define *proximity* as how far two domain artifacts x and y are from each other in terms of the number of edges along the shortest path from x to y in the structural graph of relationship that is retrieved (or recreated) from x. Thus, by definition, proximity has value greater than or equal to zero. If two artifacts are unrelated, we define their proximity to ∞. If they are the same artifacts, their proximity is equal to zero. The proximity between two domain artifacts is fixed as long as the program is not changed.

To check which domain artifacts recently visited are related, Spyglass computes the proximity between the domain artifacts associated with the first selection event and the

other events in the window of recent activities that are at least three edges apart from the first event. Figure 3.1 depicts this situation. We chose three edges based on the usage data collected from our previous studies [18] because, on average, a developer in these studies selected three Java elements before editing the source code. Moreover, the shortest possible distance someone can use to navigate from one element to another, with any tool, is one edge. We wanted to make sure that Spyglass would suggest a tool that helped shorten the navigation path by at least two edges.

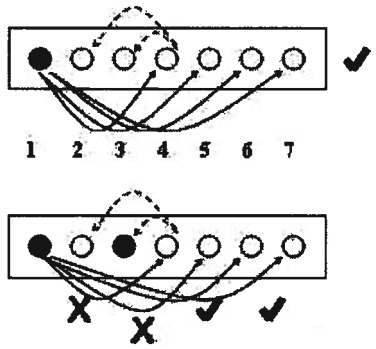


Figure 3.1: Inferring Source and Target Artifact for a Navigation

The top part of Figure 3.1 shows how Spyglass pairs up events in the window of recent activities and computes the proximity between artifacts associated with them. If any pair of artifacts are related (i.e., the proximity is not ∞), Spyglass infers them as the source and the target of the developer's navigation.

The bottom part of Figure 3.1 shows an exceptional case where Spyglass will not compute the proximity between a pair of artifacts if the same artifact as the first one is selected again within three elements before the second artifact in the pair. The two artifacts that are the same are represented by black circles in the figure. This rule is to prevent false positive recommendation in the case that the developer is switching back and forth between two artifacts.

To help understand our approach, consider a sample scenario during which the current window of recent activities looks like:

$$[Select(C), Select(H), Select(D),$$
$$Select(A), Select(B), Select(C), Select(A)]$$

where $Select(x)$ represents a selection event of an artifact x. Based on this window of recent activities, Spyglass will compute the proximity between the pairs of artifacts (C, A), (C, B), and (C, C), respectively. Spyglass will not compute the proximity between C and A when A is selected the second time because of the selection of C again before selecting A for the second time.

Figure 3.2 illustrates a structural graph–in this case, a call graph–retrieved from Method C through facilities in the development environment. In this call graph, we see how particular methods are related to each other through call relationships. The proximity between Method C and Method A is equal to two in this case, while the proximity between Method D and Method A is equal to one. Proximity between two domain artifacts is independent of the direction of their relationship.

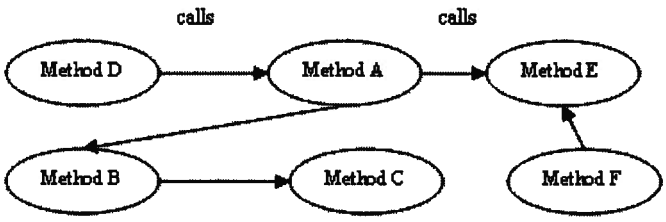


Figure 3.2: A Sample Call Graph

3.4 Selecting Tools to Recommend

After computing the proximity between two domain artifacts that the developer has visited recently, Spyglass needs to determine how many artifacts the developer has actually selected while navigating between those two artifacts within the window of recent activities. We will refer to this value as *steps actually taken*. This information is needed to decide whether the developer would benefit from using a tool. For the sample scenario we introduced, the steps actually taken to navigate between Method C to Method A for the first time is equal to three.

The Spyglass system includes a tool registry, which includes information about available navigation tools in the environment and the kind of navigation support provided by the tool.

Spyglass uses information from this registry to determine whether the developer could have shortened her navigation path by using one of the registered tools. Since the

decision needs to be made based on some information specific to each tool, Spyglass asks each tool registered with it whether it could have been useful to the developer, given the above information.

We define the *search cost* for artifact y from the output of a tool given artifact x as input to be the number of artifacts among the output returned by the tool that the developer may have to select and/or look into to get from x to y (including y), assuming the developer uses an optimal search strategy with respect to the output.

Figure 3.3 demonstrates how different tools can represent their results differently, given the same input. Supposed Method C and A are related by call relationship, as in Figure 3.2, and the developer can use either Tool 1 or Tool 2 to help find Method A given Method C. When Tool 1 and Tool 2 are invoked on Method C, they represent the output in different ways. Tool 1 shows the entire call chain, fully expanded, while Tool 2 shows only the method that directly calls Method C, requiring the developer to expand the result to see more (see the left part of Figure 3.3). A developer can expand the result returned by Tool 2 by clicking at the icon (similar to a plus sign inside a square) in front of the result. In this case, the search cost for Method A from Method C using Tool 1 and Tool 2 are equal to one and two, respectively. Tool 2 assumes that the developer would use breadth-first search as her search strategy in this case.

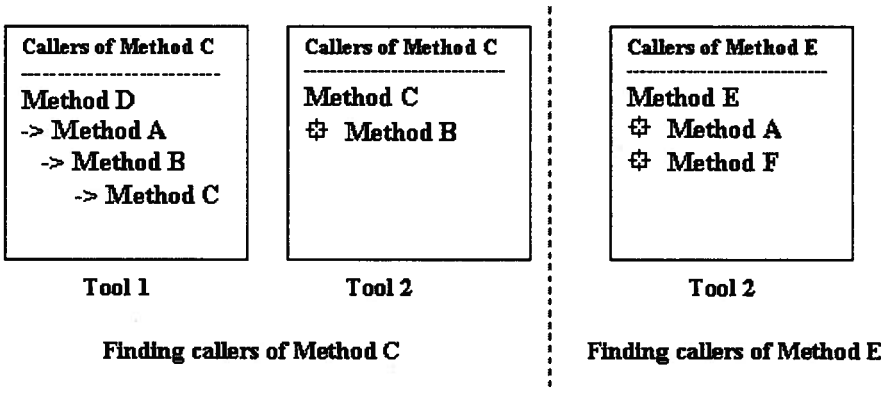


Figure 3.3: Tool Output Representation Can Differ

Search cost can vary not only across different tools, it can also vary across different source and target artifacts. For example, suppose now that there is only Tool 2 available and the developer wants to find Method D given Method E. The search cost for Method D from Method E using Tool 2 is equal to three, assuming the developer uses breadth-first search to search the output returned by Tool 2 (see the right part of Figure 3.3). Specifically, the developer will need to find the callers of Method A and Method F to

find Method D.

Therefore, every time a tool is asked whether it could have helped a developer navigate between the given domain artifacts more efficiently, it computes search cost based on the provided source and target artifacts, the type of relationship they have, and their proximity. Then, the tool determines whether it could aid the developer by considering: (1) if the search cost the developer would have to spend using the tool is smaller than the number of steps actually taken, or (2) if the number of steps actually taken is greater than the proximity between the artifacts and is smaller than or equal to the search cost, where the search cost is less than two times[5] bigger than the proximity.

In the second condition, since the number of steps actually taken is greater than the proximity betweeen the source and target artifacts, the developer could have potentially benefited from a tool. The search cost is computed based on an assumption the tool makes about the developer's search strategy used to search the output produced by the tool. In reality, the developer's intuition (e.g., name guessing) may guide them to look through fewer artifacts than estimated by the search cost. Thus, we place a bound on how much larger the search cost is than the proximity to allow for the user to use the tool more effectively than the search cost suggests.

More formally, for each tool, Spyglass evaluates the function $match(ST, P)$ (defined in Equation 3.1), where ST represents the number of steps actually taken by the developer, P represents the proximity between the domain artifacts being navigated by the developer, and SC represents the search cost computed by the tool, as defined above.

$$match(ST, P) = \begin{cases} False & \text{if } P = 0 \quad , \\ True & \text{if } \begin{array}{l} P > 0 \wedge \\ ((ST > SC) \vee \\ (ST > P \wedge \\ SC/P < 2.0)) \end{array} \quad , \\ False & \text{Otherwise} \quad . \end{cases} \tag{3.1}$$

If $match(ST, P)$ is true, then the tool is potentially useful for the developer and Spyglass will add the tool to a temporary set, which we will refer to as the *set of potential recommendations*. Spyglass then repeats the same procedure with the other pairs of artifacts left in the window of recent activities, starting from computing proximity

[5] More experimentation is needed to verify this parameter.

until adding a potentially useful tool to the set of potential recommendations.

In our scenario, Tool 1 and Tool 2 are considered potentially useful to help the developer navigate from Method C to Method A because the steps actually taken to navigate between Method C and Method A the first time is equal to three, while the search cost of both tools are equal to one and two, respectively. Thus, they will be added to the set of potential recommendations.

The tools are also considered potentially useful to help the developer navigate from Method C to Method B because the steps actually taken to navigate between Method C and Method B is equal to three, while the search cost of both tools are equal to one. Thus, Tool 1 and Tool 2 will be added to the set of potential recommendations again.

3.5 Filtering Potential Recommendations

We use a set to store potential recommendations to make sure that each tool is recommended only once per window of recent activities processed. The recommendation for a tool will refer to the pair of elements that are farthest apart in the window of recent activities that causes it. For example, based on the window of recent activities in our scenario, the set will contain only Tool 1 and Tool 2 and the recommendations will state that the tools could have helped the developer to navigate from Method C to Method B.

This set is filtered as follows before any presentation to a developer:

1. if the developer has used any tool in this set within the current window of recent activities (identified by the identifier of the place in which a domain artifact is selected), the tool will be removed from the set; or

2. if a tool has been rated by the current developer in the past as being "somewhat useless" or "very useless", it will be removed from the set.

The filtered set is final and the tools are to be added to the recommendation history, which is being observed by the UI component.

Chapter 4

Spyglass: The Implementation

Spyglass has been implemented as a plugin for IBM's Rational Team Concert (RTC) version 1.0, which is an integrated development environment built on top of Eclipse version 3.3. RTC is a development environment that is aimed to help developers collaborate. It provides some tools different from Eclipse that aid developers' collaboration. Aside from Java code elements, RTC also manages extra domain artifacts in the environment. These domain artifacts include *work items*, which are essentially work assignments for developers working in team; *changesets*, which is how RTC calls a set of program revisions; and *changes*, which are revisions of each source file stored in a changeset.

Spyglass is divided into two major parts: the core and the user interface (see Figure 4.1). The core of Spyglass performs the computations and comparisons introduced in the last chapter. The user interface is responsible for notifying users when there is a new recommendation and presenting all recommendations to users. Spyglass also has a server component that is used for storing persistent data, such as a developer's recommendation history and the past user ratings for each tool.

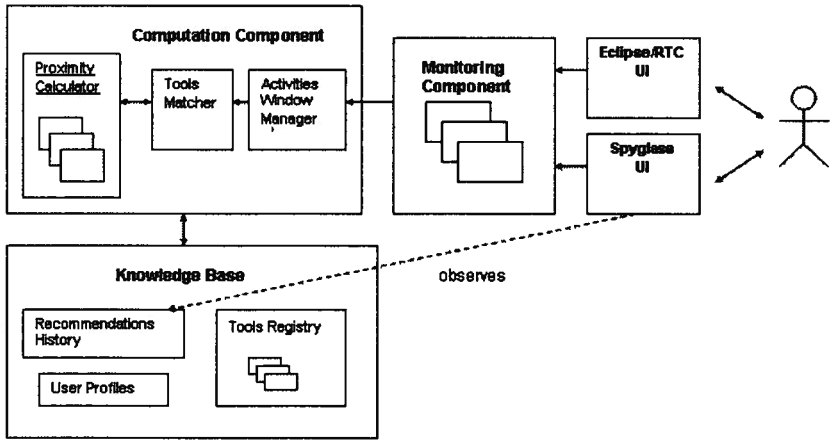


Figure 4.1: Spyglass Client Architecture

4.1 The Core

The core comprises knowledge base, monitoring component, and computation component.

4.1.1 Knowledge Base

Spyglass's knowledge base contains knowledge about user-specific information, such as a user's recommendation history and the past user ratings of each tool, and knowledge about tools themselves.

Spyglass maintains a registry of tools that it can recommend, in which it keeps track of navigation tools and their information, including their name, their keyboard shortcuts, a short description of the tool, what type of relationship for which the tool is used to navigate, and the knowledge of how it represents its output. There are three tools currently registered with Spyglass: Open Type Hierarchy, Open Call Hierarchy, and Work Item Editor Links Box.

Within Spyglass, each tool is represented by an object-oriented (Java) class. The class for each tool includes an implementation of the match operation as defined in Equation 3.1. Each class representing a tool (we refer as a *tool class* from now) must also include a method to compute search cost (as defined in Chapter 3) given a source artifact and a target artifact, the type of their relationship, and their proximity.

Open Type Hierarchy Tool The Open Type Hierarchy tool opens up a selected Java type (e.g., class or interface) in the Type Hierarchy view, showing the Java type's superclasses, subclasses, implemented interfaces, and inherited methods. Figure 4.2 demonstrates how the Open Type Hierarchy tool displays a type hierarchy of the Java class `File`. All the ancestor classes of class `File` are visible by default, whereas only direct subclasses of class `File` are visible by default. This user interface means that navigating from `File` to one of its ancestors takes only one step, whereas it can take more steps to navigate from `File` to one of its descendants.

Based on this knowledge of the user interface of the tool, the class representing the Open Type Hierarchy tool computes search cost by, first, checking whether the source and the target artifacts are Java code elements and what type of relationship they have. If they are not Java code elements or they are not related by inheritance, then the tool class returns ∞ as the search cost.

After confirming that the artifacts are Java code elements and that they are related by inheritance, the tool class accesses the inheritance graph retrieved from the source

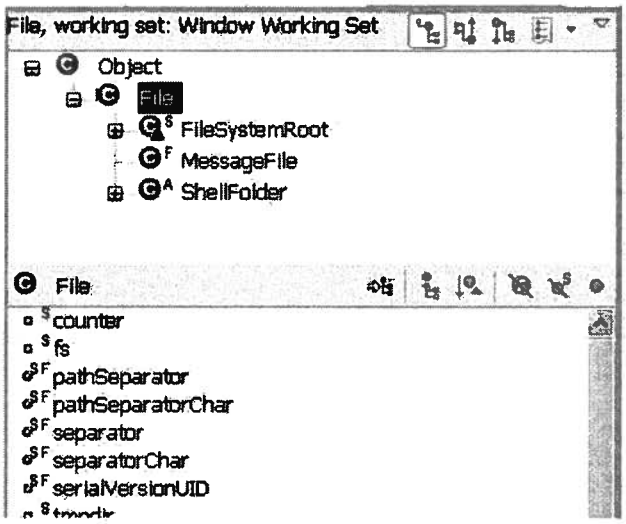


Figure 4.2: The Open Type Hierarchy Tool Output

artifact, using facilities provided in the development environment. The tool checks from the inheritance graph whether the target artifact is an ancestor or a descendant of the source artifact. If the target artifact is an ancestor, the tool class returns the search cost equal to one because a developer can navigate to the target artifact right away.

If the target artifact is a descendant of the source artifact, the tool class performs breadth-first search to search for the target artifact among all the descendants of the source and counts the number of elements visited on the way before reaching the target. This approach is used because the tool class assumes that the developer will use breadth-first search to search for the target artifact among its output.

Open Call Hierarchy Tool The Open Call Hierarchy tool displays the callers or callees of a given Java method. Figure 4.3 gives an example of how the callers of a Java method `actionPerformed` are returned by the tool and Figure 4.4 shows the callees of `actionPerformed`. Only the direct callers and callees are visible by default. The developer has to search for a caller or a callee in other levels of the call hierarchy.

Thus, to compute search cost, the class representing the Open Call Hierarchy tool first checks whether the source and the target artifacts are Java code elements and what type of relationship they have. If they are not Java code elements or they are not related by call relationship, then the tool class returns ∞ as the search cost.

Figure 4.3: The Callers Returned by the Open Call Hierarchy Tool

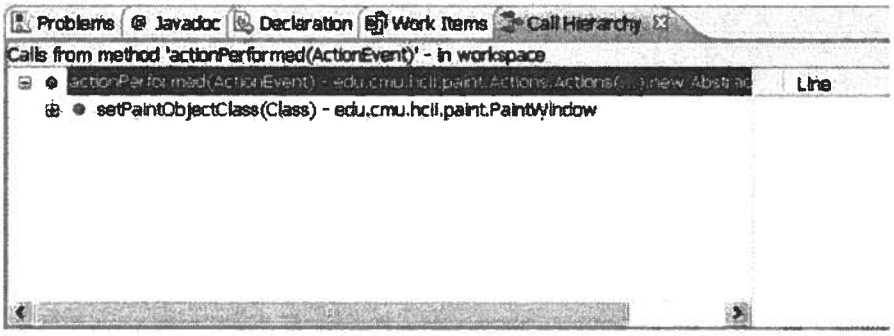


Figure 4.4: The Callees Returned by the Open Call Hierarchy Tool

After confirming that the artifacts are Java code elements and that they are related by call relationship, the tool class accesses the call graph retrieved from the source artifact, using facilities provided in the development environment. Then, the tool class performs breadth-first search to search for the target artifact among all the callers of the source and counts the number of elements visited on the way before reaching the target artifact, which is returned as the search cost in this case. As for the Open Type Hierarcy tool, the Open Call Hierarchy tool class assumes that the developer will use breadth-first search to search for the target artifact among its output.

The tool class uses the given proximity to limit the levels of the graph in which to search for the target artifact because the target artifact is guaranteed to be in the call graph no further from the source than their proximity. Thus, if the tool class cannot find the target artifact among the callers that are within this boundary, the tool class will switch to search for the target artifact among the callees, instead.

Work Item Editor Links Box A work item is essentially a work assignment for a developer. It is a domain artifact specific to RTC environment that helps a developer keep track of her and her teammates' work. The Work Item Editor is an editor in which a developer can update or obtain any information related to a particular work item. It is opened up automatically when the developer tries to open a work item.

Work items can be structurally related to one another and to other domain artifacts in the development environments. The Links box in the Work Item Editor provides access to related artifacts. Figure 4.5 shows an example in which the Links box displays that the work item of interest (Work Item 7) is related to a changeset–a group of logically related program revisions. This work item has one parent work item, one child work item, and one related work item. The Links box also shows the work items that are explicitly referenced by the creator of the work item as duplicates of Work Item 7.

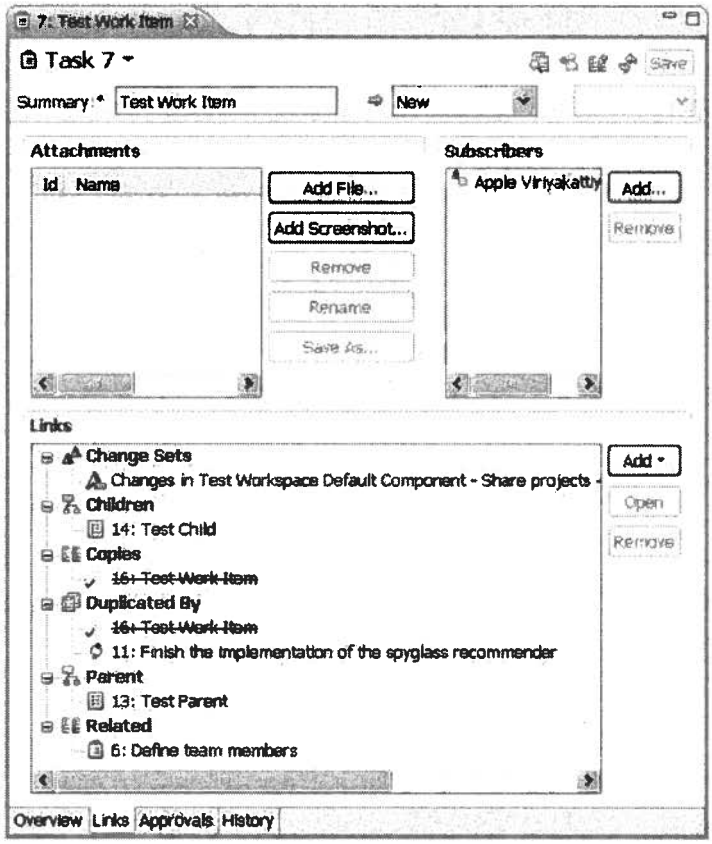


Figure 4.5: The Work Item Editor Links Box

The class representing the Work Item Editor Links Box computes search cost by first make sure that the source artifact is a work item and the target artifact is either a work item or a changeset. If the source is not a work item or the target is not a work item and not a changeset, the tool class returns ∞ as its search cost. Also, if they are not related by any type of structural relationship, the search cost is ∞.

If the source and the target are work items and they are related by inheritance, the tool class retrieves all the parents and children of the source work item using facilities provided in RTC. Then, similarly to the Open Type Hierarchy and Open Call Hierarchy tools, the tool class performs breadth-first search for the target work item among all the parents within the boundary limited by the given proximity. If it finds the target artifact, it will return the number of work items visited on the way as the search cost.

If the target artifact is not a parent, the tool class switches to perform breadth-first search among all the children of the source work item. If the source is a work item and the target artifact is a changeset or a work item that are related by any other type of reference, the search cost returned is equal to one.

4.1.2 Monitoring Component

Spyglass uses a component from the Eclipse Mylyn project–the Mylyn Monitor API (version 3.0 for Eclipse 3.3)–to monitor a developer's activities. The Mylyn Monitor records the following kinds of events representing a developer's interaction with the development environment: selection, edit, command, preference, prediction, propagation, manipulation, and attention. Selection, edit, and command events are generated from user-initiated activities, while the rest are generated from non-user-initiated activities. For each event, it records the event type, the Eclipse views or commands from which the event is originated, the type of structured element selected (if any), the ID of the structured element, timestamps, and mode of invocation for a command event. The Mylyn Monitor also creates Java objects representing these events and their information. Spyglass uses all of this information to make a recommendation, except the mode of invocation for a command event.

The Mylyn Monitor API deployed with the Eclipse Mylyn project only recognizes interaction activities with Java code elements, resources in Eclipse workspace, views and perspectives in Eclipse, and commands execution. For this thesis, we extended the Mylyn Monitor so that Spyglass can monitor the interaction activities with work items and changesets that are addtional structured elements available in RTC. Programmers working on RTC often need to explore among work items and changesets to keep track of their team work.

We also modified the Mylyn Monitor so that it passes on any event that it monitors to the Computation Component to perform any computation needed to decide what tool Spyglass should recommend (if any). The other types of events are used to determine when the developer is not navigating through domain artifacts (see Chapter 3).

4.1.3 Computation Component

This component consists of the *activities window manager* that manages the window of recent activities, monitors its state, and notifies the *tools matcher* if the window is full or needs to restart. The tools matcher is responsible for finding the right tool (if any) by asking an appropriate kind of *proximity calculator* to compute proximity between pairs of artifacts in the window of recent activities, asking each tool whether it could have helped the developer navigate between the artifacts more efficiently, and filtering the potentially useful tools before sending it to be stored in recommendation history, which is in turn observed by the Spyglass UI component (see Figure 4.1).

More specifically, a proximity calculator is given two selection events of a source and a target domain artifact and the steps the developer has actually taken to navigate between the two artifacts. The calculator returns the type of relationship the two domain artifacts have (if any) and the proximity with respect to the type of relationship.

Spyglass currently supports three types of proximity calculators: one that is responsible for calculating proximity between two Java elements, one that is responsible for calculating proximity between work items or between work items and changesets, and another one that is responsible for calculating proximity between Java elements and changes.

Source Artifacts Proximity Calculator The source artifacts proximity calculator is responsible for computing proximity between Java code elements. This calculator ensures that the given source and target artifacts are valid Java elements and retrieves objects corresponding to the given element identifiers using facilities in the development environment.

The source artifacts proximity calculator currently supports two types of relationship between Java code elements: inheritance and call. If the source and the target artifacts are the same artifact, the proximity is set to zero and is returned immediately. Otherwise, the calculator retrieves an inheritance graph and a call graph from the source artifact and uses breadth-first search to find the target artifact from the graph. To prevent an infinite search time, it uses the number of steps actually taken to limit how many edges in the graph away from the source artifact to search in each direction

(e.g., to search among the callers vs. callees).

If the target artifact is found in any of the graphs, the proximity with respect to the type of the graph is equal to the shortest path between the source and the target artifacts in that graph. The type(s) of relationship between the artifacts is(are) determined by the type(s) of graph in which the target artifact is found. If the target artifact cannot be found in any of the graphs within the boundary limited by the given number of steps actually taken, the proximity returned is equal to ∞ because Spyglass should not recommend any tool to the developer if the steps actually taken is already smaller than the proximity between the source and the target artifacts.

