Panorama – a software maintenance tool

by

Naga Bhagvanth Ram Vattumalli

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Simanta Mitra, Co-major Professor
Carl K. Chang, Co-major Professor
Shashi K. Gadia

Iowa State University
Ames, Iowa
2010

Table of Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my heartfelt gratitude to everyone who helped in my research work and thesis. First and foremost, I would like to thank Dr. Simanta Mitra for the guidance and support throughout the course of research. He inspired and guided me through every phase of the research, which helped in achieving the goals. I would like to thank my committee members Dr. Carl K. Chang and Dr. Shashi K. Gadia for their support and contributions to the work. I would like to thank Michael Pawlovich, Saravana Chellappan, and Archit Saraf for their feedback and suggestions on the case study which will help in further research on Panorama. Lastly, I would like to thank my parents V. Nageswara Rao and A. A. Manga Tayaru for providing the support and encouragement to achieve my goals.

Abstract

Much of the effort in software maintenance is spent on finding relevant information and on program comprehension. Of the several challenges encountered during this process, some are: a) inadequate documentation, b) the developer doing the maintenance activity may not be the one who actually developed it and may be unfamiliar with the application domain (in addition to the unfamiliar code), c) information overload, and d) the relevant code may be scattered across multiple files of different types making