Work-Item-Related Proximity Calculator The work-item related proximity calculator computes proximity between two work items or between a work item and a changeset. This proximity calculator knows that work items can be related either by inheritance or by reference, while a work item and a changeset can be related only by reference. Thus, if the source and the target are both work items, the calculator, using facilities provided by the development environment, retrieves an inheritance graph from the source and all the other work items related to the source. Then, the calculator looks for the target work item in the inheritance graph using the same strategy as the source artifacts proximity calculator. If found, the proximity between the source and the target artifacts, with respect to inheritance relationship, is set to the shortest path between the two artifacts in the inheritance graph.

The calculator also looks for the target work item among all the other work items related to the source. If found, the proximity according to a reference relationship between the two artifacts is set to one. On the other hand, if the target is not found in any of the related work items, the proximity is set to ∞.

If the target artifact is a changeset, the calculator retrieves all related changesets using facilities provided by RTC and tries to find the target changesets among them. If found, the proximity between the source and the target artifacts, with respect to the reference relationship, is set to one; otherwise, it is set to ∞.

4.2 The User Interface

Spyglass notifies a developer when there is a new recommendation for tools that might help the developer navigate more efficiently. This UI approach is reminiscent of Microsoft's Office Assistant or Clippy, which is seen as a failed active help system. At least one post-mortem assessment of Clippy suggests Clippy's approach failed because

it was impolite: It preemptively took control of a user's cursor, did not remember a user's past choice, such as to not receive recommendations about some functionality, and did not offer a user an option to permanently disable it in the obvious way [27].

Keeping this history in mind, we carefully design the user interface of Spyglass to avoid these pitfalls. We want to alert a user to the availability of a new recommendation in a noticeable but unobtrusive way. To achieve this goal, we place an icon for Spyglass (a wrench) on the trim of RTC (in the bottom right corner of the main window by default) so that it is visible to the user at all time.

When there is a new recommendation for the user, Spyglass pops up a small window in the bottom right corner of RTC, without taking over the cursor focus from the user's current task. The popup is accompanied by a sound similar to Windows' System Notification Sound. The Spyglass icon also changes from a wrench to a light bulb.

The information in the popup window includes the name of the recommended tool, how to access it, and what type of relationship between domain artifacts it shows (see Figure 4.6). The user can close this popup at any time, or it will close itself after fifteen seconds.

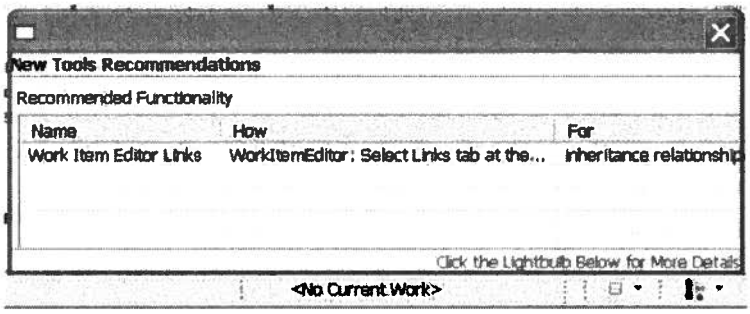


Figure 4.6: Spyglass Popup

Spyglass makes sure that each popup notification is at least one minute apart from each other. So, sometimes, even though there is a new recommendation, there will not be a popup if the last one just happened less than one minute ago. The user can disable this popup notification by using the drop-down menu next to Spyglass icon.

If the user clicks on the popup view or on the wrench/light bulb icon, an in-place view, called the Recommendation View, opens on top of the popup to give the user more details about the recommendation. The Recommendation View also shows all of the recommendations that the user has received. This approach clearly identifies the recommendations as coming from Spyglass and does not create another view that a user must manage in the development environment. One click is all that is required

to dismiss the in-place view. Figure 4.7 shows an example of a recommendation, in this case for more efficient navigation of work items, in the in-place view. Whenever the Recommendation View is opened, a second connected in-place view, called the Rationale View, is expanded to the left, which explains why the recommendation has been given, what type of relationship the tool can show, and how else the user can invoke the tool.

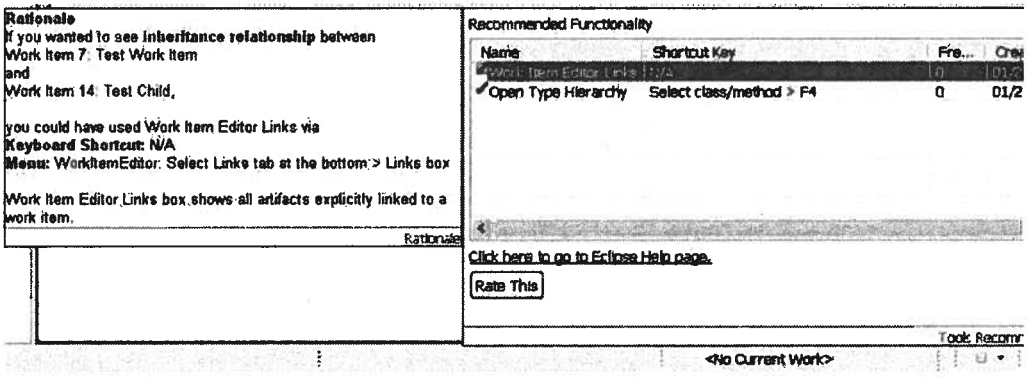


Figure 4.7: Spyglass User Interface

Spyglass also includes a capability for rating a recommended tool. The user can rate each recommended tool (not each recommendation) on a five-point Likert scale, from very useless to very useful. To do that, the user has to select a recommended tool in the Recommendation View, and click "Rate This" button on the view. Then, there will be another popup dialog, in which the user can give the rating. This rating will be used by Spyglass when making a recommendation to the same user in the future.

Chapter 5

Longitudinal User Study

Chin and Krause describe that formal evaluations of active help systems are needed to determine whether and under what conditions these systems help or hinder their users [4, 14]. These evaluations can also help assess whether the benefits of an active help system support the typically high development costs of these systems.

To evaluate the value and acceptance of the first prototype of Spyglass (called Spyglass 0.9), we conducted a longitudinal user study with students in an introductory, third-year, software engineering course at UBC. The study aimed to answer four questions.

L-1 Is Spyglass effective in helping a developer discover useful tools that are unknown to the developer?

L-2 Is Spyglass effective in encouraging a developer to use a previously unknown tool?

L-3 Does Spyglass help a developer explore source code more efficiently?

L-4 Is Spyglass's notification mechanism obtrusive?

We chose to conduct a longitudinal study because we wanted to observe the true effects of Spyglass (that might be biased in a laboratory study) on developers who were novice to a development environment over a long period of time. Since Spyglass could only recommend limited tools at the moment, we also wanted to see how long a developer would benefit from Spyglass before she became an expert and did not need its help.

5.1 Spyglass 0.9

The first prototype of Spyglass was designed to be minimally obtrusive based on published suggestions of how to design a polite user interface [27]. Spyglass 0.9 differed from the current version of Spyglass (Version 1.0 described in Section 4), primarily in

the notification mechanism. When there was a new recommendation, Spyglass 0.9 did not use a popup notification. Instead, Spyglass 0.9 changed the icon on the development environment's window trim from a wrench to a light bulb and used a noticeable but brief notification sound (similar to Windows New Mail Notification).

Spyglass 0.9 also differed in its presentation of recommendations in the Recommendation View. In Spyglass 0.9, if a developer has already received a recommendation about a certain tool that Spyglass is now recommending again in a different context, Spyglass would replace the existing recommendation in the Recommendation View with the new one. The result was that the developer would only see recommendations for unique tools in the view. Each recommendation remained in the Recommendation View for only one day.

The Rationale View of Spyglass 0.9 did not show up automatically as a developer opened the Recommendation View. Rather, it appeared only when the developer selected a recommendation in the Recommendation View. Also, the content of the Rationale View was different, describing the number of selection events the developer could have saved if she used the recommended tool to navigate between two domain artifacts that she has previously visited. Figure 5.1 shows the Recommendation View of Spyglass 0.9 with the Rationale View expanded on the left because of a selection of a recommendation. The content of the Rationale View explains that the Work Item Editor Links Box is being recommended to the developer because she is navigating suboptimally between Work Item 7 to Work Item 14 that are related by inheritance. The rationale explains that the proximity between these two work items is equal to one, and search cost is equal to one, while the developer has actually visited four artifacts in the navigation. Description about the recommended tool and other means to invoke it are also included in the Rationale View.

Spyglass 0.9 was less precise in providing recommendations than Spyglass 1.0. As soon as there was a new recommendation, Spyglass 0.9 played a notification sound, even though the recommendation might appear the same to a developer or the previous recommendation had just occured. Also, it did not consider the case that the developer was switching back and forth between two domain artifacts.

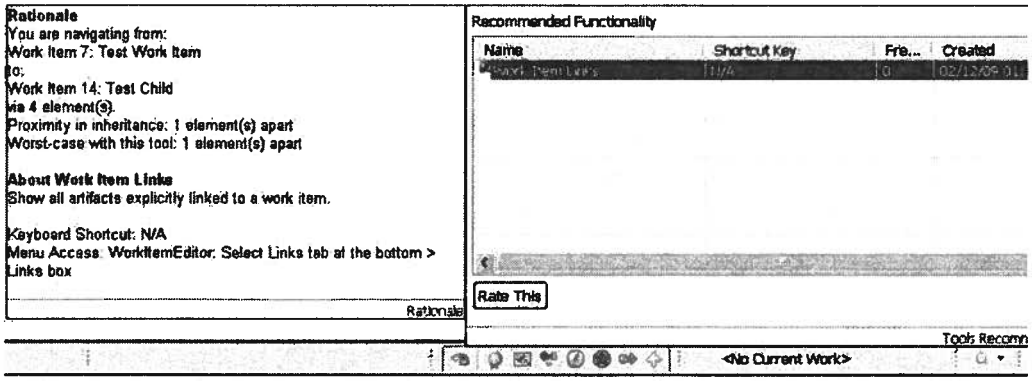


Figure 5.1: Spyglass 0.9 User Interface

5.2 Participants

We recruited participants from a third-year software engineering course at UBC in the Department of Computer Science, in which students were all required to use IBM's RTC to work on their course project. These students had no experience working in RTC environment, but did have one or two courses of experience with Eclipse development environment, on which RTC is based. To ensure anonymity of the participants, we refer to a participant as "she" or with a codename where necessary.

After a participant signed an online consent form to participate in the study, she was immediately asked to fill out a preliminary questionnaire online to assess her demographic information and technical background. Ten students signed-up for the study; one student dropped out in the middle. Nine participants were male; one was female. All participants had some experience programming in Java; however, all had less than five years of experience (see Table 5.1).

Table 5.1: Programming Experience of Participants in Longitudinal Study

Question	Number of Responses			
	None	< 1 year	1-5 years	> 6 years
How many years of experience do you have in programming in Java?	0	3	7	0

Half of the participants (five out of ten) reported experience in developing software outside of University classes; three of the five also reported working with complex libraries or code that other people wrote before. There were four participants, in total,

who have worked with complex libraries or code that other people wrote (see Table 5.2).

Table 5.2: Software Development Experience of Participants in Longitudinal Study

Question	Number of Responses	
	No	Yes
Do you have experience developing software outside of courses?	5	5
Do you have experience developing software that calls complex libraries or working on code that someone else wrote?	6	4

Eight out of ten participants reported regular use of a development environment; five out of eight said that their favorite development environment was Eclipse. None of the participants used RTC regularly. Six participants considered themselves to be intermediate or somewhat expert in Eclipse. Nine out of ten students considered themselves as novices in RTC; the tenth participant considered herself to be intermediate (see Table 5.3 and 5.4).

Table 5.3: Usage of Development Environment of Participants in Longitudinal Study

Question	Number of Responses		
	No	Yes: Eclipse or RTC	Yes: Others
Do you use an IDE regularly? If so, which ones?	2	5	3

Table 5.4: Expertise of Participants in Longitudinal Study

Question	Number of Responses				
	1 (Novice)	2	3	4	5 (Expert)
Please rate your expertise in Java	0	4	4	2	0
How familiar are you with concepts of O-O programming?	0	0	6	4	0
Please rate your expertise in Eclipse	0	4	5	1	0
Please rate your expertise in RTC	9	1	0	0	0

5.3 Experimental Design

We wanted to see whether Spyglass would help improve the participants' navigation style, and whether Spyglass would be effective in introducing and encouraging the use of tools new to the participants. Previous work in software engineering has shown that developers differed vastly in terms of working style, experience, intuitions, and intelligence (e.g., [10, 19, 20]). To account for this variances, we designed the study to be a repeated-measures experiment, in which we would measure the actions of an individual participant during a period of time, in which Spyglass was not available for her, and compare the resulting measures to our observation during the time Spyglass was made available for her.

The course in which we conducted the study was an introductory software engineering course, in which students were expected to attend lectures twice a week and their assigned lab session once a week. In each lab session, students would work on their course assignments, learn how to use RTC, and work on their course project assignments; all with the guidance of their teaching assistants. 10% of the course grades were based on the students' participation in the class activities and lab attendance; 30% were based on the course project and the project-related assignments; and 60% were based on their midterm and final exam.

We loosely controlled the study by the course schedule each participant had to follow, starting from the lab session in which the participant was introduced to RTC, until the end of her course project. This variable was only loosely controlled because there was no guarantee that all participants would follow the schedule.

5.4 Experimental Procedure

After filling out the consent form, participants were asked to install RTC and Spyglass on every computer on which they would work on their course project, such as the lab computer they used and their laptops. Each time a participant installed Spyglass, she was required to configure Spyglass with her username to personalize the recommendations made by Spyglass. If the participant forgot to perform this configuration step, she was reminded with an popup error message the next time she used RTC. Each participant was also given a link to the Spyglass tutorial online that described the meanings of Spyglass icons, how to disable or enable Spyglass, how to rate a recommended tool, and how to submit data for the study using our menus provided in RTC.

The study was divided into three phases based on the course schedule: an initial code-exploration exercise phase (one week), a second code-exploration exercise phase (one week), and an implementation phase (around two weeks). For each code-exploration exercise phase, the students were given a set of questions to answer about the Eclipse API and other libraries on which their course projects would be based. These exercises were designed to require the students to explore the source code of the Eclipse APIs and those libraries in order to answer the questions correctly. After each of the code-exploration phases, students were supposed to implement part of the system based on the knowledge they gained from the exercise and show it to their teaching assistants. The implementation phase was the period after the second code-exploration exercise phase, until the end of the course project.

During the first code-exploration phase, Spyglass was disabled for the students, meaning that participants could not interact with Spyglass or receive any recommendation. After the date and time that a participant was supposed to submit the first code-exploration exercise, Spyglass was enabled automatically. For the study, this first code exploration phase enabled all students to gain familiarity with exploring code. For the remainder of the study, Spyglass was enabled for the participants. The second code exploration phase was intended to ensure all students continued to navigate code but in an environment in which recommendations might occur. We left Spyglass enabled for the implementation phase to allow additional opportunities for the students to receive recommendations.

5.4.1 Collecting Data

We collected data about a participant's actions through two types of log files: the Mylyn Monitor log and the Spyglass log. The Mylyn Monitor log records all of a participant's

interaction events with RTC and their information that are captured by the Mylyn Monitor (see Section 4.1.2). We implemented the Spyglass logger to record the state of the window of recent activities just before Spyglass attempts to make a recommendation, the tools recommended (if any), and all of the events in which a developer interacts with the Spyglass developer interface (e.g., clicking on Spyglass icon, selecting a recommendation in the Recommendation View, and disabling/enabling Spyglass). Figure 5.2 shows an example of a portion of each log file.

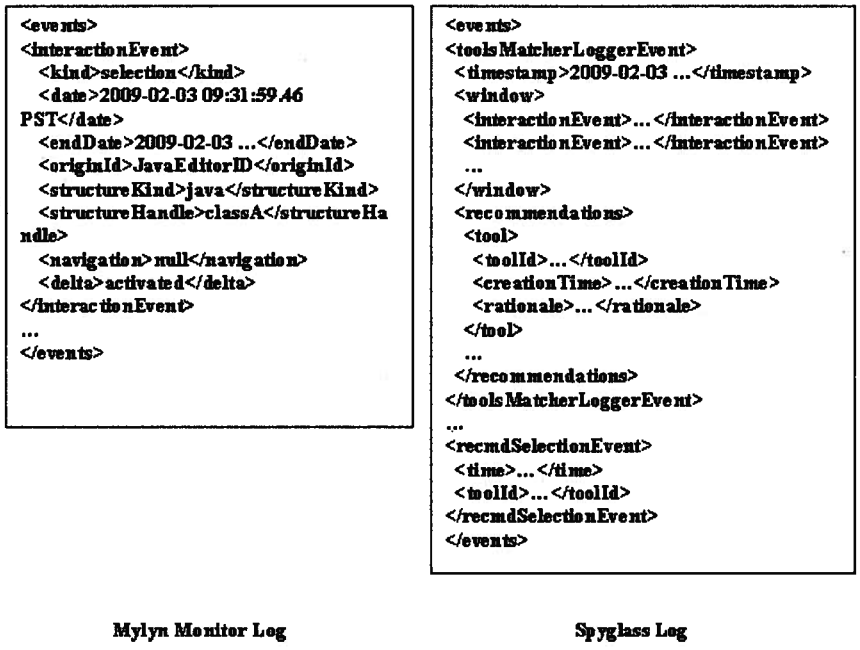


Figure 5.2: Log Files Format

The portion from the Mylyn Monitor log shows a selection event, in which a developer selects a Java code element, named ClassA, in a Java editor identified by the development environment with the ID "JavaEditorID". The portion of the Spyglass log indicates an event in which one or more recommendations are given. A list of recommendations is enclosed by the `<recommendations>` tag. The state of the window of recent activities right before Spyglass makes the recommendations is enclosed in the `<window>` tag. The `<recmdSelectionEvent>` tag also indicates that the developer has selected a given recommendation.

We asked the participants to submit these log files from within RTC using a pro-

vided menu after they submitted each code-exploration exercise, from every machine that they used to work on the exercise. We also gathered data in the form of a recurring questionnaire that was provided in RTC, which was supposed to be submitted to us at the same time as the log files. This questionnaire asked two questions intended to assess a participant's confidence in the ability to navigate source code efficiently and to discover new and useful tools in the development environment (see Appendix B.7).

After each participant had finished her project, she was asked to come to our laboratory and complete a post-experiment questionnaire (see Appendix B.3) on the experimenter's computer. Then, the participant would participate in a 30-minute semi-structured interview with the experimenter (see Appendix B.8 for the interview protocol). A participant could also choose to have an interview over Windows Live Messenger, instead of coming in to the lab, and give the answers to the post-experiment questionnaire through email.

5.5 Results

We present the quantitative and qualitative results from the longitudinal study.

5.6 Quantitative Results

Originally, we planned to compare the values of data collected during the first code-exploration exercise phase and the second code-exploration exercise to answer the first three questions of interest to the study, specifically L-1 to L-3. This data includes the answers from the recurring questionnaires, the data about target tools usage (i.e., the tools that Spyglass could recommend: Open Type Hierarchy, Open Call Hierarchy, and Work Item Editor Links Box) from the Mylyn Monitor log, and the data about recommendations and the participants' interaction with Spyglass from the Spyglass log.

A number of factors complicated our ability to analyze this data effectively, and made us unable to conclusively answer questions L-1 to L-3 as planned. For example, we discovered from the log data and the interviews that five out of ten participants did not notice where the Spyglass icon was on the main RTC window, until after the second code-exploration phase. Four out of nine participants who completed the study also never noticed the change of icon as a notification of new recommendations, and thus, did not notice many recommendations from Spyglass. Two of the participants did not find out where Spyglass was at all until the end of the study.

There were also other variations that made the quantitative data difficult to analyze. For example, some participants did not use an RTC installation that had Spyglass installed to complete their code exploration exercises. This lack of use was evident through analysis of the logs. When asked in the interviews, participants stated that they looked at the source code online. We did find useful information from analyses of log data and answers to the post-experiment questionnaire. Even though this information did not give us conclusive answers to questions L-1 to L-3, it gave us an answer to question L-4 and some insights about how to improve the Spyglass system. We report on these analyses in the next sections.

5.6.1 Log File Analysis

By analyzing the log data, we were able to learn how participants used Spyglass and the target tools. From a participant's Mylyn Monitor log files and the Spyglass log files, we extracted statistics about how often Spyglass recommended, how often each participant saw the recommendations, how often a participant read the rationale, how often a participant disabled Spyglass, and how often a participant used the target tools.

Spyglass was enabled for the participants automatically after the deadline of the first code-exploration exercise posted on the course website. We assumed that the time that a participant had to submit the exercise was the beginning time of the lab session in which the exercise was due. However, some participants submitted the log files after the Spyglass was enabled. We did not know for sure when those participants submitted their first code-exploration exercises and when they submitted their log files. They might or might not have submitted their log files immediately after submitting their code-exploration exercises. In other words, we did not know whether they received the recommendations before they submitted the first code-exploration exercise or not.

Table 5.5 describes how participants interacted with Spyglass. In particular, the second column describes the number of times Spyglass gave a new recommendation to the participant. The third column in the table describes the total number of recommendations seen by the participant, which is equal to the sum of the number of unexpired (less than one day-old) recommendations appearing in the Recommendation View whenever the participant opened it up. The number of times the participant opened the Recommendation View is shown in the fourth column. The fifth column describes the number of time the Rationale View was opened by the participant (by selecting a recommendation in the Recommendation View) and the last column shows the number of times the participant disabled Spyglass using the provided menu.

Table 5.5: Interactions with Spyglass During the First Code-Exploration Phase

Participant	No. of Unique Recommen-dations	No. of Non-Unique Recommen-dations Seen	No. of Times Recommen-dation View Opened	No. of Times Rationale View Opened	Disable Command Executed
s1	12	8	5	2	1
s3	30	2	1	0	0
s4	7	0	0	0	0
s5	0	0	0	0	0
s6	7	0	0	0	0
s7	4	0	0	0	0
s8	24	23	11	3	0
s10	11	4	3	2	0
s11	0	0	0	0	0
s12	0	0	0	0	0

More specifically, Table 5.5 shows that four out of ten participants (s1, s3, s8, and s10) saw Spyglass recommendations in the Recommendation View during the first code-exploration phase. However, a closer look at their Mylyn Monitor log files and Spyglass log files told us that only two of those participants (s8 and s10) started using the target tools after receiving recommendations about them.

Table 5.6 shows how often each participant interacted with the target tools before submitting the first set of her log data; before and after Spyglass was enabled. We use *OTH* to represent the Open Type Hierarchy tool and *OCH* to represent the Open Call Hierarchy tool.

The second set of log files was submitted after Spyglass was enabled for every participant. However, some participants were late in submitting the log files and we did not know for sure which participants submitted the log files immediately after submitting the exercise. Seven out of ten participants were supposed to receive recommendations from Spyglass; only three of them clicked on the Spyglass icons and two out of the three saw the recommendations (see Table 5.7) along with the rationale. Of those two, one has not seen any recommendation during the first code-exploration phase (participant s4). However, the participant disabled Spyglass twice. A closer look at her log files told us that the participant has been using the target tools before seeing any recommendation (see Table 5.6 and Table 5.8).

Table 5.6: Interactions with Target Tools During the First Code-Exploration Phase

Participant	Spyglass Disabled			Spyglass Enabled		
	OTH	OCH	WorkItem Editor Links	OTH	OCH	WorkItem Editor Links
s1	0	0	0	0	0	0
s3	2	2	0	0	0	0
s4	0	0	12	2	0	4
s5	0	0	0	0	0	0
s6	0	0	0	1	0	0
s7	5	0	0	3	0	0
s8	0	0	41	0	3	0
s10	0	0	0	6	0	0
s11	0	0	0	0	0	0
s12	0	0	4	0	0	0

Table 5.7: Interactions with Spyglass During the Second Code-Exploration Phase

Participant	No. of Unique Recommen-dations	No. of Non-Unique Recommen-dations Seen	No. of Times Recommen-dation View Opened	No. of Times Rationale View Opened	Disable Command Executed
s1	54	7	4	3	0
s3	18	0	0	0	0
s4	23	18	13	6	2
s5	2	0	0	0	0
s6	6	0	0	0	0
s7	0	0	0	0	0
s8	0	0	0	0	0
s10	1	0	2	0	0
s11	14	0	0	0	0
s12	0	0	0	0	0

Table 5.8: Interactions with Target Tools During the Second Code-Exploration Phase

Participant	OTH	OCH	WorkItem Editor Links
s1	0	8	0
s3	0	0	0
s4	34	2	29
s5	0	0	1
s6	18	0	0
s7	0	0	0
s8	0	0	4
s10	0	0	4
s11	0	0	0
s12	0	0	0

The implementation phase lasted around one week longer than each of the first and the second code-exploration phase. Although a longer period of time should have allowed for several recommendations, but two of them never opened up the Recommendation View at all, implying no recommendations were seen (see Table 5.9).

Table 5.9: Interactions with Spyglass During the Implementation Phase

Participant	No. of Unique Recommendations	No. of Non-Unique Recommendations Seen	No. of Times Recommendation View Opened	No. of Times Rationale View Opened	Disable Command Executed
s1	4	0	0	0	8
s3	6	3	6	3	0
s4	79	53	31	20	11
s5	24	0	0	0	0
s6	25	5	5	4	0
s7	9	4	20	1	0
s10	20	6	7	0	0
s11	41	6	3	10	0
s12	40	3	5	1	0

We also noticed that every participant used target tools more often than during the first and the second code-exploration phases overall (see Table 5.10). However, an analysis of the relationship between the times participants saw recommendations and when they used the target tools suggested that six out of nine participants started using those tools even before seeing the recommendations.

Table 5.10: Interactions with Target Tools During the Implementation Phase

Participant	OTH	OCH	WorkItem Editor Links
s1	0	0	0
s3	128	4	0
s4	58	1	8
s5	0	0	6
s7	49	2	14
s8	25	3	3
s10	0	0	1
s11	4	0	1
s12	0	2	4

5.6.2 Post-Experiment Questionnaire Analysis

One question in the post-experiment questionnaire asked participants to rate usefulness of recommendations in the context of their tasks. Table 5.11 shows the answers of all participants who remained until the end of the study. Of the participants who saw the recommendations, two of them rated the recommendations as useful, four of them were neutral, and one said that they were somewhat useless. Thus, on average, participants felt neutral (mean=2.89, stdv.=1.453) towards Spyglass's recommendations.

Table 5.11: Ratings of Recommendations and Rationales by Participants (from 1=very useless to 5=very useful)

Participant	How often did you look at recommendations?	Usefulness of Recommendations	Did you read the rationale?	Usefulness of Rationales
s1	25%	2	No	-
s3	25%	3	No	-
s4	75%	4	Yes	3
s5	Never	-	-	-
s7	100%	3	No	-
s8	100%	3	No	1
s10	100%	3	Yes	2
s11	75%	4	No	-
s12	Never	-	-	-

Table 5.12 presents how useful the participants felt about the tools recommended by

Spyglass. Each participant who saw a recommendation about the Open Type Hierarchy tool did end up using the tool; on average, these participants rated the usefulness of the Open Type Hierarchy tool as neutral. Five participants also appear to have used the Open Call Hierarchy tool because of a Spyglass recommendation; these participants rated the Open Call Hierarchy tool as very useful on average.

Table 5.12: Ratings of Usefulness of Tools by Participants (from 1=very useless to 5=very useful)

Participant	Usefulness of OTH	Usefulness of OCH
s1	2	5
s3	2	5
s4	5	5
s5	-	-
s7	3	4
s8	4	5
s10	4	-
s11	4	-
s12	-	-

The post-experiment questionnaire also asked questions that assess user-friendliness of Spyglass. Table 5.13 shows the answers to those questions. Six out of seven participants who saw a recommendation agreed that recommendations were presented in a way that was easy to understand (mean=4, stdv.=0.5774). Of three participants who heard the sound notification from Spyglass at all, two of them did not think that the sound notification was disturbing, while the third said it was very disturbing (mean=2.67, stdv.=2.0817).

Table 5.13: Ratings Regarding User-Friendliness of Spyglass by Participants (from 1=strongly disagree to 5=strongly agree)

Participant	Recommendation View was easy to understand?	The sound notification was disturbing?
s1	4	-
s3	4	-
s4	3	5
s5	-	-
s7	5	1
s8	4	2
s10	4	-
s11	4	-
s12	-	-

5.7 Qualitative Results

Based on the usage data, when a new recommendation was available, none of the participants *always* opened up the Recommendation view to look at the new recommendations. Two of the nine participants that completed the study never looked at any recommendation. In the interview, these two participants said that they were unaware of the location of the Spyglass icon, and thus, did not see the icon change when there was a new recommendation. These two participants also had their speakers off while working. The collected usage data tells a slightly different story: one of these two participants clicked the Spyglass icon five times and must have seen some recommendations. However, the participant did not realize that these recommendations were from Spyglass. These participants suggested that when a new recommendation was available, a small window should slide up in a bottom corner of the screen and stay there for awhile, similar to a toaster notification. We asked other participants whether they would mind this popup notification; most of them said that they would not mind as long as the window closed itself after a reasonable amount of time. Some suggested that Spyglass should pop up previous recommendations only once when RTC was launched, instead of having a small window slide up every once in a while.

Five out of seven participants who noticed recommendations recognized the availability of recommendations via the icon change (see Table 5.14). The other two who did not notice the icon change said that they found the recommendations when they randomly clicked the Spyglass icon. From the interview, we found that some participants also opened the Recommendation view when they were bored or frustrated about

their current tasks and did not know how to proceed. They thought they might find some useful tools that could help them or they simply wanted to read about a new tool as a break from their work. We also determined that participants did not necessarily look at recommendations right after notification if they were busy doing something else.

Table 5.14: User Perceptions of Spyglass 0.9

Participant	Did you notice the Spyglass icon changed?	Did you notice Rating button?
s1	Yes	No
s3	No	No
s4	Yes	Yes
s5	No	No
s7	Yes	No
s8	Yes	No
s10	Yes	No
s11	No	No
s12	No	No

Of the participants that looked at recommendations, only two participants remembered looking at the associated rationale for a recommendation (see Table 5.11). However, the usage data suggests that, actually, six participants had brought up the rationale view (see Table 5.9). The interview revealed that some participants just tried to guess why a recommendation appeared. Some participants who were aware of the recommended tools thought that Spyglass was suggesting they use shortcut keys rather than menu access to the tool.

We also asked each participant their thoughts whether this kind of recommendation system should be available in a development environment like RTC or Eclipse. All participants agreed that it was a good idea and it would likely help them work more efficiently. One participant also requested a recommender that evolved with his knowledge. For example, once she has learned an initial set of recommended tools, the recommender should begin recommending more advanced tools.

5.8 Summary

In trying to be a polite recommendation system, Spyglass may have become overly polite such that participants did not notice when recommendations were available. The

challenge becomes trying to design a user interface mechanism that is sufficiently intrusive to be noticeable when a developer pauses in their work, but to not remove a developer's focus from their current task. The result of this study lead to the current version of Spyglass (called Spyglass 1.0), which uses a small sliding popup window that reminds the developer about new tools, as suggested by the participants in the study. Spyglass 1.0 also retains a history of recommendations so that developers can go back to refer to prior recommendations and to peruse the recommendations on their own timetable.

The study also highlighted the difficulties of presenting recommendations in a way that a developer does not mistake the purpose of the recommendation. We tried to provide the purpose through the rationale in-place view. However, many participants did not read the rationale, instead they believed Spyglass was urging them to use keyboard shortcuts. Thus, Spyglass 1.0 forces open the Rationale View whenever a developer opens up the Recommendation View to see any recommendation.

Chapter 6

Laboratory User Study

To study more specific aspects of our active help approach for the development environment than were possible with the longitudinal study, we conducted a controlled laboratory study. With this study, we sought to explore the following questions:

Q-1 Can Spyglass help developers complete tasks more successfully?

Q-2 Can Spyglass help navigate through pieces of information more efficiently?

Q-3 Can Spyglass raise a developer's awareness and use of tools in a development environment?

Q-4 Are the tools recommended by Spyglass perceived as useful by developers?

Q-5 Do developers perceive Spyglass as suggesting the right tools at the right time?

Q-6 How often does Spyglass suggest the right tool at the right time?

The usefulness of an active help system also depends upon the user's awareness of a new recommendation. Yet, using a popup notification to notify a developer of a new recommendation may make the system become less user-friendly. Thus, we also sought an answer to the following question:

Q-7 Does Spyglass's popup notification mechanism overly disturb a developer's workflow?

With the laboratory study, we wanted to compare how developers who learn about tools passively through tutorials and developers who learn about tools actively through Spyglass perform on software change tasks. The study involved two target tools for the developers to learn and use: the Open Call Hierarchy tool and the Open Type Hierarchy Tool. The Open Call Hierarchy tool displays the callers or callees of a given Java method. The Open Type Hierarchy Tool shows all superclasses, subclasses, implemented interfaces, and all inherited methods of a given Java type (i.e., class and interface).

6.1 Participants

For the study, we needed participants with some experience in Java programming and some experience with the target development environment to allow the use of moderately complex software change tasks that would require the use of tools. Thus, we sought participants who have programmed in Java for at least eight months (or finished at least two programming courses) and had used Eclipse for at least two months, but had not known about all the target tools that Spyglass would recommend to them. In addition, we wanted participants to be newcomers to the software system on which they would be asked to perform software change tasks, called JFreeChart[6].

An individual indicated interest in the study by visiting our website and filling out an online consent form. After giving consent, the individual was presented with a questionnaire to gather data about her experience with programming in Java and with Eclipse, including what tools in Eclipse she had used or known before. If an individual indicated knowledge about both of the target tools, we thanked the individual for her interest but removed her from the participants list.

We recruited a total of eighteen qualified participants: four females and fourteen males. Table 6.1 details the programming experience of the participants and Table 6.2 details the participants' experience with Eclipse development environment. We use *OTH* to represent the Open Type Hierarchy tool and *OCH* to represent the Open Call Hierarchy tool. Ten of the participants were undergraduate students in the Computer Science Department at UBC; eight were graduate students: six of them were affiliated with the Computer Science Department or the Electrical and Computer Engineering Department, and the rest were affiliated with other departments at UBC.

Table 6.1: Programming Experience of Participants in Lab Study

Group	Java Experience (months)			Biggest Java Program (classes)			Has Worked with Others' Code?	
	8-11	12-15	>15	<50	50-75	>75	Yes	No
All	9	3	6	13	2	3	13	5
Spyglass	5	1	3	8	1	0	6	3
Tutorial	4	2	3	5	1	3	7	2

[6] `http://www.jfree.org/jfreechart/`, verified April 16, 2009

Table 6.2: Experience with Eclipse Environment of Participants in Lab Study

Group	Target Tools Known Before		Eclipse Experience (months)		
	OTH	OCH	<8	8-11	>11
All	1	4	3	6	9
Spyglass	1	1	2	3	4
tutorial	0	3	1	3	5

6.2 Experimental Design

The study used an independent-measures design in which participants were divided into two groups: one group was provided with a passive tutorial that included descriptions of the target tools and an Eclipse help document; the other group was provided with a passive tutorial that excluded descriptions of the target tools, an Eclipse help document, and Spyglass. We refer to the former as the tutorial group and the latter as the Spyglass group.

To try to ensure participants in both groups had roughly the same background, we tried to match participants in one group to another (as they signed up) based on their self-reported experience with Java programming, their experience with Eclipse, and the target tools they knew before. Each group ended up with two female participants and seven male participants in each group. The second and the third rows of Table 6.1 and Table 6.2 show the number of participants in each group with different levels of Java and Eclipse experience, number of participants who had worked on different sizes of Java programs, and how many of them knew some target tools before. All of the participants in the Spyglass group never worked with Java programs that consisted of more than 75 classes before, whereas three participants in the tutorial had.

6.3 Experimental Procedure

The procedure we followed in this study was the result of having conducted two pilot studies. The first one was conducted with three graduate students in our laboratory who already knew the target tools and it included only one task. The second one was conducted with four graduate students who did not know the target tools before and it included two tasks, similar to what we used in this laboratory study.

Qualified participants were scheduled to come to our laboratory to perform soft-

ware change tasks on JFreeChart. JFreeChart is a Java-based open-source chart drawing library. The participants were asked to read a provided document that introduced the main control flow of chart drawing in JFreeChart and terminology related to chart drawing (see Appendix C.5). The request sent to participants emphasized that the document would help them understand JFreeChart and would prepare them for the laboratory session.

Each trial in the lab was divided into three sessions: training, programming, and interview. The purpose of the training session, which lasted up to 30 minutes, was to get each participant familiar with the nature of the tasks, with the software development environment, and with the think-aloud protocol of the study. For the training session, participants in the tutorial group were given a tutorial document that introduces six useful Eclipse-based tools: Open Declaration, Open Call Hierarchy, Search References in Workspace, Open Type Hierarchy, Javadoc View, and File Search (see Appendix C.7). Each participant was then asked to read the tutorial and complete two training tasks in RTC 1.0 without Spyglass installed. The tasks were to fix some problems on a drawing program, called Paint[7] [13]. Each participant was encouraged to use the tools described in the tutorial during or after the training tasks, until they said they were comfortable with the tools.

Participants in the Spyglass followed roughly the same procedure. To ensure that participants in the Spyglass group knew about the other tools in the same tutorial document given to the tutorial group, which Spyglass would not be able to recommend, the experimenter gave these participants a shorter version of the tutorial document that excluded descriptions of the Open Call Hierarchy and Open Type Hierarchy tools (see Appendix C.6). The training then proceeded as for the tutorial group with the same two training tasks. The training for the Spyglass group concluded with the provision of another document that introduced them to where they could find the Spyglass icon, what different Spyglass icons meant, how a recommendation looked, and how to disable the popup notification from Spyglass (see Appendix C.8 for details).

All participants performed all of the given software change tasks using a USB desktop keyboard, a USB mouse, and a 19-inch LCD screen all connected to a Lenovo ThinkPad T60 laptop, on which all the software ran. The laptop was also connected to speakers, with audio set to the same volume for each participant.

The next session was the programming session (lasted up to 2 hours and 10 minutes), in which participants in both groups were asked to complete two similar software evolution tasks on JFreeChart library. Each participant was given the tasks in the same

[7] Andrew J. Ko provided the source to the Paint program used in his earlier study

order, and was allowed up to one hour to finish each task, but she could finish a task before an hour. There was a ten-minute break between each task.

The first task was to make the `PieChartDemo1` application draw a pie chart with a label for each pie sector drawn outside of the pie instead of on top of the pie sectors, as currently provided by the application. The second task was to make the `BarChartDemo1` application draw missing axis labels for both X and Y axes. Both tasks require a participant to understand the control flow of how JFreeChart draws a chart, which might not be obvious by tracing back from the `main` method of the applications. In the `PieChartDemo1` and `BarChartDemo1` applications, JFreeChart chart objects are created and put in an object that represents a window panel. However, when the applications are run the method `draw` in the `JFreeChart` object is called to draw the chart.

Participants in the tutorial group performed the tasks in RTC with Spyglass 1.0 installed but its UI disabled. Spyglass was running in the background to gather some data, but the participants were not aware of its existance. The Spyglass group performed the tasks in the programming session with Spyglass 1.0 installed and active.

After the first task, the experimenter explained to each participant how the pie chart in the first task was drawn with JFreeChart, starting from the `draw` method in `JFreeChart` class. The amount of explanation varied depending on how much the participant had completed in the first task. We chose to provide this information because the drawing mechanism was critical to making progress in the second task.

The interview session consisted of a post-test questionnaire that assessed a participant' feedback on both tasks and how useful they thought the two target tools were for the tasks if they used any of them (see Appendix C.4). For a participant in the Spyglass group, the questionnaire also assessed the participant's feedback on Spyglass's recommendations and notification mechanism (see Appendix C.3). The experimenter also conducted a short interview to assess how a participant normally learn how to use functionality in any new software, and to clarify the participant's answers in the post-test questionnaire. The interview protocol is provided in Appendix C.14.

During the programming session, a participant's interaction with RTC and Spyglass were recorded in the Mylyn Monitor log and the Spyglass log files (see Section 5.4.1 for details). We also used CamStudio 2.0 software[8] to capture the computer screen and the participant's speech. The experimenter also observed each participant as she worked and took notes. We encouraged participants to speak out loud what they were thinking during the tasks.

[8] `http://camstudio.org/`, verified April 16, 2009

6.4 Results

One factor that we could not and did not want to control in the experiment was the working style of our participants. Participants attacked the programming problems in a variety of ways involving a variety of navigation style. Some of the styles used by participants did not result in Spyglass being able to make a recommendation: neither for the Spyglass group participants who could see the recommendations nor for the tutorial group participants who could not see the recommendations even though Spyglass was computing possible recommendations in the background.

In our analysis, we used the information about who had a working style that could potentially receive a recommendation from Spyglass to form two implicit sub-groups: the sub-group of participants in the Spyglass group who received recommendations (denoted by $\mathcal{S}$) and the sub-group of participants in the tutorial group who would have received recommendations had Spyglass been enabled in their environment (denoted by $\mathcal{T}$). We considered these two sub-groups to have a similar working style.

Figure 6.1 shows when participants in the Spyglass group received recommendations during their first task and when they used the recommended tools. On the other hand, Figure 6.2 shows when participants in $\mathcal{T}$ used the target tools during the first task. In Task 1, three out of nine participants (e6, e7, and e8) in the Spyglass group received recommendations from Spyglass. These participants received the recommendations close in time to finish the task: one received a recommendation eleven minutes before finishing (around 35% into the task), the others received recommendations in less than five minutes before finishing the task (around 70% and 90% into the tasks). Because so few participants in the Spyglass group received recommendations in this task and because the recommendations were received so close to the end of the tasks, making it difficult for the participants to then use the tools, we chose to treat this task as an additional training task and did not analyze it any further.

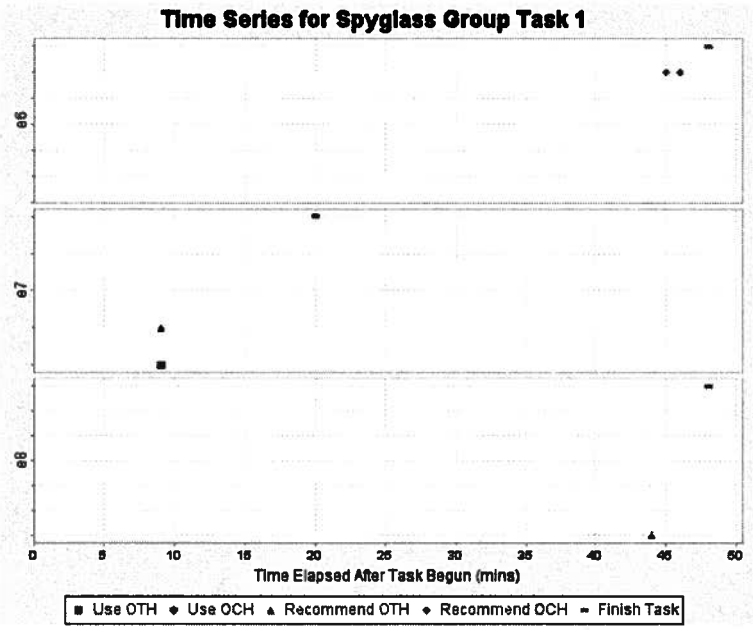


Figure 6.1: Recommendations and Use of Target Tools by Spyglass Group in Task 1

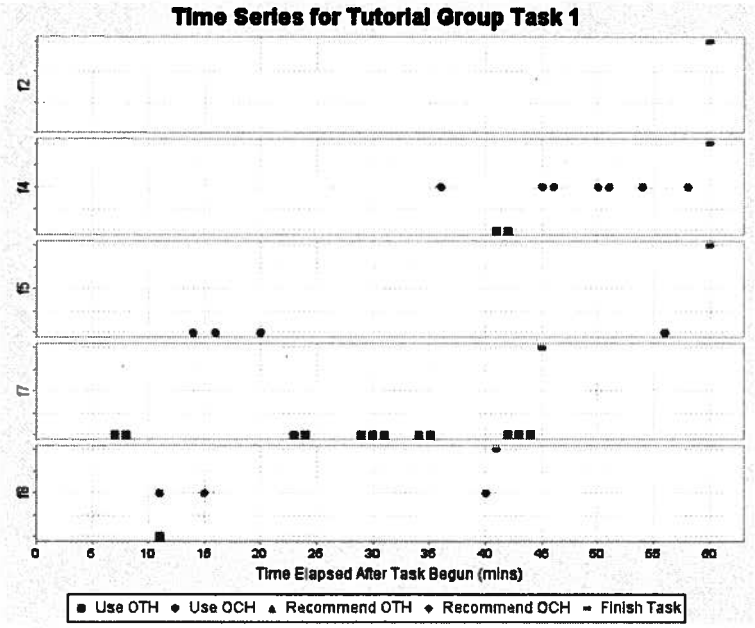


Figure 6.2: Recommendations and Use of Target Tools by Sub-Group of Tutorial Group in Task 1

In Task 2, Figure 6.3 shows that there were eight out of nine participants in $\mathcal{S}$ (e1, e3, e4, e5, e6, e7, e8, e10). These participants received recommendations earlier in this task than in Task 1. Also, there were eight out of nine participants in $\mathcal{T}$ (f2, f3, f4, f5, f6, f8, f9, f10). Figure 6.4 shows the use of target tools by these participants. Given the large number of participants with similar working style in each group, we were able to apply hypotheses testing to the data gathered from Task 2.

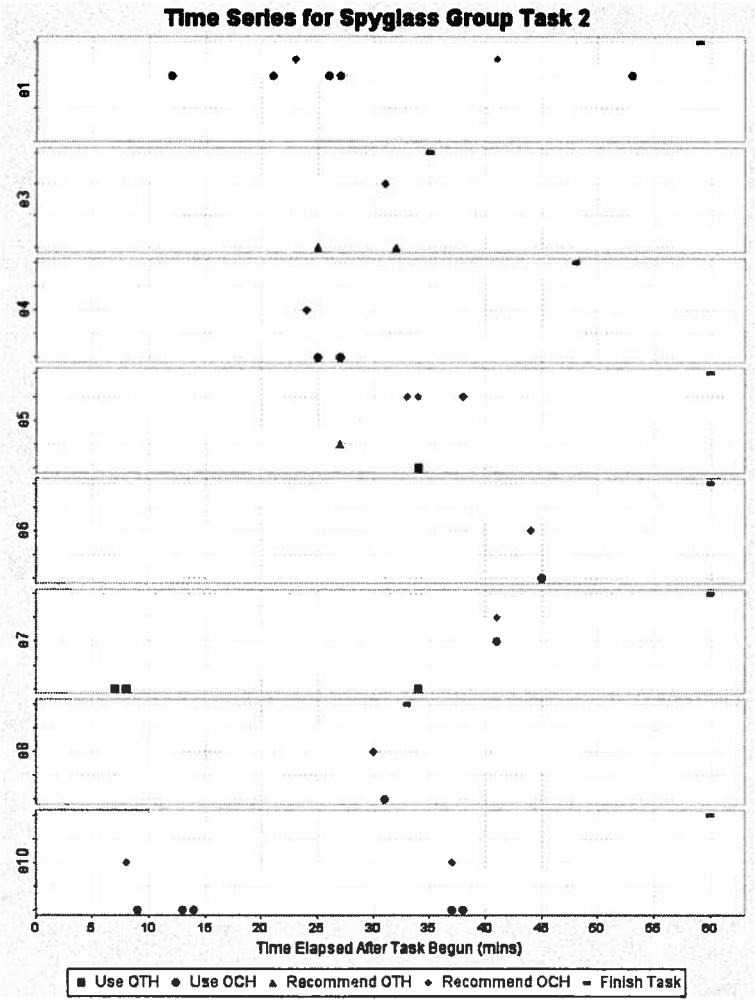


Figure 6.3: Recommendations and Use of Target Tools by Spyglass Group in Task 2

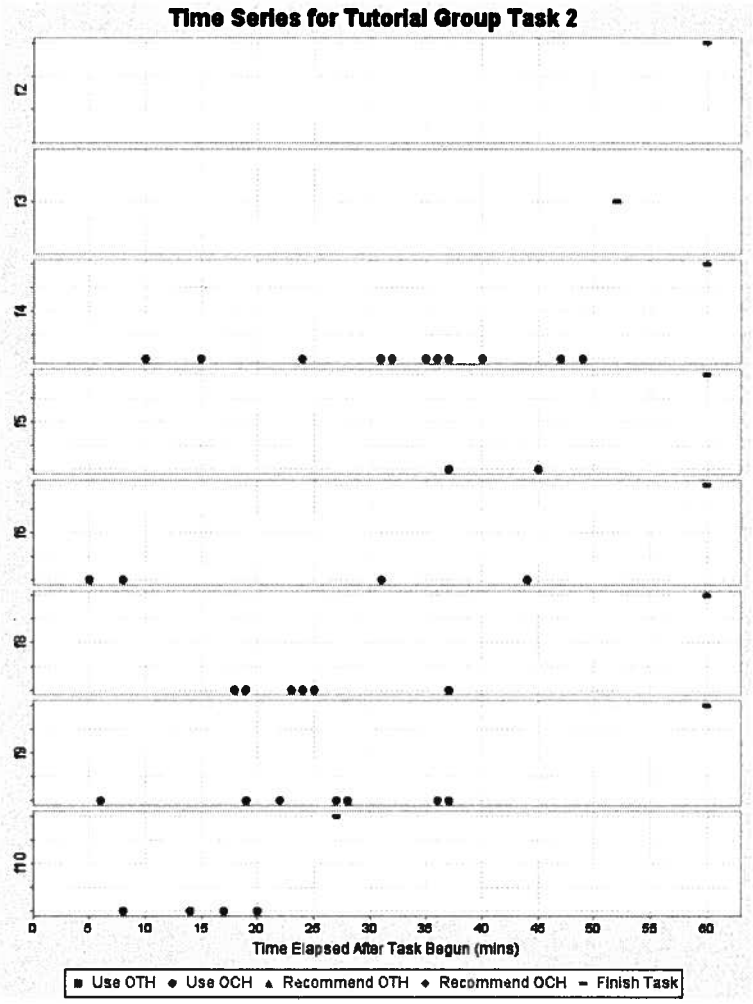


Figure 6.4: Recommendations and Use of Target Tools by Sub-Group of Tutorial Group in Task 2

6.4.1 Quantitative Results

For each question this study was intended to investigate, we derived a hypothesis and analyzed a corresponding measure. When a statistical hypothesis test was possible, we used the Mann-Whitney U test because our data either did not meet the normal distribution assumption, or the homogeneity of variance assumption, or both, which were requirements for a parametric hypothesis test. When we use the term "the task" in this section, we are referring to Task 2.

Q-1 Can Spyglass help developers complete tasks more successfully?

Hypothesis 1. *Participants who received recommendations from Spyglass complete tasks more successfully than participants in the tutorial group who have similar working style, on average.*

Measure We analyzed the source code to determine a list of code elements that are highly relevant for someone to understand the cause of the problems in each task, and we call these elements the *base elements* (see Appendix C.11 for a full list). Then, we defined a marking scheme to assess the success of a participant on the task as a score out 100%. The marking scheme rewards behaviours in which the participant visited the base elements. The scheme also gives marks based on whether or not the participant solved the problem successfully, and whether there was an evidence that the participant understood the control flow of chart drawing. Table 6.3 provides a portion of the marking scheme; the full marking scheme is provided in Appendix C.16. The scores were given by the experimenter while observing a participant during a task.

Table 6.3: Example of Marking Scheme for Task 2 Completion

Scheme No.	Description	Indication	Score
1	Looked inside `createChart` in `BarChartDemo1` and understood that `CategoryPlot` was associated with the `JFreeChart` object created and was used to draw the plot area in the bar chart.	• Hovered the mouse over `CategoryPlot` in `createChart` and read the comments, or • Opened up `CategoryPlot`	10%
2	Understood that `CategoryAxis` represented domain axis and `NumberAxis` or `ValueAxis` represented range axis in this bar chart demo.	• Hovered the mouse over `CategoryAxis` and `NumberAxis` in the method `createChart` of `BarChartDemo1`, or • Hovered the mouse over `CategoryAxis` and `Number` or `ValueAxis` in the method `createBarChart` of `ChartFactory`, or • Hovered the mouse over `CategoryAxis` and `ValueAxis` in the method `drawAxes` of `CategoryPlot`	10%

Results Table 6.4 shows the completion scores of all participants in $\mathcal{S}$ and in $\mathcal{T}$. To accept the hypothesis, the mean of scores of participants in $\mathcal{S}$ must be greater than the mean of the scores of participants in $\mathcal{T}$. Table 6.5 shows that the mean of scores of participants in $\mathcal{S}$ was higher than that of participants in $\mathcal{T}$, in Task 2. However, the result of Mann-Whitney U Test stated that the difference was not significant ($U = 19.5, p = 0.0975$, one-tailed, $N_{\mathcal{S}} = 8, N_{\mathcal{T}} = 8$). Thus, we did not have enough evidence to accept this hypothesis.

Table 6.4: Completion Scores from Task 2 Used for Testing Hypothesis 1

Participant	Group	Time Used (mins)	Completion Scores (out of 1.00)
e1	Spyglass	59	1.00
e3	Spyglass	35	1.00
e4	Spyglass	48	0.95
e5	Spyglass	60	0.20
e6	Spyglass	60	0.20
e7	Spyglass	60	0.80
e8	Spyglass	33	1.00
e10	Spyglass	60	0.60
f2	tutorial	60	0.20
f3	tutorial	52	0.85
f4	tutorial	60	0.35
f5	tutorial	60	0.10
f6	tutorial	60	0.65
f8	tutorial	60	0.35
f9	tutorial	60	0.20
f10	tutorial	27	1.00

Table 6.5: Descriptive Statistics of Completion Scores Used for Testing Hypothesis 1

Group	Mean	Stdv	Variance	Mean Rank
Spyglass	0.719	0.348	0.121	10.06
tutorial	0.462	0.331	0.110	6.94

Q-2 Can Spyglass help navigate through pieces of information more efficiently?

Hypothesis 2. *Participants who have received recommendations from Spyglass will visit less code elements that are unrelated to a task than participants in the tutorial group who have similar working style, on average.*

Measure We have defined base elements as code elements that are highly relevant for someone to understand the cause of the problems in each task above. For this hypothesis, we define code elements that we consider *unrelated to the task* as code elements that are unrelated to the base elements or any code element that is related to the base elements.

A Java code element is considered as related to the base elements if it is called or referenced by any base element or ancestors of any base element. A code element that is called or referenced by any related element or is an ancestor of any related element is also considered related to the base elements.

We extracted from the Mylyn log data all code elements each participant has visited and counted the number of the ones that were unrelated to the task. We used the number as our measure for evaluating how efficiently participants in both groups navigated source code in order to complete the tasks.

Results To accept Hypothesis 2, the mean of the number of code elements unrelated to the tasks visited by participants in $\mathcal{S}$ must be smaller than the mean of the number of unrelated code elements visited by participants in $\mathcal{T}$. Table 6.6 shows the number of unrelated code elements visited by every participant in $\mathcal{S}$ and $\mathcal{T}$. On average, participants in $\mathcal{S}$ visited less unrelated code elements than the ones in $\mathcal{T}$, as we expected (see Table 6.7). The result from the Mann-Whitney U test indicated that the difference was significant ($U = 10.5, p = 0.0105$, one-tailed, $N_{\mathcal{S}} = 8, N_{\mathcal{T}} = 8$). Therefore, we can accept this hypothesis.

It was possible that this result occured because the individuals in $\mathcal{S}$ were more competent than the individuals in $\mathcal{T}$. To investigate this possibility, we matched participants in $\mathcal{S}$ with participants in $\mathcal{T}$ who had similar completion scores (see Table 6.4). Then, we had five pairs of participants: (e10, f6), (e7, f3), (e5, f2), (e6, f9), and (e8, f10). We will refer to the sub-group of $\mathcal{S}$ as $\mathcal{S}$' and the sub-group of $\mathcal{T}$ as $\mathcal{T}$'. We computed the average number of unrelated code elements visited for each sub-group ($\bar{X}'_{\mathcal{S}}$ and $\bar{X}'_{\mathcal{T}}$), finding $\bar{X}'_{\mathcal{S}} = 3.2$ and $\bar{X}'_{\mathcal{T}} = 7.2$. These results are consistent with the average number of unrelated code elements visited by all participants in $\mathcal{S}$ ($\bar{X}_{\mathcal{S}} = 3$) and all participants in $\mathcal{T}$ ($\bar{X}_{\mathcal{T}} = 7.78$). This results provides evidence that the difference in

Table 6.6: Number of Unrelated Code Elements Visited in Task 2 Used for Testing Hypothesis 2

Participant	Group	Number of Unrelated Code Elements Visited
e1	Spyglass	0
e3	Spyglass	9
e4	Spyglass	2
e5	Spyglass	1
e6	Spyglass	2
e7	Spyglass	4
e8	Spyglass	4
e10	Spyglass	5
f2	tutorial	10
f3	tutorial	12
f4	tutorial	14
f5	tutorial	13
f6	tutorial	3
f8	tutorial	5
f9	tutorial	8
f10	tutorial	3

Table 6.7: Descriptive Statistics of Number of Unrelated Code Elements Used for Testing Hypothesis 2

Group	Mean	Stdv	Variance	Mean Rank
Spyglass	3.38	2.83	7.98	5.81
tutorial	8.50	4.44	19.71	11.19

the number of unrelated code elements was not influenced by participants' competency (as measured by their completion scores).

Q-3 Can Spyglass raise a developer's awareness and use of tools in a development environment?

We did not expect an active help system to be more self-explanatory than a step-by-step tutorial document. However, we expected that an active help system could potentially be more effective in introducing more tools to a user on as-needed basis. As our laboratory experiment only lasted for up to three hours, and participants in the tutorial group learned and practiced using the new tools right before the tasks, we thought that the participants in the tutorial group might remember the target tools and used them

frequently. We thus expected that, Spyglass could at best, in the context of this study, be as effective as a passive tutorial that was given just before the tasks in making a programer aware of and learn how to use new tools in RTC.

Hypothesis 3. *On average, participants who receive recommendations from Spyglass interact with the target tools that are recommended at least as often as participants in the tutorial group who have similar working style.*

Measure To assess this hypothesis, we consider the number of times each participant interacted with a target tool. We retrieved this number from the Mylyn log data. However, this simple measure does not account for the different patterns of recommendation and use that Figure 6.1 and Figure 6.3 depict. To account for the time from when a participant first knew about the tool (either by Spyglass or from tutorial) until she finished a task, we introduce the *density of tool usage* or DU (unit is frequency/minute):

$$DU_{ij} = \frac{\text{No. of interactions with tool i}}{\text{time that participant j has known tool i}} \tag{6.1}$$

We computed the denominator from the first time that participant j was introduced with tool i (either by Spyglass or by tutorial) until the end of the task in minutes. If a participant learned about tool i in the first task, we considered that the participant has known about tool i from the beginning of the second task til the end.

Results Table 6.8 shows the usage frequency and DU of each participant in $\mathcal{S}$ and $\mathcal{T}$. On average, the simple count of interactions with target tools of participants in $\mathcal{S}$ was lower than of participants in $\mathcal{T}$, in Task 2 (see Table 6.9). However, on average, the DU's of participants in $\mathcal{S}$ were higher than that of participants in $\mathcal{T}$ for the Open Type Hierarchy tool but were almost equal for the Open Call Hierarchy tool (see Table 6.10). The difference was not significant, as confirmed by the result of Mann-Whitney U test on DU's, for each tool. For the Open Type Hierarchy tool, $U = 8, p = 0.107$, one-tailed, $N_{\mathcal{S}} = 4, N_{\mathcal{T}} = 8$. For the Open Call Hierarchy tool, $U = 26.5, p = 0.287$, one-tailed, $N_{\mathcal{S}} = 8, N_{\mathcal{T}} = 8$.

Therefore, given the same amount of time participants have known about a tool, participants who learned about a tool from Spyglass interacted with it as frequently as participants who learned about the same tool from a tutorial. Also, as a side note, Table 6.8 shows that the number of participants in $\mathcal{S}$ who used the recommended tools is equal to the number of participants in $\mathcal{T}$ who used any of the target tools.

Table 6.8: Target Tools Usage in Task 2 Used for Testing Hypothesis 3

Participant	Group	Frequency of Using OTH	Frequency of Using OCH	Time Known OTH (mins)	Time Known OCH (mins)	DU for OTH	DU for OCH
e1	Spyglass	0	8	0	60	-	0.13
e3	Spyglass	0	0	10	4	0	0
e4	Spyglass	0	4	0	24	-	0.17
e5	Spyglass	2	0	33	27	0.06	0
e6	Spyglass	0	1	0	60	-	0.02
e7	Spyglass	9	1	60	19	0.15	0.05
e8	Spyglass	0	3	60	3	0	1
e10	Spyglass	0	8	0	52	-	0.15
f2	tutorial	0	0	60	60	0	0
f3	tutorial	0	0	60	60	0	0
f4	tutorial	0	31	60	60	0	0.52
f5	tutorial	0	3	60	60	0	0.05
f6	tutorial	0	8	60	60	0	0.13
f8	tutorial	0	21	60	60	0	0.35
f9	tutorial	0	15	60	60	0	0.25
f10	tutorial	0	11	60	60	0	0.18

Table 6.9: Descriptive Statistics of Tools Usage Frequency in Task 2

Tool	Group	Mean	Stdv	Variance
OTH	Spyglass	1.38	3.16	9.98
	tutorial	0	0	0
OCH	Spyglass	3.13	3.31	10.98
	tutorial	11.13	10.90	118.70

Table 6.10: Descriptive Statistics of Density of Tools Usage in Task 2

Tool	Group	Mean	Stdv	Variance
OTH	Spyglass	0.053	0.071	0.005
	tutorial	0	0	0
OCH	Spyglass	0.190	0.334	0.112
	tutorial	0.185	0.182	0.033

Q-4 Are the tools recommended by Spyglass perceived as useful by developers?

Hypothesis 4. *Participants who receive recommendations from Spyglass find recommended tools useful on average.*

Measure In the post-experiment questionnaire, we asked each participant in $\mathcal{S}$ to rate the usefulness of any target tool recommended by Spyglass that they used on a five-point Likert scale (from 1=very useless to 5=very useful). We computed the average scores from the answers and used it as our measure to investigate this hypothesis.

Results The answers to the question regarding the usefulness of each recommended tool from the post-experiment questionnaire are shown in Table 6.11. The mean of the scores from participants in $\mathcal{S}$ was equal to 4 ($Stdv. = 1$) in the case of the Open Call Hierarchy tool, and was equal to 3.5 ($Stdv. = 0.71$) in the case of the Open Type Hierarchy tool as shown in Figure 6.5. If you consider the distribution of the ratings given by participants in $\mathcal{S}$ for each tool (which are normally distributed), you will see that the lower quartile and the upper quartile for the ratings of the Open Type Hierarchy tool are closer to each other than in the case of the Open Call Hierarchy tool.

Table 6.11: Usefulness of Target Tools Used for Testing Hypothesis 4

Participant	Group	Usefulness of OTH	Usefulness of OCH
e1	Spyglass	-	5
e3	Spyglass	-	-
e4	Spyglass	-	3
e5	Spyglass	-	-
e6	Spyglass	-	5
e7	Spyglass	3	-
e8	Spyglass	4	4
e10	Spyglass	-	3
f2	tutorial	-	-
f3	tutorial	-	-
f4	tutorial	3	4
f5	tutorial	-	2
f6	tutorial	4	5
f8	tutorial	-	4
f9	tutorial	3	4
f10	tutorial	-	5

This indicates that the opinions of participants in $\mathcal{S}$ who used the Open Call Hierarchy tool as recommended by Spyglass varied more than of the Open Type Hierarchy

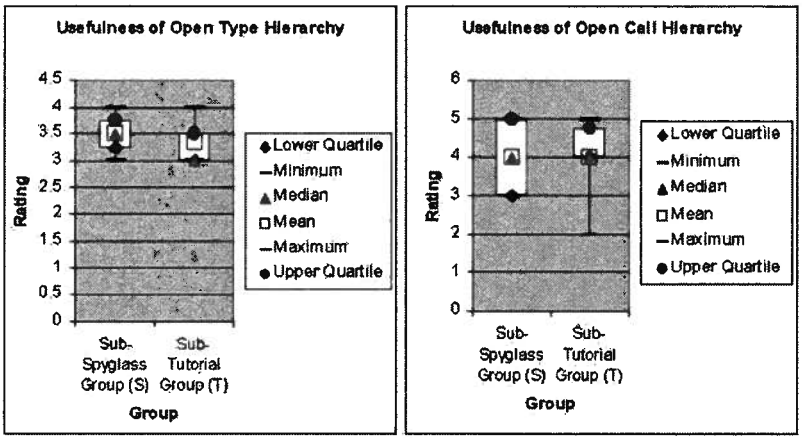


Figure 6.5: Usefulness of Target Tools

tool, but they were all above being "useless". Thus, we could conclude that participants in $\mathcal{S}$, on average, thought that the Open Call Hierarchy tool was useful, while they were indifferent for the Open Type Hierarchy tool.

Also, on average, the ratings for both tools by participants in $\mathcal{T}$ are about the same as the ones from $\mathcal{S}$ ($Mean = 3.33$, $Stdv. = 0.58$) for the Open Type Hierarchy tool and ($Mean = 4$, $Stdv. = 1.10$) for the Open Call Hierarchy tool.

Q-5 Do developers perceive Spyglass as suggesting the right tools at the right time?

Fischer stated that a good active help system must be able to recommend the right tool to users at the right time and in the right way [6]. This question addresses how well Spyglass performs as an active help system.

Hypothesis 5. *Participants who receive recommendations from Spyglass find the recommendations useful on average.*

Measure In the post-experiment questionnaire, we asked each participant in $\mathcal{S}$ to rate the usefulness of recommendations they received from Spyglass with respect to what they were doing at that time and the usefulness of rationales accompanying each recommendation on five-point Likert scales (from 1=very useless to 5=very useful). We computed the average scores from the answers and used them to investigate this hypothesis.

Results Table 6.12 shows how often participants in $\mathcal{S}$ looked at the recommendations (either in the popup view or in the Recommendation View) and how useful they felt about the recommendations, with respect to what they were doing at that time. The mean of the ratings was 2.5 ($Stdv. = 0.98$), which implied that they felt the recommendations were not useful with respect to what they were doing.

Table 6.12: Usefulness of Recommendations and Their Rationales

Participant	Group	How often did you look at recommendations?	Usefulness of Recommendations	Did you read the rationale?	Usefulness of Rationales
e1	Spyglass	100%	3	Yes	4
e3	Spyglass	25%	2	Yes	3
e4	Spyglass	100%	1	Yes	5
e5	Spyglass	25%	2	Yes	2
e6	Spyglass	100%	3	No	-
e7	Spyglass	100%	2	Yes	3
e8	Spyglass	100%	4	Yes	3
e10	Spyglass	100%	3	No	-

Table 6.12 also shows which participants read the rationale accompanying a recommendation and how useful they felt about the rationales. The mean of the ratings was 3.33 ($Stdv. = 1.10$), which indicated neutral feelings.

Q-6 How often does Spyglass suggest the right tool at the right time?

Aside from the hypothesis above, we were also curious about how often Spyglass gave relevant recommendations (the ones that the user would feel useful). Thus, we also attempted to calculate a precision rate for Spyglass recommendations. The *precision* is a ratio of the number of relevant recommendations to the number of all recommendations given.

To compute the precision, we extracted the total number of times recommendations appeared (or would have appeared for participants in $\mathcal{T}$) along with the timestamps from the Spyglass log, and then assessed, using the recorded videos, which ones were relevant and which ones were not. To decide relevancy, we looked three minutes before and after a recommendation, if there was evidence that the user was looking for code elements that were related to a particular element in a call relationship or in an inheritance relationship and the recommendation was for that purpose, then we said the

recommendation was relevant (see Appendix C.13 for full details).

In total, 62 recommendations appeared for participants in $\mathcal{S}$ (or would have appeared for participants in $\mathcal{T}$) in both Task 1 and Task 2. Of those recommendations, 33 recommendations were determined as relevant by us. Thus, the precision of Spyglass recommendations was equal to 0.53, meaning only half of the time was a Spyglass recommendation relevant. Given that the participants also did not report the recommendations as useful with respect to what they were doing, Spyglass does not yet suggest the right tool at the right time.

Q-7 Does Spyglass's popup notification mechanism overly disturb a developer's workflow?

Hypothesis 6. *Participants who receive recommendations from Spyglass do not find the notification mechanism disturbing on average.*

Measure For the revised notification mechanism used in the laboratory study that was based on feedback from the longitudinal study, we asked participants in $\mathcal{S}$ to rate how much the notification popup and sound disturb her during the task (from 1=not at all to 5=very disturbing) on five-point Likert scales, in the post-experiment questionnaire. We computed the average ratings to investigate this hypothesis.

Results Table 6.13 shows the ratings from participants in $\mathcal{S}$ regarding disturbance of popup and sound notification of Spyglass. The mean ratings for questions regarding popup and sound notifications in the post-test questionnaire were both equal to 1.63 ($Stdv. = 1.13$ for the popup notification and $Stdv. = 1.25$ for the sound notification). Thus, we concluded that, on average, participants who received recommendations from Spyglass did not feel that having popup notification or sound notification when there was a new recommendation was disturbing.

Table 6.13: Disturbance of Popup and Sound Notification

Participant	Group	Disturbance of Popup	Disturbance of Sound Notification
e1	Spyglass	3	1
e3	Spyglass	4	3
e4	Spyglass	1	4
e5	Spyglass	1	1
e6	Spyglass	1	1
e7	Spyglass	1	1
e8	Spyglass	1	1
e10	Spyglass	1	1

6.4.2 Qualitative Results

During the interviews, three out of eight participants who received recommendations from Spyglass expressed that they liked having Spyglass in the development environment. Each of these participants received two to three recommendations throughout both tasks and three out of five recommendations were rated (by us) as relevant. One of these participants said that,

> I think it's a great idea... And since there is a lot of stuff going on, particularly for a new user, I think it's extremely helpful to have something to help you get around it. Anyway, I think it's very useful. It's not going to interrupt you if you are using the most efficient path anyway, and if you're not, it will help you do it more efficiently. I mean–there's not really a down side to it–from my perspectives.(e8)

The other four participants who received recommendations from Spyglass said that Spyglass could potentially be useful with some improvements. One suggested that Spyglass should recommend more often; another suggested that Spyglass should be more precise and take into account other factors (such as the user's idling time) before deciding when to give a recommendation. Each of these four participants seemed to be confused by the recommendations because they either did not feel that the code elements mentioned in the rationale were what they were looking for or they were led to completely unrelated code elements after they tried using the recommended tool on the current method/class in focus. However, all of them but one had some idea about the functionality provided by the recommended tools from the tool's names. One participant suggested that Spyglass might be more suitable for a developer who was very

experienced in Java and familiar with the problem domain, but was novice to the development environment. She said that such developer would not feel too overwhelmed or frustrated by the task to try using the recommended tools, and he/she might understand what Spyglass was trying to tell him/her better.

Only one participant expressed that she was not satisfied with Spyglass and did not use any recommended tool. She said that the recommendation did not seem to relate to what she was doing and the popup notification and sound were annoying, while she was trying to solve the tasks. She also did not understand what the recommended tools were for:

> ...because I don't use the recommendation system when I'm using Eclipse, so I just see it as an annoying popup. If I actually use it more and rely on it, it would have been more... I don't know. Whenever I'm using some software and there is a popup, I just have the tendency to close it and ignore it.(e3)

We asked some participants who did not use the target tools more than once right after they received recommendations why they did not use the tools more often. Some of them said that there were too many results returned by the tools, which were hard for them to find the ones related to the tasks, and thus, they felt that the recommended tools were not more helpful than the other tools. Some of them said that the recommended tools did not "sink in" to them, yet, so they tended to forget and fall back to use the tools that they were already familiar with.

As for the tutorial group, five out of nine participants remembered both the Open Call and Open Type Hierarchy tools from the tutorial and used them during the tasks. Three out of nine remembered only one of the target tools, while one participant remembered none of the target tools. There was also one participant who claimed that she remembered both of the tools, but she used neither of them. When asked, she said that she preferred to use the tools that she was already familiar with.

We also asked participants from both groups how they usually learned to use new software. Most participants in both groups said that they usually learned how to use new software by trial-and-error, or by browsing through menu bars or context menus. They would look up for tutorials or examples on internet only when they encounter some problems, or they could not find out by themselves how to accomplish some task.

6.5 Results Summary

The results of the laboratory study confirmed that Spyglass was at least as effective as passive tutorial document in terms of making developers aware of new useful tools and encouraging them to use the tools. It also confirmed that using sliding popup window and sound to notify developers of a new recommendation was not obtrusive for majority of developers.

The study also showed that Spyglass helped keep participants on the right track since the participants who received the recommendations visited significantly less code elements unrelated to the task than participants (with similar working style) in the tutorial group, on average. Although we could not detect significant difference between the average completion scores of participants who received recommendations and participants in the tutorial group who had similar working style, the former was higher than the latter as we expected.

One may wonder how could Spyglass help participants visit less unrelated code elements even though those participants did not appear to use the recommended tools as frequently as the participants in the tutorial group who had similar working style. We have looked back at the usage of other tools introduced in the tutorial given to the participants in both groups at the beginning. It turned out that even though participants who received recommendations did not use the recommended tools as frequently, they did use other tools presented in the tutorial (e.g., Open Declaration, Search References) more frequently than participants in the tutorial group with similar working style on average.

Thus, one explanation regarding this phenomenon is the small tutorial presented to the Spyglass group made it easier for the participants in that group to remember the tools described in that passive tutorial. These tools could help keep participants on the right track as well as the target tools even though they might not save the number of elements visited along the way. Frequently use of tools also does not necessary imply effective use of tools.

The study also demonstrated that Spyglass might need to improve its accuracy in recommending the right tool at the right time. Participants, on average, did not feel that the recommendations given by Spyglass were useful in the context of what they were doing. The computed precision of Spyglass's recommendations was 0.53, meaning that, half of the time, Spyglass gave irrelevant recommendations to participants.

The qualitative results confirmed the findings from the quantitative analysis. Four out of eight participants who received recommendations from Spyglass said that Spyglass could be potentially useful with improvement in its precision. Only one out of

the eight participant did not think that Spyglass's recommendations were useful and did not try to use the recommended tools at all even though she received recommendations three times. The other three expressed that they liked Spyglass; however, these three participants received a small number (two to three) of mostly relevant recommendations.

Our interviews confirmed that majority of participants in both groups usually learned how to use new software (not necessary Eclipse) by trial-and-error, or by browsing through menu bars or context menus. They would look up for tutorials or examples on internet only when they encounter some problems, or they could not find out by themselves how to accomplish some task.

6.6 Threats to Validity

In this section, we will discuss possible threats to validity of our laboratory user study.

6.6.1 Construct Validity

As a measure of navigational efficiency, we used number of code elements that were unrelated to the tasks visited to measure the navigational efficiency of each participant. Although this measure could tell us which participants were on the right track and which ones were lost, it did not tell us directly whether a participant used the shortest possible path to navigate between the structurally related code elements they visited or not, which was what Spyglass was trying to encourage. This is a threat to the construct validity of our study.

Other threats to construct validity in our study are social threats like hypothesis guessing (participants were guessing what the experimenter was looking to measure) and the fact that the experimenter was sitting behind them, observing them, and writing notes may have disturbed some participants or affected the decisions and behaviours of participants. Some participants may have taken the experimenter's action of taking notes as a hint that they were in the right/wrong track.

6.6.2 Internal Validity

Programmers' ability and working methods are vastly different [19]. Even though we have tried to make sure that participants in both group have similar background as much as possible, we still found during the studies that the background information did not really reflect each person's skills accurately. This may be remedied if we had

more participants or if we asked them to do some pretests and changed the design to a matched-pairs experiment.

As a result, we further categorized participants in each group based on the fact that they received or would have received recommendations from Spyglass. Since Spyglass would only be able to detect suboptimal behaviours if developers navigate through domain artifacts in certain ways, we were able to separate different working style of participants who did not recieve or would not have received any recommendation from participants who received or would have received some recommendations.

6.6.3 External Validity

We studied only two tasks with bugs introduced to the system to the create the tasks by ourselves. This may not reflect tasks in the field even though we have made sure that the program size and complexity are compatible with industry programs.

The study was run in the laboratory, under a short period of time, which differs from a real work environment in which each task typically has more time allocated to it. If participants were allowed to complete the tasks at their own pace, they might have been more willing to try out new tools recommended by Spyglass. Moreover, in a real work environment, developers are usually given more time to learn about the program they are going to work on before they are assigned to make any change to it, while we only asked them to read the introduction document without letting them see the program before beginning the tasks.

Chapter 7

Discussion

From the studies we have conducted, we have determined several ways in which the Spyglass approach and the user interface could be improved. We have also learned about potential improvements to the studies used to assess approaches like Spyglass. We discuss each of these areas in this chapter.

7.1 Approach Improvements

Based on the results from the laboratory study, Spyglass provides relevant recommendations half of the time. There are a number of possible ways to improve the precision of Spyglass's recommendations.

For example, we can improve the user model of Spyglass by taking into account whether a user has used target tools in the past few days, and if so, how often. Spyglass can use this information to recommend only tools the user is not using. Spyglass may also only recommend a tool that has shown up in the Recommendation History more than a certain number of times, perhaps three. Moreover, if a user ignores a recommended tool after it has been recommended more than a certain number of times, Spyglass should not recommend that tool again.

Aside from improving the user model, we can also try experimenting with Spyglass's configuration parameters, such as the maximum size of the windows of recent activities, or the threshold that sets the minimum number of intermediate selection events between source and target domain artifacts for which Spyglass will consider computing proximity.

Another reason that might have cause the recommendation to be given out of context was the delay time in computation. Spyglass performed any computation before giving a recommendation in a separate thread, and notified the main thread of new recommendations once it was done. The computation time could have been long because of the size of structural graph being retrieved or because of suboptimal algorithms used in implementation, and thus, the recommendation might be given to the user a little bit later than it was supposed to. This problem can be solved if we have a faster machine

or if we try optimizing the computation process.

7.2 UI Improvements

Some participants of the longitudinal study suggested that Spyglass should also provide suggestions passively, such as recommending a new tool on demand, rather than based on the current task. A possible improvement is to make Spyglass a mixed-initiative system supporting both active and passive mode [11]. However, rather than giving a recommendation about random tools in the passive mode, Spyglass should take the user's initiation as a sign to start finding what tool will be appropriate for the user given the user's recently selected artifacts up to that point.

7.3 Study Improvements

More conclusive results might be obtained from a laboratory study that used a matched-pairs design. One way to run such a study is to have qualified participants (decided by the first questionnaire) come for two experimental sessions each. On the first visit, each participant would be given the Eclipse tutorial (introducing some basic useful functionalities) that excludes the target tools. They would be asked to perform the first task with JFreeChart within an hour (fixing the `PieChartDemo1`) in RTC with Spyglass UI disabled. However, Spyglass would be left running in the background to collect the participants' interactions and recommendation history. At the end of the task, every participant would be shown the correct solution and would be given more explanation about the control flow. Then, we would measure the completion scores and the working style of each participant based on the data we collected, and we would assign a matched-pair of participants into two different groups: the tutorial and the Spyglass group.

On the second visit, participants in the tutorial group would be given the Eclipse tutorial again, but with the target tools included. Participants in the Spyglass group would be given a brief tutorial about Spyglass. Then, participants in both groups would be asked to perform the second task with JFreeChart (fixing the `BarChartDemo1`) within an hour. The Spyglass group would work in RTC with Spyglass UI enabled, while the tutorial group would work in RTC with Spyglass UI disabled (but still running in the background). After an hour, participants would be engaged in a semi-structured interview with the experimenter. This visit should take up to two hours.

This type of matched-pair experiment would reduce threats to internal validity.

Also, we could reduce social threats to construct validity by conducting an experiment in a private room that is partitioned into two sections: the user study area and the observer area. So, the observer would not be sitting behind the participant's back while she was performing a task, but would observe the participant through a surveillance camera through a monitor screen that is connected to the computer on which the participant was performing the task.

Chapter 8

Conclusion and Future Work

Current software development environments have evolved to support more tools (functionality) than users can possibly know about or use effectively. We investigated the use of an active help system to make developers aware of useful tools for navigation at a point in their activities when the tool(s) might help. This active help system, called Spyglass, considers the static structural relationships between pieces of information whose models are stored in the development environment (i.e., domain artifacts) and compares the navigation path between these informations that a user has taken to an optimal path that can possibly be obtained by using an available tool. Through multiple user studies–one longitudinal and one in the laboratory–we evaluated the effects of the approach on developers in terms of improvement in their performance, and their awareness of new tools and how to use them. Through those studies, we also evaluated the performance of our approach.

The results of the laboratory study confirmed that our active help system did improve developers' navigation between domain artifacts and it was as effective as a tutorial document in terms of making developers aware of new tools. However, half of the participants who received recommendations from Spyglass in the laboratory study suggested that our prototype still needed improvement to provide more meaningful and accurate recommendations. We have also confirmed through both longitudinal and laboratory studies that majority of participants would like to have such active help system in their development environments. They agreed that our active help system had user-friendly interface and was not obtrusive.

While our prototype shows promise, more work is needed so that it recommends tools at the right time. Some possibilities to achieve this goal include investigating a more complete user model, finding the best configuration parameters for the system, and revising the algorithm for making a recommendation. One may want to investigate how our approach can be expand to a bigger domain–how it can recommend tools other than navigational tools. There should also be a thorough comparative study between existing approaches to active help systems and our approach, in an integrated development environment like Eclipse, to investigate the pros and cons of each approach with

respect to software development.

Bibliography

[1] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39, New York, NY, USA, 2005. ACM.

[2] Richard R. Burton and John Seeley Brown. An investigation of computer coaching for informal learning activities. *International Journal of Man-Machine Studies*, 11(1):5–24, 1979.

[3] John M. Carroll and Caroline Carrithers. Training wheels in a user interface. *Commun. ACM*, 27(8):800–806, 1984.

[4] David N. Chin. Empirical evaluation of user models and user-adapted systems. *User Modeling and User-Adapted Interaction*, 11(1-2):181–194, 2001.

[5] Leah Findlater, Joanna McGrenere, and David Modjeska. Evaluation of a role-based approach for customizing a complex development environment. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 1267–1270, New York, NY, USA, 2008. ACM.

[6] Gerhard Fischer. User modeling in human-computer interaction. *User Modeling and User Adapted Interaction*, 11(1-2), 2001.

[7] Gerhard Fischer, Andreas Lemke, and Thomas Schwab. Active help systems. In *Proc. of the 2nd European conference on Readings on cognitive ergonomics - mind and computers*, pages 116–131, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[8] Gerhard Fischer, Andreas Lemke, and Thomas Schwab. Knowledge-based help systems. In *CHI '85: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 161–167, New York, NY, USA, 1985. ACM.

[9] Saul Greenberg. *The computer user as toolsmith: the use, reuse, and organization of computer-based tools*. Cambridge University Press, New York, NY, USA, 1993.

[10] A. Gupta and P. Jalote. An experimental evaluation of the effectiveness and efficiency of the test driven development. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 285–294, Sept. 2007.

[11] Eric Horvitz. Principles of mixed-initiative user interfaces. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 159–166, New York, NY, USA, 1999. ACM.

[12] Eric Horvitz, Jack Breese, David Heckerman, David Hovel, and Koos Rommelse. The lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *In Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 256–265. Morgan Kaufmann, 1998.

[13] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented ides: A detailed study of corrective and perfective maintenance tasks. In *Proceeding of the 27th International Conference on Software Engineering (ICSE). To Appear*, pages 126–135, 2005.

[14] Jrgen Krause, Eva Mittermaier, and Astrid Hirschmann. The intelligent help system comfohelp. *User Modeling and User-Adapted Interaction*, 3(3):249–282, 1993.

[15] Bill Kules. User modeling for adaptive and adaptable software systems. `http://www.otal.umd.edu/uuguide/wmk/`, April 2000.

[16] Joanna McGrenere, Ronald M. Baecker, and Kellogg S. Booth. An evaluation of a multiple interface design solution for bloated software. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 164–170, New York, NY, USA, 2002. ACM.

[17] J. Mitchell and B. Shneiderman. Dynamic versus static menus: an exploratory comparison. *SIGCHI Bull.*, 20(4):33–37, 1989.

[18] Gail C. Murphy, Petcharat Viriyakattiyaporn, and David C. Shepherd. Using activity traces to characterize programming behavior beyond the lab. To Appear In ICPC '09: The 17th International Conference on Program Comprehension, 2009.

[19] M.P. Robillard, W. Coelho, and G.C. Murphy. How effective developers investigate source code: an exploratory study. *Software Engineering, IEEE Transactions on*, 30(12):889–903, Dec. 2004.

[20] Mary Beth Rosson. Human factors in programming and software development. *ACM Comput. Surv.*, 28(1):193–195, 1996.

[21] Andrew Sears and Ben Shneiderman. Split menus: effectively using selection frequency to organize menus. *ACM Trans. Comput.-Hum. Interact.*, 1(1):27–51, 1994.

[22] Ben Shneiderman. Promoting universal usability with multi-layer interface design. In *CUU '03: Proceedings of the 2003 conference on Universal usability*, pages 1–8, New York, NY, USA, 2003. ACM.

[23] Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. User interface façades: towards fully adaptable user interfaces. In *UIST '06: Proceedings of the 19th annual ACM symposium on User interface software and technology*, pages 309–318, New York, NY, USA, 2006. ACM.

[24] Bill Tomlinson, Eric Baumer, Man Lok Yau, Paul Mac Alpine, Lorenzo Canales, Andrew Correa, Bryant Hornick, and Anju Sharma. Dreaming of adaptive interface agents. In *CHI '07: CHI '07 extended abstracts on Human factors in computing systems*, pages 2007–2012, New York, NY, USA, 2007. ACM.

[25] Petcharat Viriyakattiyaporn and Gail C. Murphy. Challenges in the user interface design of an ide tool recommender. To Appear In CHASE '09: The 2009 International Workshop on Cooperative and Human Aspects of Software Engineering, 2009.

[26] Maria Virvou and Benedict Du. Human plausible reasoning for intelligent help. *User Modeling and User-Adapted Interaction*, 9(4):321–375, 1999.

[27] Brian Whitworth. Polite computing. *Behaviour and Information Technology*, 24:353–363(11), September 2005.

[28] Robert Wilensky, Yigal Arens, and David Chin. Talking to unix in english: an overview of uc. *Commun. ACM*, 27(6):574–593, 1984.

Appendix A

BREB Approval Certificate

A.1 For the Longitudinal Study

The University of British Columbia
Office of Research Services
Behavioural Research Ethics Board
Suite 102, 6190 Agronomy Road, Vancouver, B.C. V6T 1Z3

CERTIFICATE OF APPROVAL - MINIMAL RISK

PRINCIPAL INVESTIGATOR:	INSTITUTION / DEPARTMENT:	UBC BREB NUMBER:
Gail C. Murphy	UBC/Science/Computer Science	H08-01848

INSTITUTION(S) WHERE RESEARCH WILL BE CARRIED OUT:

Institution	Site
UBC	Vancouver (excludes UBC Hospital)

Other locations where the research will be conducted:
N/A

CO-INVESTIGATOR(S):
Petcharat Viriyakattiyapom

SPONSORING AGENCIES:
IBM Canada Ltd.
Natural Sciences and Engineering Research Council of Canada (NSERC)

PROJECT TITLE:
Evaluation of a Tool Recommendation System for a Software Development Environment

CERTIFICATE EXPIRY DATE: September 23, 2009

DOCUMENTS INCLUDED IN THIS APPROVAL:	DATE APPROVED: September 23, 2008	
Document Name	**Version**	**Date**
Consent Forms:		
Consent Form (PDF version of a web page)	1.4	September 11, 2008
Advertisements:		
recruitment email	N/A	August 28, 2009
Questionnaire, Questionnaire Cover Letter, Tests:		
semi-structured interview questions	N/A	August 28, 2009
preliminary questionnaire	1.2	August 28, 2008
code exploration questions protocol	1.0	August 28, 2008
recurring questionnaire	1.1	August 28, 2008
post-experiment questionnaire	1.3	August 28, 2008
Other:		

The web site will be used to sign the consent form (attached in Section 9.2) and to serve the preliminary questionnaire (attached in Section 9.6). The URL will be http://stavanger.cs.ubc.ca:8080/spyglass/consent.htm.

The application for ethical review and the document(s) listed above have been reviewed and the procedures were found to be acceptable on ethical grounds for research involving human subjects.

Approval is issued on behalf of the Behavioural Research Ethics Board and signed electronically by one of the following:

Dr. M. Judith Lynam, Chair
Dr. Ken Craig, Chair
Dr. Jim Rupert, Associate Chair
Dr. Laurie Ford, Associate Chair
Dr. Daniel Salhani, Associate Chair
Dr. Anita Ho, Associate Chair

A.2 For the Laboratory Study

The University of British Columbia
Office of Research Services
Behavioural Research Ethics Board
Suite 102, 6190 Agronomy Road, Vancouver, B.C. V6T 1Z3

CERTIFICATE OF APPROVAL - MINIMAL RISK AMENDMENT

PRINCIPAL INVESTIGATOR:	DEPARTMENT:	UBC BREB NUMBER:
Gail C. Murphy	UBC/Science/Computer Science	H08-01848

INSTITUTION(S) WHERE RESEARCH WILL BE CARRIED OUT:

Institution	Site
UBC	Vancouver (excludes UBC Hospital)

Other locations where the research will be conducted:
N/A

CO-INVESTIGATOR(S):
Petcharat Viriyakattiyaporn

SPONSORING AGENCIES:
IBM Canada Ltd.
Natural Sciences and Engineering Research Council of Canada (NSERC)

PROJECT TITLE:
Evaluation of a Tool Recommendation System for a Software Development Environment

Expiry Date - Approval of an amendment does not change the expiry date on the current UBC BREB approval of this study. An application for renewal is required on or before: September 23, 2009

AMENDMENT(S): **AMENDMENT APPROVAL DATE: January 25, 2009**

Document Name	Version	Date
Consent Forms:		
Conset Form (PDF version of a web page)	1.7	January 10, 2009
Consent Form (PDF version of a web page)	1.8	January 16, 2009
Advertisements:		
recruitment email	1.5	January 13, 2009
recruitment poster	1.2	January 13, 2009
Questionnaire, Questionnaire Cover Letter, Tests:		
semi-structurad interview questions	1.0	January 10, 2009
preliminary questionnaire (PDF version of a web page)	1.6	January 10, 2009
Spyglass Tutorial	N/A	January 10, 2009
Training Tasks	1.0	January 10, 2009
Tasks	1.4	January 10, 2009
post-experiment questionnaire	1.6	January 13, 2009
Eclipse Tutorial	N/A	January 13, 2009
JFreeChart introduction	1.1	January 6, 2009

Other:
The web site will be used to sign the consent form (attached in Section 9.2) and to serve the preliminary questionnaire. The URL for the consent form will be http://stavanger.cs.ubc.ca:8080/spyglass/consent.htm The URL for the preliminary questionnaire will be http://stavanger.cs.ubc.ca:8080/spyglass/preliminary_questionnaire.htm

The amendment(s) and the document(s) listed above have been reviewed and the procedures were found to be acceptable on ethical grounds for research involving human subjects.

Approval is issued on behalf of the Behavioural Research Ethics Board

and signed electronically by one of the following:

Dr. M. Judith Lynam, Chair
Dr. Ken Craig, Chair
Dr. Jim Rupert, Associate Chair
Dr. Laurie Ford, Associate Chair
Dr. Daniel Salhani, Associate Chair
Dr. Anita Ho, Associate Chair

Appendix B

Additional Materials Related to the Longitudinal Study

B.1 Consent Form

THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Consent Form

Evaluation of a Tool Recommendation System for a Software Development Environment

Principal Investigator: Dr Gail Murphy, Dept of Computer Science (murphy@cs.ubc.ca).

Co-Investigator(s): Petcharat Viriyakattiyaporn, M.Sc. Student, Dept of Computer Science (pviriya@cs.ubc.ca). This study is part of Petcharat Viriyakattiyaporn's graduate thesis research, in which she is supervised by Dr Murphy.

Purpose:
The primary objective of this study is to determine whether the tools recommendation system that will be integrated to the Jazz integrated Software Development Environment (IDE) can help programmers navigate code more efficiently and can help the programmer discover useful tools in the IDE. This study is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a collaborative grant with IBM Canada Ltd.

Study Procedures:
As part of this study, you will be required to:

1. Complete an on-line questionnaire that will be displayed after you consent to the study that gathers information about your previous programming experience.
2. Download and install plugins for the recommendation system into an IBM Jazz 1.0 IDE running on your laptop or desktop. We expect installation will take approximately 30 minutes. If you have any trouble during installation, you will be provided with contact information for the co-investigator who can help you complete the installation process.
3. After the completion of the first code exploration lab, you will be prompted in the IBM Jazz 1.0 IDE to answer a short questionnaire about the IDE and submit the answers along with an automatically collected log of your use of the IDE. The answers to these questions and the log will be available only to the investigators associated with this study and will not be seen by personnel associated with CPSC 310.

4. After submission of the questionnaire and log data, the recommender will be enabled in your installation of the IBM Jazz 1.0 IDE and will begin making recommendations about tools that may be useful to you in the IDE. After completion of the second code exploration lab, you will again be prompted in the IBM Jazz 1.0 IDE to answer a questionnaire about the IDE and to submit an automatically collected log of your use of the IDE.
5. At the end of the course project, you will be prompted by email to send the automatically collected log of your exploration during the course project through a feature built into the IDE.
6. At the end of the course project, an interview will be scheduled between you and the co-investigator at a mutually convenient time in which you will be asked to complete a post-experiment questionnaire and answer questions about your experiences with the recommender.

At any time, you may choose to withdraw from the study by uninstalling the recommender. When you download the recommender, you will be provided instructions on how to uninstall the recommender.

Potential Risks:
The main risk during this study is the loss of your time that you spend on study activities.

Potential Benefits:
You will become more familiar with the target IDE and start to develop habits of working efficiently in the IDE in a short period of time.

Confidentiality:
All documents that have your identifiable information (e.g., your name, your username, and email address) will be kept in digital format on password protected computers that belong to the Department of Computer Science. We will never refer to you by your real name in any publication or report.

In particular, your participation/non-participation in the study will NOT be revealed to the course instructor or any personnel associated with CPSC310.

The log data we collect of your use of the IDE has no identifiable information and may be shared with other researchers.

Your interview that has been audio-taped will be transcribed by the principal investigator and the co-investigator only.

Compensation:
Upon completion of the entire study, you will receive a $25 gift certificate of your choice for Amazon.ca or iTunes. If you choose not to complete the entire study, but you do complete the second code exploration lab and submit a log for the study to that point, you will receive a $15 gift certificate of your choice for Amazon.ca or iTunes.

Contact for information about the study:
If you have any questions or desire further information with respect to this study, you may contact Dr Gail Murphy at +1 604 822-5169.

Contact for concerns about the rights of research subjects:
If you have any concerns about your treatment or rights as a research subject, you may contact the Research Subject Information Line in the UBC Office of Research Services at 604-822-8598.

Consent:
Your participation in this study is entirely voluntary and you may refuse to participate or withdraw from the study at any time without jeopardy to your class standing. Your decision to participate or not to participate in the study will NOT be revealed to the course instructor or any personnel associated to CPSC310.

Clicking 'Accept' button indicates that you consent to participate in this study. You will be able to print a copy of this form afterward.

Printed Name of the Subject

Accept | Decline

Version 1.4: September 11, 2008

B.2 Preliminary Questionnaire

THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Preliminary Questionnaire

Evaluation of a Tool Recommendation System for a Software Development Environment

Name: _____ Email: _____

UGrad username: _____ (E.g. a1b1) Lab Section: _____ (Format: L1[B/D/F/J])

Gender: ☐ M ☐ F

1. How many years of experience do you have in developing programs in Java?

   ☐ None ☐ Less than 1 year ☐ 1 - 5 years ☐ 6 - 10 years

2. How would you rate your expertise in Java?

   Novice Expert

   ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

3. How familiar are you with concepts of object-oriented programming in general?

   Unfamiliar Very Familiar

   ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

4. Do you have experience developing software outside of courses that you have taken at UBC or elsewhere?

   ☐ Yes ☐ No

5. Do you have any experience developing software that calls complex libraries or code that someone else wrote?

   ☐ Yes ☐ No

6. Do you use an IDE regularly (e.g., Visual Studio, Eclipse, NetBeans, etc.)?

   ☐ a. Yes. I use _____ regularly.

   ☐ b. No. I do not use IDE regularly.

7. Do you have previous experience with Eclipse and/or IBM Jazz/Rational Team Concert?

☐ Yes ☐ No

8. How would you rate your expertise in Eclipse?

Novice Expert

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

9. How would you rate your expertise in IBM Jazz?

Novice Expert

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Submit

Version 1.2: August 28, 2008

B.3 Post-Experiment Questionnaire

THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Post-Experiment Questionnaire

Evaluation of a Tool Recommendation System on a Software Development Environment

UGrad username: _____

1. Please circle the tools below that you started to use because of the recommendation system.
   a. Open Type Hierarchy
   b. Open Call Hierarchy
   c. The Links tab on Work Item Editor
   d. Team > Show History

2. How useful were the recommendations given by the recommendation system?

Very Useless		Neutral		Very Useful
1	2	3	4	5

3. Each recommendation has an associated rationale. Did you every look at the rationale for a recommendation?

   Yes No

4. If you answered Yes in Question #3, how often did you find that the rationale is accurate?

Never		Half the Time		Every time
1	2	3	4	5

5. The information presented in the recommendation view was easy to understand.

Strongly Disagree		Neutral		Strongly Agree
1	2	3	4	5

Version 1.3: August 28, 2008 1

6. Did you have the notification sound enabled when a recommendation was made?

   Yes No

7. If Yes, did the notification sound disturb you while you were performing your task?

Not At All		Neutral		Very Disturbing
1	2	3	4	5

8. Did you notice when the recommender icon changed?

   Yes No

9. The recommendation system is intended to remember your rating for each tool and never re-recommend a tool that is rated poorly. Did you know the recommendation system had this functionality?

   Yes No

10. How often did you look at new recommendations?

Never		Half the Time		Always
1	2	3	4	5

11. At this point, I feel that I am able to navigate source code efficiently.

Strongly Disagree		Neutral		Strongly Agree
1	2	3	4	5

12. At this point, I feel that I will be able to discover new and useful navigational functionalities provided in the IDE.

Strongly Disagree		Neutral		Strongly Agree
1	2	3	4	5

B.4 Spyglass Introduction

Spyglass Tutorial

Table of Contents

1. Configuring Spyglass
2. Status of Spyglass
3. Reading Recommendations
4. Enabling/Disabling Spyglass
5. Rating Recommended Tool
6. Submitting Recurring Questionnaire
7. Uploading Spyglass Data
8. Using Spyglass in a Lab

Spyglass stores your recommendation history and ratings for recommended tools in a database on a server. To enable Spyglass to communicate with this database, please make sure you are connected to the Internet while you are using Jazz. The following tutorial is written based on the assumption that you are connected to the internet while using Jazz.

1. Configuring Spyglass

To support storage of your recommendation history and ratings, Spyglass needs to know who you are. For every workspace/machine that you are using to work on your project, the first time you run Jazz on that workspace/machine, please configure Spyglass as follows:

- On the menu bar, select Window > Preferences. Once the Preferences dialog pops up, select **Spyglass**.
- Type your UGrad ID (i.e. the same one that you filled out in the preliminary questionnaire) in the text field provided and click 'OK'.

2. Status of Spyglass

Spyglass can be in different states, such as disabled, enabled (without new recommendations), and enabled with new recommendations. Each state is indicated with a different icon.

- (Disabled): After you have successfully installed Spyglass, you should see this icon in a bottom corner of your Jazz IDE. This indicates that Spyglass UI is disabled. You will not receive any notification about new tools recommendations.
- (Enabled): This icon indicates that Spyglass is enabled with no new recommendations at the moment. You will be able to enable/disable the Spyglass notification by using the drop-down arrow beside this icon.
- (Enabled with New Recommendations): This icon indicates that Spyglass is enabled and there is a new recommendation for you. Once you click on this icon, it will revert back to the Enabled state.

NOTE: For a period of time at the start of the study, Spyglass will remain disabled and you will not be able to enable it. Please do not worry. After a certain point, Spyglass will be enabled automatically, assuming that you are connected to the Internet while using Jazz, from which point,

you will have full control over it.

3. Reading Recommendations

At any time, you can click on the Spyglass icon at the bottom corner of Jazz IDe. Once you click on it, you will see an in-place view similar to the following picture. This view shows the name of a tool being recommended to you, the tool's keyboard shortcut, how many times it has been recommended (Frequency), and the last time it was recommended.

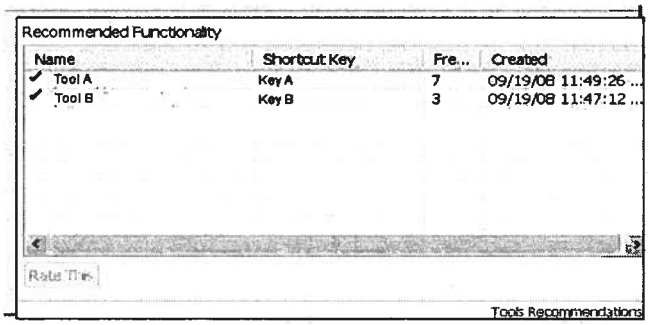


Figure 1: Recommendations In-Place View

If you select one of the recommendations in the in-place view, you will be able to see the rationale behind the recommendation. The rationale includes why the tool is recommended and a detailed description of the tool.

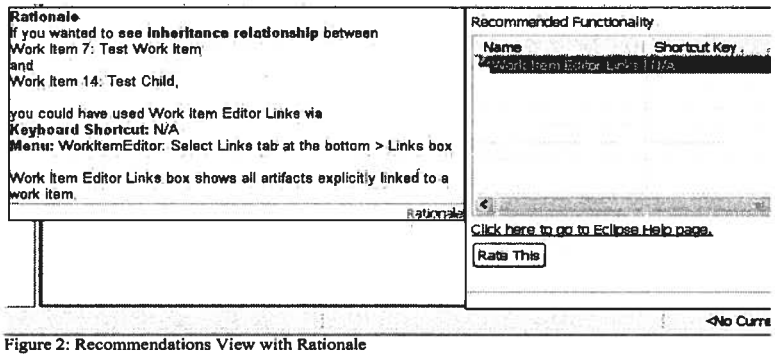


Figure 2: Recommendations View with Rationale

4. Enabling/Disabling Spyglass

Beside the Spyglass icon in a bottom corner of your Jazz IDE, you will also see a drop-down arrow. This arrow brings up a menu that allows you to enable or disable Spyglass notification.

5. Rating Recommended Tool

You are allowed to rate each tool exactly once. After you have rated it, Spyglass will remember your rating regarding that tool and will refer to it when it is making recommendations in the future. For example, if you rate Tool A as 'Somewhat Useless' or 'Very Useless', Spyglass will not recommend Tool A to you ever again. Thus, we recommend you to rate a tool after you have used tried using the tool for a while so that the rating is accurate.

To rate a tool, click on 'Rate This' button as shown in Figure 2. Then, you will see a popup dialog that asks you to give rating for the tool as shown in the figure below.

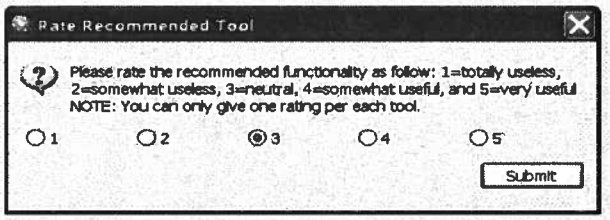


Figure 3: Rating Dialog Box

6. Submitting Recurring Questionnaire

As part of our study, you are asked to submit recurring questionnaire twice after each code exploration lab. It is a short questionnaire with only two questions. To submit this questionnaire, go to *Recommender Menu > Start Recurring Questionnaire* . Please make sure you are connected to the internet.

7. Uploading Spyglass Data

Aside from the recurring questionnaire, you are also asked to submit all the log data to our server, once you have submitted your recurring questionnaire. The log data includes all kinds of activities that you have performed on the IDE (e.g. select, edit, command, change perspective, change preference setting, etc) and also includes windows of activities used in our recommendation algorithm and your interaction activities with Spyglass (e.g. view recommendations, enable, and disable). To upload all the data, you simply have to click *Reommender Menu > Upload Data Files*. Again, please make sure you are connected to the internet.

It's best to try to minimize the number of workspaces that you use to work on your project. However, if you need to work on your project on more than one machine/workspace, then you are also asked to submit the log data from all those machines/workspaces after you have submitted a recurring questionnaire.

8. Using Spyglass in a Lab

Currently, Spyglass is installed on the Jazz RTC 1.0 client in the lab. In order for Spyglass to distinguish who is participating in the study, **you must configure Spyglass as in Step 1 above**. If you are not a participant of this study, please do not configure Spyglass and it will remain disabled and won't do anything to disturb you at all.

B.5 The First Code-Exploration Exercise

API Exploration Exercise 1

Introduction
The purpose of the API exploration exercises you will work on in this and the next lab are to help you learn about the APIs necessary to implement the Marking System for your project.

Setting Up the Project
The Marking System consists of two plugins: a component plugin that provides the student functionality (which we will refer to as the Student's plugin) and a component plugin that provides the marking functionality (which we will refer to as the Marker's plugin). You should already have these plugins in your workspace (they were created, or accepted from RTC in the Jazz Tutorial #2)

In each project, open the MANIFEST.MF file in the Manifest Editor. Open the Dependencies tab. Add the following as the base dependencies (you can add more dependencies later as needed) for both Marker's plugin and Student's plugin:
- org.eclipse.core.runtime (This provides support for the runtime platform, core utility methods and the extension registry)
- org.eclipse.core.resources (This provides basic support for managing a workspace and its resources)
- org.eclipse.ui (Application programming interfaces for interaction with and extension of the Eclipse Platform User Interface)

Adding dependencies allows you to explore the implementation of any Eclipse public API. This may be useful whenever the Javadoc is not provided or provided but not clear. It also means you can navigate to the code using the type hierarchy and other tools. This technique works for other libraries/API implementations that you add to the project provided that JARs of the added libraries and APIs include the source code.

If you now go to the Package Explorer view and expand your plugin project you will see a *Plug-in Dependencies* tree node under it. If you keep expanding each dependency, you will find some .class files. Try opening one of them—you should see the source code of that file. If you do not, that means the library does not have source code attached with the JAR files. You should try checking another library.

Exercise
This exercise will focus on the API necessary to implement the plugin. You should answer the following questions independently by using resources provided, including the Jazz Help Contents, the provided online articles, and the source code of any Eclipse library and example plugin. You may discuss the exercise and the questions with your classmates, but you must complete the entire exercise individually.

All questions regarding implementation of a method should be answered in plain language (unless stated otherwise) that describes what is done within the method. It should be more specific than Javadoc comments of that method.

You must hand in your answers to the questions at the beginning of your lab during the week of October 27 – 31.

Version 1.1: October 15, 2008 1

Part 1: Marking View in Marker's Plugin

To start, let's explore the API necessary to create the some of the UI of the Marking View in Marker's plugin (The Marking View in Student's plugin should be the same less the functionality to edit the marks). Based on the SRS, the marker should be able to open, create, or edit a marking scheme and export it. One way to allow this is to create a view that visualizes the marking scheme and allows the marker to edit some details (of course, you can also create a new editor specifically for a marking scheme, but it is not necessary). There is a guide on how to contribute a view in Eclipse in the Jazz Help Contents (Help > Help Contents):

Eclipse documentation > Platform plug-in developer guide > Programmer's Guide > Plugging into the workbench > Basic workbench extension points using actions > views

In the rest of this exercise, we will be referring to the Readme Tool example plugin that we used in the Eclipse Architecture lab. If you don't have this plugin in your workspace, please follow the instructions in the Eclipse Architecture lab to check it out from the Eclipse repository.

Please answer the following questions:

1. What Eclipse extension point do you need to extend in order to add a view to Eclipse?
2. What interface does one need to implement to create a view in Eclipse
3. ***ViewPart*** is an abstract class that implements the interface in question 2. What are the methods in ViewPart that a subclass must implement?
    a. What does the default implementation of *init(IViewSite)* in ViewPart do?
    Sample Answer: *init(IViewSite)* caches view site that is associated to this *ViewPart*. It also sets *<some fields>* if *<some condition>* holds.

    b. What does the default implementation of *dispose()* in ViewPart do?
    c. How does the implementation of *dispose()* in ***ReadmeSectionsView*** differ from the default implementation in ***ViewPart***?

    Hint: In this kind of question, you may consult the source code of Eclipse API. You should open up the ***ReadmeSectionsView*** in the Readme Tool plugin and start from there. Please recall that, in a Java Editor, if you select a class name or a method and right click on it, you will see a lot of functionality that you can use to navigate between classes and methods.

4. What is a Composite? (Composite in the SWT library, not the Composite design pattern that we saw in class)
5. What is a default layout for a Composite (please give the class name)?
6. What is the abstract superclass of all windowed user interface classes (e.g. Composite, Button, Text, etc)?
7. Name at least 3 classes (that are not in question 6) that you are planning to use to create widgets on the Marking View. Also, state what they are used for.
8. What widgets are used to create the ReadmeSectionsView?
9. What extension points do you need to extend in order to add a menu on a view's toolbar?
10. What interface do you need to implement if you want to contribute a menu to the view's toolbar?
11. ***ViewActionDelegate*** in the Readme Tool plugin implements that interface. How does it handle a selection change event? Explain in plain language and be more specific than the Javadoc comments.

Part 2: Preference Page
The Marker's plugin should allow a marker to save his/her name and email address as a system preference. In Eclipse, preferences are gathered via preference pages. To gather the desired information, we should create a preference page for the Marker's plugin. There is a guide in how to contribute and implement a preference page in Eclipse Help Contents:
Eclipse documentation > Platform plug-in developer guide > Programmer's Guide > Plugging into the workbench > Preference pages

1. What Eclipse extension point do you need to extend in order to add a preference page to Eclipse?
2. What interface does one need to implement in order to create a preference page for Eclipse?
3. What is the superclass of ***ReadmePreferencePage***?
4. What does the default implementation of *doGetPreferenceStore()* in the superclass of ***ReadmePreferencePage*** do?
5. What does the default implementation of *performOK()*, *performApply()*, and *performCancel()*in the superclass of ***ReadmePreferencePage*** do?
6. What abstract class does a plugin that contributes a preference page have to extend?
    a. What is the importance of method ***start(BundleContext)*** to a plugin (Hint: For any type of plugin)?
    b. What is the importance of method ***stop(BundleContext)*** to a plugin?

Part 3: Popup Menu and Wizard
One functional requirement for the Marking plugin states that a marker must be able to highlight some part of the code and add comments about that chunk of code to a particular section in a marking report. One way to do this is using a Wizard that associates the selected chunk of code with comments and stores it under a marking section, for future retrieval.
These articles may help:
- Selection Service in Eclipse:
    <http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html>.
- How to create a JFace Wizard:
    <http://www.eclipse.org/articles/article.php?file=Article-JFaceWizards/index.html>. You should download the sample plugin provided on this website as well (as a .zip file).

Then, in Jazz, right click on the Package Explorer > Import > Existing Projects into Workspace. In "Select archive file", browse for the .zip file that you just downloaded. Select the project and click Finish.

1. What extension point do you need to extend in order to add a menu on a popup menu when a marker selects some text in a file?
2. What interface do you need to implement if you want to contribute such a menu?
3. From the interface that you answered in 2, what method can we use to obtain the view/editor being selected? How?
4. How do we obtain the highlighted chunk of code? Write a code snippet to do this or explain the idea in pseudo code based on existing API methods (Hint: How do you obtain an ISelection object?).
5. A wizard needs to be contained in a container, defined by IWizardContainer. WizardDialog is an abstract class that implements that interface. How does WizardDialog implement the ***updateButtons()*** method? (Hint: Look in ***HolidayAction#run(IAction)*** in the *com.xyz.article.wizards* plugin).
6. When the Next button on a wizard dialog is pressed, what methods are called?

7. To create a wizard, one needs to implement IWizard and IWizardPage and add the IWizard instance into the container. ***Wizard*** and ***WizardPage*** are abstract classes that implement those interfaces. Answer the following in plain language.
   a. How is ***createPageControl(Composite)*** implemented in ***Wizard***?
   b. How is ***performFinish()*** implemented in ***Wizard***?
   c. How is ***canFinish()*** implemented in ***Wizard***?
   d. How is ***canFlipToNextPage()*** implemented in ***WizardPage***?
   e. How is ***getNextPage()*** implemented in ***WizardPage***?
   f. What is the default value returned from ***isPageComplete()*** in ***WizardPage***?
8. In **HolidayMainPage** in *com.xyz.article.wizards*, how is ***canFlipToNextPage()*** implemented?
9. In **HolidayWizard** in *com.xyz.article.wizards*, how is ***performFinish()*** implemented?

B.6 The Second Code-Exploration Exercise

API Exploration Exercise 2

Introduction
The purpose of the API exploration exercise you will work on in this lab is to help you learn more about the APIs necessary to implement the Marking System for your project.

Setting Up the Project
We are assuming that you have already created two plugin projects for the Marking system during the last lab. In this exercise, aside from the Eclipse API, we will explore the JFreeChart library, which you can use to create graphs to show the grade statistics. You can download the JARs for JFreeChart and its dependency (JCommon) from the 310 WebCT-Vista site.

Copy and paste *JFreeChart1.0.11.jar* and *JCommon1.0.14.jar* under your plugin project directory in your workspace (the top-level directory of the plugin project).

To include the JFreeChart and JCommon JARs in your plugin's runtime build path, please do the following:

1. Open up MANIFEST.MF in the Manifest Editor.
2. Open the **Runtime** tab and select the **Classpath** box. Click Add.
3. In the popup dialog, you should see your JFreeChart1.0.11.jar and JCommon1.0.14.jar. Select both and click OK. If you don't see the JARs, close the dialog. Right click on your project directory and select **Refresh**. Start from step 1 again.
4. Open the **Build** tab. Make sure that the two JAR files are selected in the **Binary Build** box.
5. Save the changes in MANIFEST.MF.
6. Copy **First.java** that is provided on WebCT to your plugin's src directory. If it is compiled without any error, that means you have configured your Build Path correctly.

Also, to import the sample project *cs310.lab.examples* into your workspace, you can do the following:

1. Right click in Package Explorer and select **Import**.
2. Under **General**, select **Existing Projects into Workspace**.
3. In the **Select archive file** section, use Browse to select *cs310.lab.examples.zip*. Then, click **Finish**.

Exercise
This exercise will focus on the API necessary to implement the Marking System. You should answer the following questions independently using resources provided, including the Jazz Help Contents, the provided online articles, the source code of any Eclipse library and example plugin. You may discuss the exercise and the questions with your classmates, but you must complete the entire exercise individually.

All questions regarding implementation of a method should be answered in plain language (unless stated otherwise) that describes what is done within the method. It should be more specific than Javadoc comments of that method.

You must hand in your answers to the questions at the beginning of your lab during the week of November 3rd – 7th, 2008.

Version 1.1: October 22, 2008 1

Part 1: CompilationUnitEditor
In the last code exploration exercise, we learned roughly how to get the selected text in a Java source file and associate it with a comment and marking report using a wizard. Now, we will continue to learn how to open up a source file in an editor and highlight a certain part of the file given a position in that file and a length of the highlighted text.

First, you need to know that the default editor for a Java source file is called the CompilationUnitEditor and a Java source file is called a CompilationUnit. There is an article that explains how one can open a file in an editor programmatically in Eclipse: http://wiki.eclipse.org/FAQ_How_do_I_open_an_editor_programmatically%3F

Please refer to the class **FileEditorSample** in the *cs310.lab.examples* plugin to answer the following questions:

1. Please explain in plain language what an **IWorkbenchPage** instance is.
2. In the method ***openFileInEditor(String)***, we retrieve an IFile object from the given file path using the method ***getFile(IPath)*** in the interface **IWorkspaceRoot**.
    a. What interface(s) does **IWorkspaceRoot** directly extend?
    b. What is the format of the path passed in to ***getFile*** as its parameter? Suppose the workspace is located at "C:\workspace". If we pass in the path "C:\Desktop\readme.txt", how would it be interpreted by ***getFile***? Will the IFile object returned from ***getFile*** represent the same file that we want?
3. You may notice that the static global variable *currentEditor* is initialized to null. How would you modify ***openFileInEditor(String)*** so that we can remember the editor that our file is opened in as *currentEditor*?
4. ***highlightText(int, int)*** is supposed to highlight a chunk of text in a Java file (represented by the **CompilationUnit** class) that is opened in *currentEditor*. Currently, it does not do anything. Fill in the missing code that makes ***highlightText*** do what it is supposed to do under the //TODO comments (Hint: Look in **CompilationUnitEditor** for all methods, including the inherited ones).

Part 2: Statistics View using JFreeChart
There is a requirement that a marker must be able to view statistics (mean, mode, median, min, and max) for all marked assignments. We suggest that you use JFreeChart library to create graphics for these statistics. With respect to this requirement, you are free to choose the type of chart(s) that you want to show. JFreeChart also provides some demos in its JAR file as well. To see the demos, go to the Package Explorer view. Assuming that you have added the JFreeChart and JCommon libraries to your build path as instructed above, you should be able to find **JFreeChart1.0.11.jar** and **JCommon1.0.14.jar** under the node *Referenced Libraries* in your plugin project.

Expand **JFreeChart1.0.11.jar** to find the package ***org.jfree.chart.demo***. There are three demos: *BarChartDemo1, PieChartDemo1, and TimeSeriesDemo1*. You can run each of them as a Java Application to see the charts.

Please answer the following questions:

1. In **BarChartDemo1.java**, what method of **ChartFactory** gets called to create the bar chart?
2. **CategoryDataset** is the interface that is used to create a bar chart.

   a. Please list the names of existing concrete classes that **directly** implement this interface.
   b. Please list the names of the interfaces that **directly** extend this interface.
   c. What does a row key in **CategoryDataset** represent in a bar chart?
   d. What does a column key in **CategoryDataset** represent in a bar chart?
3. In **BarChartDemo1**, **DefaultCategoryDataset** is used to create a dataset for the bar chart. There are methods called ***addValue(double, Comparable, Comparable)*** and ***setValue(double, Comparable, Comparable)*** in **DefaultCategoryDataset**. How do these two methods differ?
4. **ChartFactory** provides many utility methods that help create some standard charts with default appearance. These methods have the name pattern ***create<some chart name>***, and they all return the same type of object, which is **JFreeChart**.
   a. How many public constructors are there in **JFreeChart**?
   b. For all constructors in **JFreeChart**, what is/are the parameter(s) that must be passed in?
   c. A **JFreeChart** registers itself as a listener to a plot change event (see a constructor). What is the method that handles a plot change event in **JFreeChart**? What is its default implementation?
5. Back to **ChartFactory**. Please look in the method **createBarChart**.
   a. What type of plot is used to create the returned **JFreeChart**?
   b. What type of renderer is used to create the plot?
   c. What does **createBarChart** do with the dataset that is passed in as an argument?
6. What class is responsible of drawing axes and data for a **JFreeChart** (Hint: Look in ***draw(Graphics2D, Rectangle2D, Point2D, ChartRenderingInfo)*** in **JFreeChart**)?
7. In **CategoryPlot**, what method gets called to map a particular dataset to an axis (Hint: Look in the CategoryPlot constructor)?
   a. What does this imply about the number of datasets that a **CategoryPlot** can manage?
   b. What does this imply about the number of axes a **CategoryPlot** can manage?
8. There is also a requirement that students must be able to view a histogram displaying all the grades for their assignments. You may want to review what a histogram is (including the terms used in histogram) by reading this article:
   http://www.nctmba.com/statistics/histogram/
   a. What method in **ChartFactory** should be called to create a histogram?
   b. What concrete class should be used to create a histogram dataset (pick one)?
   c. Suppose you are given this raw observation data.
      {1, 1, 1, 1, 2, 2, 3, 4, 4, 4, 5}

      Write a code snippet that shows how to create and initialize the dataset (only the dataset!) for a histogram, where **Index** appears on the **X** axis and **Frequency** appears on the Y axis, with the series name 'Series1'.

Index	Frequency
1	4
2	2
3	1
4	3
5	1

Part 3: Perspective

A non-functional requirement states that the functionality for marking system should be broken into two perspectives: Student Perspective and Marker Perspective. A perspective is basically a collection of views. There is a guide describing how to add a perspective in Eclipse on the Help Contents (under menu Help > Help Contents):

Eclipse documentation > Platform plug-in developer guide > Programmer's Guide > Advanced workbench concepts > Perspectives

You may also refer to the *myperspective* package in the *cs310.lab.examples* plugin, for an example of how to add a new perspective to Eclipse.. You can also try launching it as an Eclipse Application to see the layout in this perspective. The perspective's name that is shown in the menu Window > Open Perspective is *Example Perspective*.

Please answer the following questions:

1. What extension point do you need to extend in order to add a new perspective to Eclipse?
2. What interface does one need to implement to create a perspective in Eclipse?
3. In the method ***createInitialLayout(IPageLayout)*** in **MyPerspectiveFactory**, it defines the "top left" folder layout to contain the Navigation view and the Bookmarks view. How wide is this folder layout, relative to the width of the current editor area in the Example Perspective (please answer as a percentage)? Which statement defines this?
4. In the same method, what is the height of the "bottom left" folder layout, relative to the "top left" folder layout (please answer as a percentage)? Which statement defines this?
5. Please list all the views whose ID's are provided as constants in **IPageLayout**.
6. What is the String constant that is the ID of the *Problem View*?
7. What method in **IPageLayout** will put the name of your new view in the menu *Window > Show View* so that you can open your view without going to *Window > Show View > Other*?
8. What is a placeholder of a view in a perspective?

B.7 Recurring Questionnaire

THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Recurring Questionnaire

Evaluation of a Tool Recommendation System for a Software Development Environment

UGrad username:

1. At this point, I feel that I am able to navigate source code efficiently.

Strongly Disagree		Neutral		Strongly Agree
☐ 1	☐ 2	☐ 3	☐ 4	☐ 5

2. At this point, I feel that I will be able to discover new and useful navigational functionalities provided in the IDE.

Strongly Disagree		Neutral		Strongly Agree
☐ 1	☐ 2	☐ 3	☐ 4	☐ 5

Submit

Version 1.1: August 28, 2008

[1] **NOTE**: This questionnaire will only be given to the participating students at the beginning of code exploration phase of their project (before the recommender is enabled) and at the end of the code exploration phase (after the recommender is enabled). It will be submitted through the IDE interface.

B.8 Interview Protocol

Spyglass Recommendation System Interview Protocol

Background Related Questions
1. Can you describe your experience of Java development (i.e. only in classes, co-op, etc)?
2. Do you usually read Help document or manuals that come with your software? If not, how do you learn about functionality of your new software?
3. Do you read Help document/Tutorial of RTC? Is it easy to find the answers you are looking for?

Tasks Related Questions
1. Did you complete the 1st and 2nd code exploration exercises? How do you think about them?
2. How did you complete them? What resources did you consult?
3. If you did not look at the source code that much, can you explain the reasons why?
4. What part of the project were you responsible for?
5. How would you rate the difficulty of your project? Please also provide reasons.

Spyglass Usage Questions
1. You said you never look at new recommendations even though you noticed the icon change, could you explain why?
2. How far apart each notification should be so that it does not annoy you?
3. How do you think about the idea of having a recommendation system on the IDE?

User Interface Related Questions
1. Based on Q1 in the questionnaire[1], you have used tool <X> because of the recommendation system. After you have uninstalled the recommendation system, do you still continue using it? Do you find it helpful?
2. From your log data, we have observed that you disabled the recommendation system very soon after you use it. Could you tell us why?
   OR
   From your log data, we have observed that you disabled the recommendation system once/twice/etc and enabled it again. Can you tell us why?
3. Based on Q2 in the questionnaire, you said that the recommendations were useless. Is it because you have known the recommended tools, already?
4. Based on Q3, you never looked at the rationale associated with a recommended tool. Why is that?
5. Can you elaborate on Q12 in the questionnaire a little bit? Why do you feel that you will be able to or will not be able to discover new and useful navigational functionalities provided in the IDE?
6. *Based on the short questionnaire that you filled out over time, it seems that your confidence has decreased over time. Can you explain why?
7. Do you have any suggestion about design of the recommendation system? Should we change the notification mechanism?
8. Are you aware that there are a limited number of tools that the system can recommend?
9. If you think this system is a good idea but it still needs improvement, can you suggest what we should improve?
10. Is there any other questions or comments?

[1] We are referring to the post-experiment questionnaire.

Appendix C

Additional Materials Related to the Laboratory Study

C.1 Consent Form

THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Consent Form

Evaluation of a Tool Recommendation System for a Software Development Environment

Principal Investigator: Dr Gail Murphy, Dept of Computer Science (murphy@cs.ubc.ca).

Co-investigator(s): Petcharat Viriyakattiyaporn, M.Sc. Student, Dept of Computer Science (pviriya@cs.ubc.ca). This study is part of Petcharat Viriyakattiyaporn's graduate thesis research, in which she is supervised by Dr Murphy.

Purpose:
The primary objective of this study is to determine whether the tools recommendation system that will be integrated to the Rational Team Concert integrated Software Development Environment (IDE) can help programmers navigate code more efficiently and can help the programmer discover useful tools in the IDE. This study is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a collaborative grant with IBM Canada Ltd.

Inclusion Criteria:
Before you proceed, please make sure that you meet the following criteria:
1. Have at least 8 months of experience with programming using Java.
2. Have at least 2 months of experience in using Eclipse to develop Java program.
3. Have never contributed to JFreeChart open-source project.

Study Procedures:
As part of this study, you will be required to:
1. Complete an on-line questionnaire that will be sent to you one day after you consent to the study that gathers information about your previous programming experience.
2. We will use the answers to the questionnaire to determine whether you can proceed to the next step in this study or not. If so, an email with an introduction to JFreeChart (http://www.jfree.org) open source library will be sent to you to your email address that is given in the questionnaire.

Otherwise, you will be asked to withdraw from the study. Please spend some time reading this introduction so that you become familiar with the JFreeChart architecture.

3. Also, after submitting the preliminary questionnaire, we will make an appointment with you to come in to our lab at your convenient time to perform some programming tasks related to JFreeChart.
4. Before you start doing the tasks, we will briefly review the basic functionalities of the IDE and JFreeChart architecture for you, and how to use the recommendation system. We will also ask you to complete two simple training tasks. These will take around 20 minutes.
5. You will be given two hours to complete two programming tasks (one hour per task) in order, with five minutes break in between. While you are performing the tasks, we will use screen-capturing software to record your browsing history. You are encouraged to speak out loud what you are thinking.
6. The co-investigator will be observing you quietly. You may ignore her.
7. After finishing the tasks or two hours have passed, you will be asked to fill out a questionnaire assessing your opinion regarding the tasks and the recommendation system.
8. Lastly, you will have a 15-minute semi-structured interview with the co-investigator.

At any time, you may choose to withdraw from the study by notifying the co-investigator.

Potential Risks:
The main risk during this study is the loss of your time that you spend on study activities.

Potential Benefits:
You will become more familiar with the target IDE and start to develop habits of working efficiently in the IDE in a short period of time.

Confidentiality:
All documents that have your identifiable information (e.g., your name, your username, and email address) will be kept in digital format on password protected computers that belong to the Department of Computer Science. We will never refer to you by your real name in any publication or report.

The log data we collect of your use of the IDE has no identifiable information and may be shared with other researchers.

Your interview and browsing history that have been recorded will be transcribed by the principal investigator and the co-investigator only.

Compensation:

If you are allowed to proceed after the preliminary questionnaire, you will receive a $10 certificate per hour, for Amazon or iTunes of your choice as compensation. Upon completion of the entire study, you will receive up to $30 gift certificate in total.

Contact for information about the study:
If you have any questions or desire further information with respect to this study, you may contact Dr Gail Murphy at +1 604 822-5169.

Contact for concerns about the rights of research subjects:
If you have any concerns about your treatment or rights as a research subject, you may contact the Research Subject Information Line in the UBC Office of Research Services at 604-822-8598.

Consent:
Your participation in this study is entirely voluntary and you may refuse to participate or withdraw from the study at any time.

Clicking 'Accept' button indicates that you consent to participate in this study. You will be able to print a copy of this form afterward.

Printed Name of the Subject

Accept | Decline

Version 1.8: January 16, 2009

C.2 Preliminary Questionnaire

THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Preliminary Questionnaire

Evaluation of a Tool Recommendation System for a Software Development Environment

Name: _____ Email: _____

Gender: ☐ M ☐ F

1. How many months of experience do you have in developing programs in Java?

   ☐ no greater than 3 ☐ 4-7 ☐ 8-11 ☐ 12-15 ☐ more than 15

2. How big is the biggest Java program that you have developed (in term of number of classes)?

   ☐ up to 25 ☐ 25-50 ☐ 50-75 ☐ 75-100 ☐ more than 100

3. Have you ever contributed to JFreeChart open source project (http://www.jfree.org/jfreechart/)?

   ☐ Yes ☐ No

4. Do you have any experience developing software that calls complex libraries or code that someone else wrote?

   ☐ Yes ☐ No

5. Do you have previous experience with Eclipse?

   ☐ Yes ☐ No

6. If Yes, how many months of experience do you have in using Eclipse?

   ☐ no greater than 3 ☐ 4-7 ☐ 8-11 ☐ 12-15 ☐ more than 15

7. Please select all the following tools in Eclipse that you **have used** before OR know exactly what they do.

   - ☐ Open Declaration (F3 or Ctrl+Left Click)
   - ☐ Open Call Hierarchy (Ctrl+Alt+H)

- ☐ Text Search (Ctrl+H)
- ☐ Open Type (Ctrl+Shift+T)
- ☐ Open Type Hierarchy (F4)
- ☐ Debugger
- ☐ All of the above

Submit

Version 1.7: January 26, 2009

C.3 Post-Experiment Questionnaire for the Spyglass Group

THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Post-Experiment Questionnaire

Evaluation of a Tool Recommendation System on a Software Development Environment

Participant's Name: _____

NASA Task Load Index: The following questions will assess the difficulty of the given tasks, in the scale from 1 to 21. Please type an X in the bucket that best reflects your experience.

Task 1: Fix Pie Chart Labels

Mental Demand — How mentally demanding was the task?

very low ——— very high

Temporal Demand — How hurried or rush was the pace of the task?

very low ——— very high

Performance — How successful were you in accomplishing what you were asked to do?

perfect ——— failure

Effort — How hard did you have to work to accomplish your level of performance?

very low ——— very high

Frustration — How insecure, discouraged, irritated, stressed, and annoyed were you?

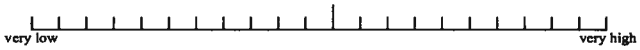
very low ——— very high

Version 1.6: January 13, 2009 1

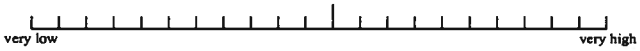
C.3. Post-Experiment Questionnaire for the Spyglass Group

Task 2: Add Bar Chart Axes Labels

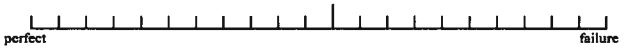
Mental Demand — How mentally demanding was the task?



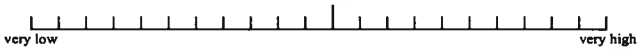
Temporal Demand — How hurried or rush was the pace of the task?



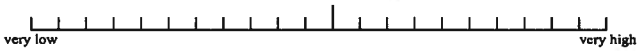
Performance — How successful were you in accomplishing what you were asked to do?



Effort — How hard did you have to work to accomplish your level of performance?



Frustration — How insecure, discouraged, irritated, stressed, and annoyed were you?



IDE Usage Experience: The following questions will assess your experience from using the IDE to complete the assigned tasks.

1. Please underline the tools below that you used and rate their usefulness in the scale of 1-5 (1 is very useless and 5 is very useful).
   a. Open Type Hierarchy 1 2 3 4 5
   b. Open Call Hierarchy 1 2 3 4 5
2. How often did you click at the recommendation system icon (the wrench/light bulb icon)?

Never		Half the Time		Always
1	2	3	4	5

   If you answered Never, please skip to Question 7.
3. How useful were the recommendations given by the recommendation system?

Very Useless		Neutral		Very Useful
1	2	3	4	5

4. Each recommendation has an associated rationale/explanation. Did you ever look at the rationale for a recommendation?

   Yes No

5. If you answered Yes in Question 4, how useful are the rationales?

Very Useless		Neutral		Very Useful
1	2	3	4	5

6. The information presented in the popup notification was easy to understand.

Strongly Disagree		Neutral		Strongly Agree
1	2	3	4	5

7. Did the popup disturb you while you were performing your task?

Not At All		Neutral		Very Disturbing
1	2	3	4	5

8. Did the notification sound disturb you while you were performing your task?

Not At All		Neutral		Very Disturbing
1	2	3	4	5

9. Did you notice when the recommender icon changed (from a wrench to a light bulb)?

   Yes No

C.4 Post-Experiment Questionnaire for the Tutorial Group

THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Post-Experiment Questionnaire

Evaluation of a Tool Recommendation System on a Software Development Environment

Participant's Name: _____

NASA Task Load Index: The following questions will assess the difficulty of the given tasks, in the scale from 1 to 21. Please type an X in the bucket that best reflects your experience.

Task 1: Fix Pie Chart Labels

Mental Demand — How mentally demanding was the task?

very low ——— very high

Temporal Demand — How hurried or rush was the pace of the task?

very low ——— very high

Performance — How successful were you in accomplishing what you were asked to do?

perfect ——— failure

Effort — How hard did you have to work to accomplish your level of performance?

very low ——— very high

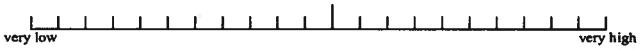
Frustration — How insecure, discouraged, irritated, stressed, and annoyed were you?

very low ——— very high

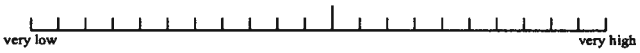
Version 1.6: January 13, 2009 1

Task 2: Add Bar Chart Axes Labels

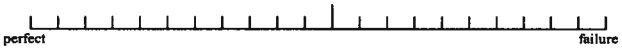
Mental Demand — How mentally demanding was the task?



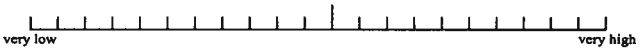
Temporal Demand — How hurried or rush was the pace of the task?



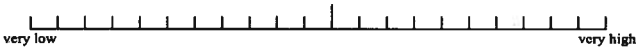
Performance — How successful were you in accomplishing what you were asked to do?



Effort — How hard did you have to work to accomplish your level of performance?



Frustration — How insecure, discouraged, irritated, stressed, and annoyed were you?



IDE Usage Experience: The following questions will assess your experience from using the IDE to complete the assigned tasks.

1. Please underline the tools below that you used and rate their usefulness in the scale of 1-5 (1 is very useless and 5 is very useful).

a. Open Type Hierarchy	1	2	3	4	5
b. Open Call Hierarchy	1	2	3	4	5

C.5 JFreeChart Introduction Document

Introduction to JFreeChart

JFreeChart is an open-source Java library that can be used to draw various types of charts, including bar charts, pie charts, XY charts, box plots, etc.

A Brief Introduction to JFreeChart
`JFreeChart` (JFreeChart1.0.11/source/org.jfre./chart/JFreeChart.java) is the key class of the JFreeChart1.0.11 library that is responsible for drawing the entire chart.

When a chart is drawn on a Java2D graphics device, the `JFreeChart#draw` method is executed. The `JFreeChart` class uses a number of other classes to achieve the drawing: instances of the `Title` class (one instance typically represents a legend), the `Plot` class and the `org.jfree.data.general.Dataset` class (the plot in turn manages a domain axis and a range axis).

Terminology
JFreeChart uses specific terminology for particular parts of a chart. These terms are labeled on the sample chart below.

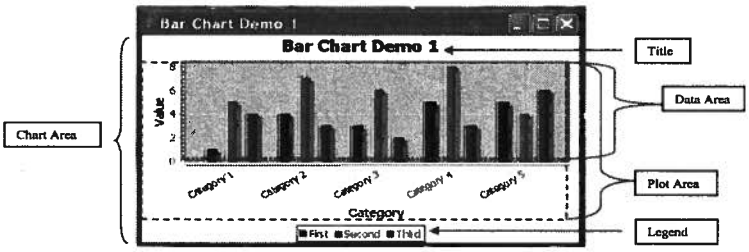


Figure 1: Bar Chart components illustration

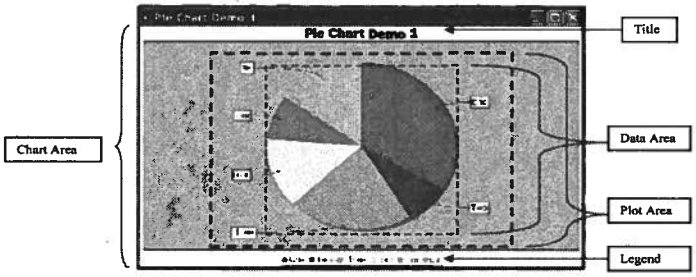
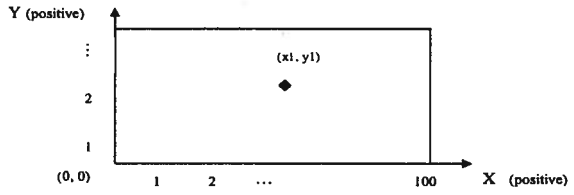


Figure 2: Pie Chart components illustration

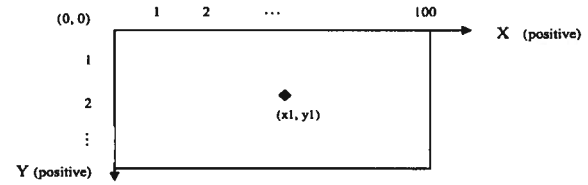
1. *Chart Area*: This term refers to the entire chart. You can think of it as the canvas on which a plot is drawn.
2. *Plot*: This refers to the visualization of data, axes, and axes labels. A Plot looks different depending on the type of chart it is drawn in. For example, a bar chart plot visualizes data in categories and each category is divided into series, each series is drawn as a bar in an XY plane. On the other hand, a pie chart plot visualizes data as sections in a pie.
3. *Plot Area*: This is the area in which data, axes, and axes labels reside.
4. *Data Area*: This is a sub-area of a plot area in which only data is rendered.
5. *Axes*: In some kinds of charts whose data area is rectangular (e.g., a bar chart, box plot, XY plot), the data area is surrounded by axes. An axis is a straight line on a border of a rectangular plot. Usually, there are labels along an axis to mark some values of data.
6. *Insets*: Insets represent inner margins of a container. It specifies the space that the container must leave at each of its edge. The space can be a border, a blank space, or a title.

Coordinates

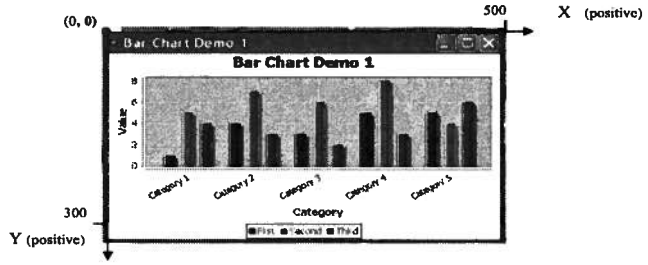
A chart is considered a 2D digital image. When we draw a 2D digital image, we are drawing it on a grid. Each cell of the grid is called a "pixel". A 2D digital image is made of multiple pixels filled with different colors. One needs to know the coordinate system of the grid in order to be able to specify the position where each graphic component is drawn. In geometry, you might be familiar with the coordinate system that looks like this:



However, by convention, the coordinate system of a 2D digital image looks like this:



In other words, in 2D digital image, we only have positive coordinates, and the origin is at the top-left corner of the grid. The coordinate of each pixel in a chart drawn by JFreeChart is relative to the window it is drawn on, as shown in the image below.



C.6 Eclipse-Based Tools Tutorial for the Spyglass Group

Brief Eclipse Tutorial

Table of Contents

1. Open Declaration
2. Search References in Workspace
3. Javadoc View
4. File Search

Eclipse is an integrated development environment that includes a number of tools (accessed through commands and menus) to help you develop source code. In this experiment, you will likely have to navigate the source code we provide for you to work on. This tutorial introduces some tools in Eclipse that are meant to help you navigate the code more efficiently. Note that this is a selection of tools in Eclipse.

1. **Open Declaration**

   If you want to know where a class, method or field is declared, you can use the Open Declaration tool. This tool opens up a source file in which the requested method/class is "declared" (not necessary where it is implemented).

   There are many ways that you can invoke Open Declaration.
   Keyboard shortcut: Click on a class/method/field name > F3
   Menu: On the Menu bar, select Navigate > Open Declaration
   Context menu: Right click on a class/method/field name > Open Declaration

2. **Search References in Workspace**

   If you want to know what parts of the source code reference or are referenced by a particular method, you can use the References tool. This tool searches for all the methods in the current workspace, in which the selected method is referenced.

   Followings are ways that you can search for references in the workspace.
   Keyboard shortcut: Click on a method name > Ctrl+Shift+G
   Menu: On the Menu bar, select Search > References > Workspace
   Context menu: Right click on a field/method/class name > References > Workspace

3. **Javadoc View**

   You can read the Javadoc comments of the program in the Javadoc view. This view renders comments of the selected class/method in the Javadoc style, if the comments are considered Javadoc comments.

   Followings are ways that you can open the Javadoc view.
   Keyboard shortcut: Click on a class/method/field name > Alt+Shift+Q, J
   Menu: On the Menu bar, select Window > Show View > Javadoc

Figure 3 shows Javadoc comments of *ActionListener* interface.

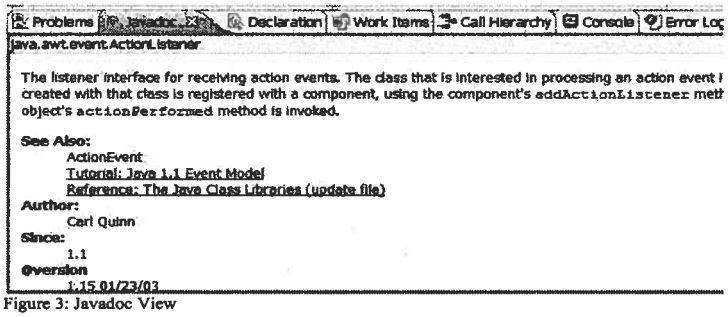


Figure 3: Javadoc View

4. **File Search**

If you want to search for all files that contain certain keywords, you can use this global file search. You can modify the filter to search among files with certain extensions or to limit the search space to within workspace or a project.

Followings are ways that you can invoke file search.
Keyboard shortcut: Click on a class/method/field name > Ctrl+H > Select File Search
Menu: On the Menu bar, select Search > Search... > Select File Search

Figure 4 shows the File search dialog that will search for all XML files within the current workspace that contain the keyword "tutorialURL".

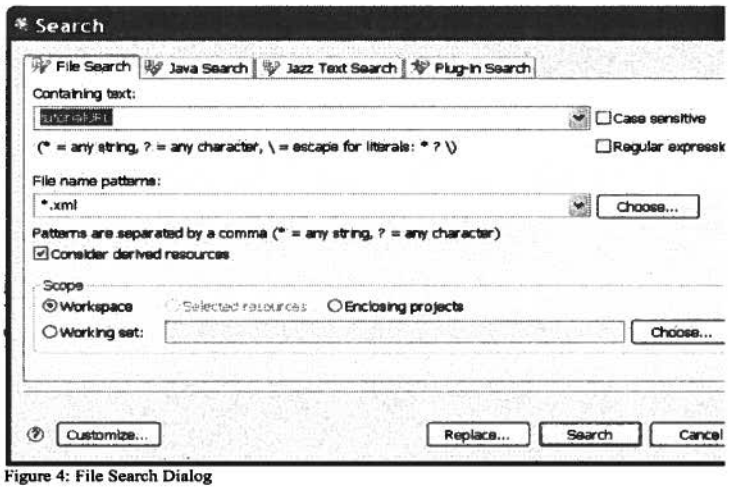


Figure 4: File Search Dialog

C.7 Eclipse-Based Tools Tutorial for the Tutorial Group

Brief Eclipse Tutorial

Table of Contents

1. Open Declaration
2. Open Call Hierarchy
3. Search References in Workspace
4. Open Type Hierarchy
5. Javadoc View
6. File Search

Eclipse is an integrated development environment that includes a number of tools (accessed through commands and menus) to help you develop source code. In this experiment, you will likely have to navigate the source code we provide for you to work on. This tutorial introduces some tools in Eclipse that are meant to help you navigate the code more efficiently. Note that this is a selection of tools in Eclipse.

1. **Open Declaration**

   If you want to know where a class, method or field is declared, you can use the Open Declaration tool. This tool opens up a source file in which the requested method/class is "declared" (not necessary where it is implemented).

   There are many ways that you can invoke Open Declaration.
   Keyboard shortcut: Click on a class/method/field name > F3
   Menu: On the Menu bar, select Navigate > Open Declaration
   Context menu: Right click on a class/method/field name > Open Declaration

2. **Open Call Hierarchy**

   If you want to know how a method is called within the system, you can use the Open Call Hierarchy tool. For a selected method (e.g., in the Package Explorer View, in the Java Editor, in the Outline View, etc.), this tool displays that method's callers and callees in the Call Hierarchy view.

   Followings are ways that you can invoke Open Call Hierarchy.
   Keyboard shortcut: Click on a method name > Ctrl+Alt+H
   Menu: On the Menu bar, select Navigate > Open Call Hierarchy
   Context menu: Right click on a method name > Open Call Hierarchy

   Figure 1 shows all the methods that call method *actionPerformed* within the workspace. To view all the methods that get called by *actionPerformed*, you can toggle the button labeled by A in the figure.

Figure 1: Open Call Hierarchy View

3. **Search References in Workspace**

   If you want to know what parts of the source code reference or are referenced by a particular method, you can use the References tool. This tool searches for all the methods in the current workspace, in which the selected method is referenced.

   Followings are ways that you can search for references in the workspace.
   Keyboard shortcut: Click on a method name > Ctrl+Shift+G
   Menu: On the Menu bar, select Search > References > Workspace
   Context menu: Right click on a field/method/class name > References > Workspace

4. **Open Type Hierarchy**

   If you want to know where a type (e.g. class/interface/method) exists within the hierarchy of types in the system, you can use the Open Type Hierarchy tool. This tool opens up the selected type in the Type Hierarchy view that shows its superclasses, subclasses, implemented interfaces, and all inherited methods.

   Followings are ways that you can invoke Open Type Hierarchy.
   Keyboard shortcut: Click on a class/interface/method name > F4
   Menu: On the Menu bar, select Navigate > Open Type Hierarchy
   Context menu: Right click on a class/interface/method name > Open Type Hierarchy

   Figure 2 shows all the classes that extend the *Observable* class in the top pane in the Type Hierarchy view.

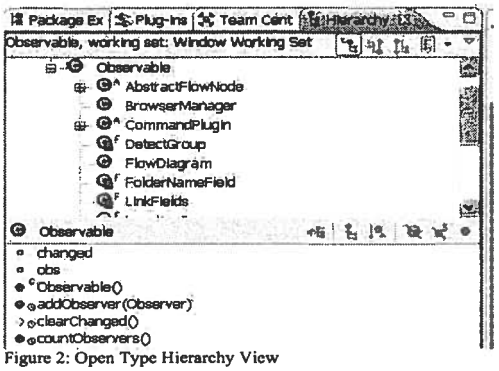


Figure 2: Open Type Hierarchy View

5. **Javadoc View**

You can read the Javadoc comments of the program in the Javadoc view. This view renders comments of the selected class/method in the Javadoc style, if the comments are considered Javadoc comments.

Followings are ways that you can open the Javadoc view.
Keyboard shortcut: Click on a class/method/field name > Alt+Shift+Q, J
Menu: On the Menu bar, select Window > Show View > Javadoc

Figure 3 shows Javadoc comments of *ActionListener* interface.

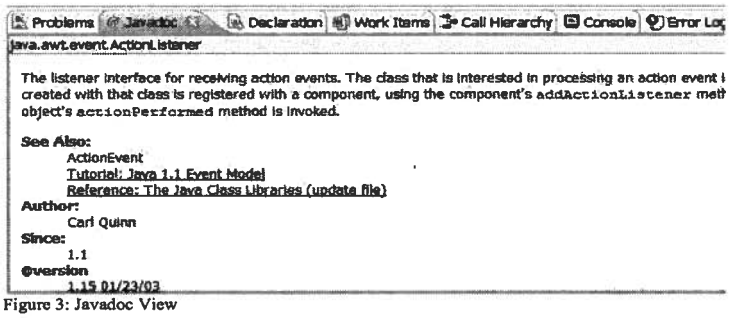


Figure 3: Javadoc View

C.8 Spyglass Introduction

Spyglass Tutorial

Table of Contents

1. Status of Spyglass
2. Reading Recommendations
3. Enabling/Disabling Spyglass Popup Notification

Rational Team Concert (RTC) is an integrated development environment (IDE), which is built on top of Eclipse, a popular open-source Java IDE. It includes most features of Eclipse and adds several collaborative features. Thus, it can accommodate a variety of tool sets for multiple purposes. For clarification, we will refer to each functionality provided by the IDE as a "tool" and a tool belongs to a "feature" or a tool set.

In this experiment, you will be using a prototype called Spyglass that recommends potentially useful functionalities provided in RTC based on your interactions with RTC. In this experiment, Spyglass is limited to recommending you tools that might help you navigate the code on which you are working more efficiently. The rest of this document will help you learn how to use Spyglass.

1. Status of Spyglass

Spyglass can be in different states. Each state is indicated with a different icon.

- : This icon indicates that Spyglass has no new recommendations at the moment. You will be able to enable/disable the Spyglass popup notification by using the drop-down arrow beside this icon.
- (Has New Recommendations): This icon indicates that Spyglass has a new recommendation for you. Once you click on this icon, it will revert back to the Enabled state.

2. Reading Recommendations

At any time, you can click on the Spyglass icon at the bottom corner of the IDE. Once you click on it, you will see two in-place views similar to Figure 1. The one on the rigth is called Recommendation view. This view lists the names of tools being recommended to you, the tools' keyboard shortcuts, how many times they have been recommended (Frequency), and the time they were recommended. If you select one of the recommendations in the Recommendation view, you will be able to see the Rationale view associated with the recommendation. The Rationale view shows why the tool is recommended and in what circumstance you should use the tool. By default, the Rationale view associated with the top-most recommendation is opened.

In Figure 1, the tool "Work Item Editor Links" is recommended. This tool does not have a shortcut key. If you look at the rationale, you will see that this tool is recommended because Work Item 14 is a child of Work Item 7. The user visited each of these work items through a series of navigation commands, but did not use the most efficient tool (the Work Item Editor Links) to view each item. This tool could have helped the user navigate between the two work items faster.

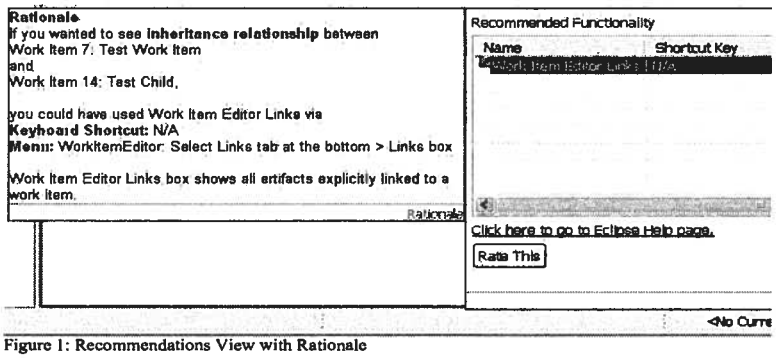


Figure 1: Recommendations View with Rationale

3. Enabling/Disabling Spyglass Popup Notification

By default, when there is a new recommendation for you, there will be a small popup window in the right bottom corner of the IDE along with icon change. If you decide that you do not want this popup, you may disable it at any time. Beside the wrench/light bulb icon in a bottom corner of your IDE, you will see a drop-down arrow. This arrow brings up a menu that allows you to enable or disable our recommendation system popup notification. Once disabled, you will only see the icon change when there is a new recommendation for you.

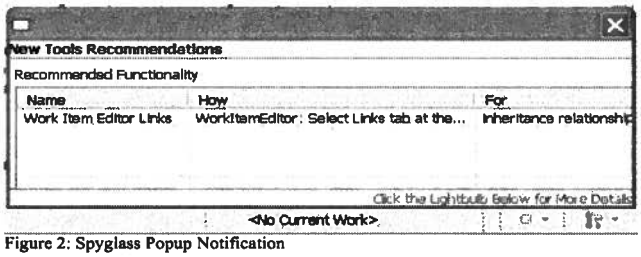


Figure 2: Spyglass Popup Notification

C.9 Training Tasks

THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Training Tasks

Evaluation of a Tool Recommendation System on a Software Development Environment

You are given a simple Paint program. The main class for this Paint program is `PaintWindow.java`, in which the main method is declared and implemented. However, as same as many other graphical-based applications, the `main` method is not the key method that is responsible for "drawing" all the graphic components on the application window. On the other hand, the method `PencilPaint#paint` is the one responsible for rendering the line drawn by the user. Also, this program is event-driven, so it employs listeners for many types of mouse events, such as mouse pressed, mouse dragged, and mouse released.

Task 1: Fix Pencil Draw

Please try running the Paint program. You may notice that the Paint drawing tool does not work. If you try to draw, using Pencil tool, in the space provided on the right hand side, nothing happens. Please fix this. (Hint: Look in `PencilPaint#paint`)

Task 2: Fix Line Draw

After you have finished the first task, you should be able to draw something in the space provided. However, Line drawing tool does not work correctly. Line drawing tool currently will do the same thing as whatever the previous tool selected does. For example, if you select Eraser then Line, the Line tool acts as Eraser tool. If you select Pencil and then Line, the Line tool acts as Pencil tool. Please fix this so that Line always acts as Pencil tool. (Hint: Look in `PaintWindow.java`)

C.10 Tasks for the Programming Session

THE UNIVERSITY OF BRITISH COLUMBIA

Department of Computer Science
2366 Main Mall
Vancouver, B.C., V6T 1Z4

Experiment Tasks

Evaluation of a Tool Recommendation System on a Software Development Environment

Task 1: Fix Pie Chart Labels

Currently, the labels on the pie chart created by PieChartDemo1 appear on top of the pie sections. You can run the PieChartDemo1 by:

1. Right Click on *JFreeChart1.0.11/source/org.jfree.chart.demo/PieChartDemo1.java* in the Package Explorer View.
2. Select "Run As Java Application"

Figure 1 shows what you should see.

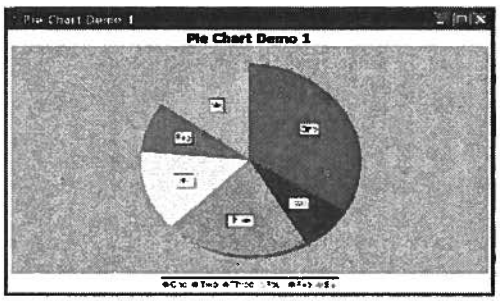


Figure 1: Pie Chart Demo1 at the moment

Your first task is to alter the JFreeChart1 source code to have labels drawn outside of the pie, connected by "doggy" legs to the appropriate pie section. Each label's y-coordinate

should be at the same position as the middle of the corresponding section. Your goal is to make the pie chart look like the screenshot below.

Hint1: Please recall that `draw` method in *JFreeChart1.0.11/source/org.jfree.chart/JFreeChart.java* is the one that is responsible for drawing the chart.
Hint2: There is already a method that is responsible for drawing the "doggy" legs and labels. All you need to do is to find that method and make sure that it gets executed.
Hint3: Pay attention to comments. ☺

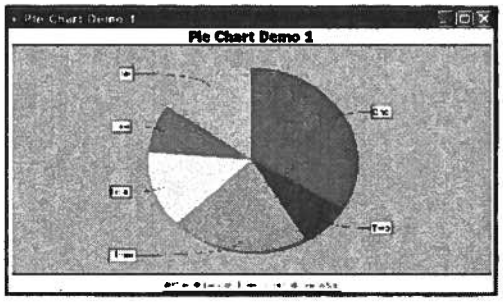


Figure 2: The correct Pie Chart Demo1

Task 2: Add Bar Chart Axes Labels

Currently, JFreeChart does not create labels for the bar chart in BarChartDemo1 correctly. You can run *JFreeChart1.0.11/source/org.jfree.chart.demo/BarChartDemo1.java* the same way you run PieChartDemo1 in Task 1 to see how it looks like. You may notice that even though category-axis (X-axis) label and value-axis (Y-axis) label are specified, the output bar chart does not show label on any axis.

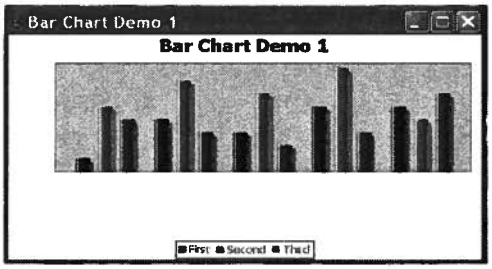


Figure 3: Bar Chart Demo1 at the moment

Please fix JFreeChart so that we see correct labels on value and category axes EXACTLY as shown in the following screenshot.

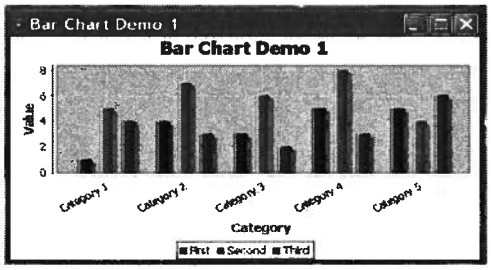


Figure 4: The correct Bar Chart Demo1

Hint: Similar to previous task, there are already some methods that are responsible for drawing the axes and their labels. You have to make sure that they are **"working"** correctly.

C.11 List of Base Code Elements for Task Completion

C.11.1 Task 1

The following are the base code elements highly relevant for completing Task 1 to fix the PieChartDemo1 application.

Table C.1: List of the Base Code Elements for Task 1 Completion

Name of Code Element
`JFreeChart>draw`
`Plot`
`PiePlot PiePlot>draw`
`PiePlot` constructor (the one that is called from within `ChartFactory>createPieChart` only)
`PiePlot>drawPie`
`PiePlot>drawLabels`
`PiePlot>drawSimpleLabels`
Field `simpleLabels` in `PiePlot`
`PiePlot>setSimpleLabels`
`BarChartDemo1>createChart`
`BarChartDemo1>createDataset`
`BarChartDemo1>main`
`ChartFactory>createPieChart` (the one that is called from within `BarChartDemo1>createChart` only)

C.11.2 Task 2

The following are the base code elements highly relevant for completing Task 1 to fix the BarChartDemo1 application.

Table C.2: List of the Base Code Elements for Task 2 Completion

Name of Code Element
`JFreeChart>draw`
`Plot`
`CategoryPlot CategoryPlot>draw`
`CategoryPlot>drawAxes`
`Axis`
`Axis>draw`
`CategoryAxis`
`CategoryAxis>draw`
`NumberAxis`
`NumberAxis>draw`
`ValueAxis`

C.12 Marking Scheme for Task Completion

The following are the criteria that we use to determine how successful a participant is in completing the given experiment tasks in our laboratory study. A successful completion is determined by whether the participant has visited all relevant classes in order to deduce what went wrong in the program, and obtained a very good understanding of how a chart is drawn by JFreeChart library and what causes the problems.

C.12.1 Task 1: Fix Pie Chart Labels

(100%)

1. Looked inside `JFreeChart>draw` and understood that it delegated drawing task to `Plot`. (10%)
   Indications:
   - Read the body of `JFreeChart>draw`, and
   - Stopped at the line where `this.plot.draw` was called and/or went to see the declaration of `Plot>draw` or the `draw` method of any subclass of Plot
2. Understood that `JFreeChart>draw` was called back to draw the chart. (10%)
   Indications:
   - Opened up `JFreeChart` and went to the draw method of JFreeChart, but did not read the body of the method
3. Looked inside `PieChartDemo1>createChart` and understood the control from main method to there. (10%)
   Indications:
   - Visited the `main` method and `createChart` method in `PieChartDemo1` or scanned the entire class
4. Knew that `PiePlot` was associated with `JFreeChart` in this demo. (10%)
   Indications:
   - Opened up PiePlot class
5. Understood that `simpleLabels` controlled whether the labels were drawn on pie sections or not. (25%)
   Indications:

- Paused to read comments associated with `simpleLabels` or attempted to change its boolean value

6. Changed the value of `simpleLabels` in `PiePlot` constructor or used `setSimpleLabels` method to change the value of `simpleLabels`. (25%)
   If the participant fixed the problem with other method (e.g. comment out the other part of if-else clause), we gave only 20% because he didn't find where `simpleLabels` ccould be set/reset.

7. Looked in `PiePlot>draw`, `drawPie`, `drawLabels`, `drawSimpleLabels`, and knew that whether `drawLabels` or `drawSimpleLabels` was called depended on the value of `simpleLabels` variable. (10%)
   Indications:

   - - Traced through all the methods above and stopped in `PiePlot>drawPie`, where there was an if-else clauses, in which `drawSimpleLabels` or `drawLabels` was called, and
   - - Hovered over `simpleLabels` as condition for that clause or stated its name out loud.

C.12.2 Task 2: Fix Bar Chart Labels

(100%)

1. Looked inside `BarChartDemo1>createChart` and understood that `CategoryPlot` was associated with the `JFreeChart` object created and was used to draw the plot area in the bar chart.
   (10%) Indications:

   - Hovered over `CategoryPlot` in `createChart` and read the comments, or
   - Opened up `CategoryPlot`

2. Understood that `CategoryAxis` represented domain axis and `NumberAxis` or `ValueAxis` represented range axis in this bar chart demo. (10%)
   Indications:

   - In `BarChartDemo1>createChart`, hovered over `CategoryAxis` and `NumberAxis`, or

- In `ChartFactory>createBarChart`, hovered over `CategoryAxis` and `Number` or `ValueAxis`, or
- In `CategoryPlot>drawAxes`, hovered over `CategoryAxis` and `ValueAxis`

3. Looked inside `CategoryPlot>draw`. (10%)
4. Understood that `CategoryPlot>drawAxes` was called to draw all the axes, but there was a missing part for the left axis. Correctly fixed the method. (30%)
   Indications:
   - Looked inside `CategoryPlot>drawAxes`, noticed the missing piece (where the comment `drawLeftLabels` was), and tried to fill in the missing piece
5. Looked inside `CategoryAxis>draw` and uncommented the commented code correctly. (15%)
6. Looked inside `NumberAxis>draw` and uncommented the commented code correctly. (20%)
7. Knew that `CategoryPlot>drawAxes` delegated tasks of drawing each axis to the axis itself. (5%)
   Indications:
   - Hovered over the part in `CategoryPlot>drawAxes`, in which `axis.draw` was called, or
   - Navigated from `CategoryAxis>draw` to `CategoryPlot>drawAxes`

C.13 Criteria to Determine Relevant Recommendation

Relevant Recommendation is a recommendation that is given by Spyglass when there is evidence that the recommended tool will become handy. To decide whether the recommended tool will become handy, we consider the following.

1. If the recommended tool is Open Call Hierarchy, we take the following behaviors as evidence if they happen 3 minutes before or after the recommendation:

   - The user wonders out loud what calls a particular method
   - The user uses Search for References tool on a method name and selects the caller of a particular method from the results
   - The user uses File/Java Search or local search with a method name as a keyword and stops when he finds the place where the method gets called

   The following are also considered as evidence:

   - The user (in the tutorial group) uses Open Call Hierarchy within 3 minutes after the recommendation has been created
   - The user (in the Spyglass group) uses Open Call Hierarchy on a method whose callers are not known to the user within 3 minutes after the recommendation has been made

2. If the recommended tool is Open Type Hierarchy, we take the following behaviors as evidence if they happens 3 minutes before or after the recommendation:

   - The user wonders out loud what concrete class implements a particular interface or extend a particular class
   - The user uses Search for References tool on an interface or a class name and selects the class that implements the interface or extends the class
   - The user uses File/Java Search or local search with an interface name or a class name as a keyword and stops when he finds a concrete class that implements/extends it
   - The user follows the inheritance of a class up the type hierarchy (e.g. repeatedly opens the declaration of classes/interfaces that were extended/implemented)
   - The user looks through all the classes, in the Package Explorer view, that have similar names or reside in the same package

   The following are also considered as evidence:

- The user (in the tutorial group) uses Open Type Hierarchy within 3 minutes after the recommendation has been created
- The user (in the Spyglass group) uses Open Type Hierarchy on a class/interface whose type hierarchy are not known to the user within 3 minutes after the recommendation has been made

C.14 Interview Protocol

1. What do you think Open Type and Call Hierarchy do? Why do you use them?
2. How do you think about the idea of having a tool recommendation system like Spyglass in Eclipse?
3. What kind of tools you wish the recommendation suggested you?
4. When you have new software, like Photoshop or MS Office, how do you learn to use them?
5. Do you have any other question or comments?

C.15 Critical Value Table for the Mann-Whitney U Test

In order to accept a hypothesis, the U value of the test scores has to be lower than the critical U value.

U-Distribution Table

Critical Value Table for the Mann-Whitney U Test Chapter 24

1-tail test at α = 0.025 or 2-tail test at α = 0.05

N_2 \ N_1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1																				
2								0	0	0	0	1	1	1	1	1	2	2	2	2
3					0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8
4				0	1	2	3	4	4	5	6	7	8	9	10	11	11	12	13	13
5			0	1	2	3	5	6	7	8	9	11	12	13	14	15	17	18	19	20
6			1	2	3	5	6	8	10	11	13	14	16	17	19	21	22	24	25	27
7			1	3	5	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34
8		0	2	4	6	8	10	13	15	17	19	22	24	26	29	21	34	36	38	41
9		0	2	4	7	10	12	15	17	20	23	26	28	31	34	37	39	42	45	48
10		0	3	5	8	11	14	17	20	23	26	29	33	36	39	42	45	48	52	55
11		0	3	6	9	13	16	19	23	26	30	33	37	40	44	47	51	55	58	62
12		1	4	7	11	14	18	22	26	29	33	37	41	45	49	53	57	61	65	69
13		1	4	8	12	16	20	24	28	33	37	41	45	50	54	59	63	67	72	76
14		1	5	9	13	17	22	26	31	36	40	45	50	55	59	64	67	74	78	83
15		1	5	10	14	19	24	29	34	39	44	49	54	59	64	70	75	80	85	90
16		1	6	11	15	21	26	31	37	42	47	53	59	64	70	75	81	86	92	98
17		2	6	11	17	22	28	34	39	45	51	57	63	67	75	81	87	93	99	105
18		2	7	12	18	24	30	36	42	48	55	61	67	74	80	86	93	99	106	112
19		2	7	13	19	25	32	38	45	52	58	65	72	78	95	92	99	106	113	119
20		2	8	13	20	27	34	41	48	55	62	69	76	83	90	98	105	112	119	127

1-tail test at α = 0.05 or 2-tail test at α = 0.10

N_2 \ N_1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1																				
2					0	0	0	1	1	1	1	2	2	2	3	3	3	4	4	4
3			0	0	1	2	2	3	3	4	5	5	6	7	7	8	9	9	10	11
4			0	1	2	3	4	5	6	7	8	9	10	11	12	14	15	16	17	18
5		0	1	2	4	5	6	8	9	11	12	13	15	16	18	19	20	22	23	25
6		0	2	3	5	7	8	10	12	14	16	17	19	21	23	25	26	28	30	32
7		0	2	4	6	8	11	13	15	17	19	21	24	26	28	30	33	35	37	39
8		1	3	5	8	10	13	15	18	20	23	26	28	31	33	36	39	41	44	47
9		1	3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54
10		1	4	7	11	14	17	20	24	27	31	34	37	41	44	48	51	55	58	62
11		1	5	8	12	16	19	23	27	31	34	38	42	46	50	54	57	61	65	69
12		2	5	9	13	17	21	26	30	34	38	42	47	51	55	60	64	68	72	77
13		2	6	10	15	19	24	28	33	37	42	47	51	56	61	65	70	75	80	84
14		2	7	11	16	21	26	31	36	41	46	51	56	61	66	71	77	82	87	92
15		3	7	12	18	23	28	33	39	44	50	55	61	66	72	77	83	88	94	100
16		3	8	14	19	25	30	36	42	48	54	60	65	71	77	83	89	95	101	107
17		3	9	15	20	26	33	39	45	51	57	64	70	77	83	89	96	102	109	115
18		4	9	16	22	28	35	41	48	55	61	68	75	82	88	95	102	109	116	123
19	0	4	10	17	23	30	37	44	51	58	65	72	80	87	94	101	109	116	123	130
20	0	4	11	18	25	32	39	47	54	62	69	77	84	92	100	107	115	123	130	138

$N_1 \leq N_2$

Adapted from Robert Johnson, *Elementary Statistics, 6th ed.* (Boston: PWS-Kent Publishing, 1992), Appendix F-20 **A3-9**

C.16 Completion Scores of All Participants

Table C.3: Completion Scores of All Participants in Task 2

Participant	Group	Time Used (mins)	Completion Scores (out of 1)
e1	Spyglass	59	1
e2	Spyglass	24	1
e3	Spyglass	35	1
e4	Spyglass	48	.95
e5	Spyglass	60	.20
e6	Spyglass	60	.20
e7	Spyglass	60	.80
e8	Spyglass	33	1
e10	Spyglass	60	.60
f2	tutorial	60	.20
f3	tutorial	52	.85
f4	tutorial	60	.35
f5	tutorial	60	.10
f6	tutorial	60	.65
f7	tutorial	60	.45
f8	tutorial	60	.35
f9	tutorial	60	.20
f10	tutorial	27	1

Table C.4: Descriptive Statistics of All Completion Scores in Task 2

Group	Mean	Stdv	Variance
Spyglass	0.75	0.339	0.115
tutorial	0.46	0.310	0.096

C.17 Unrelated Code Elements Visited by All Participants

Table C.5: Number of Unrelated Code Elements Visited by All Participants in Task 2

Participant	Group	Number of Unrelated Code Elements Visited
e1	Spyglass	0
e2	Spyglass	0
e3	Spyglass	9
e4	Spyglass	2
e5	Spyglass	1
e6	Spyglass	2
e7	Spyglass	4
e8	Spyglass	4
e10	Spyglass	5
f2	tutorial	10
f3	tutorial	12
f4	tutorial	14
f5	tutorial	13
f6	tutorial	3
f7	tutorial	2
f8	tutorial	5
f9	tutorial	8
f10	tutorial	3

Table C.6: Descriptive Statistics of Unrelated Code Elements Visited by All in Task 2

Group	Mean	Stdv	Variance
Spyglass	3	2.872	8.250
tutorial	7.78	4.684	21.944

C.18 Target Tools Usage Frequency of All Participants

Table C.7: Target Tools Usage of All Participants in Task 2

Participant	Group	Frequency of Using OTH	Frequency of Using OCH	Time Known OTH (mins)	Time Known OCH (mins)	DU for OTH	DU for OCH
e1	Spyglass	0	8	0	60	-	0.133
e2	Spyglass	0	0	0	0	-	-
e3	Spyglass	0	0	10	4	0	0
e4	Spyglass	0	4	0	24	-	0.167
e5	Spyglass	2	0	33	27	0.06	0
e6	Spyglass	0	1	0	60	-	0.017
e7	Spyglass	9	1	60	19	0.15	0.053
e8	Spyglass	0	3	60	3	0	1
e10	Spyglass	0	8	0	52	-	0.154
f2	tutorial	0	0	60	60	0	0
f3	tutorial	0	0	60	60	0	0
f4	tutorial	0	31	60	60	0	0.517
f5	tutorial	0	3	60	60	0	0.05
f6	tutorial	0	8	60	60	0	0.133
f7	tutorial	49	0	60	60	0.817	0
f8	tutorial	0	21	60	60	0	0.35
f9	tutorial	0	15	60	60	0	0.25
f10	tutorial	0	11	60	60	0	0.183

Table C.8: Descriptive Statistics of Target Tools Usage Frequency for All Participants in Task 2

Tool	Group	Mean	Stdv	Variance
OTH	Spyglass	1.22	2.991	8.944
	tutorial	5.44	16.333	266.778
OCH	Spyglass	2.78	3.270	10.694
	tutorial	9.89	10.845	117.611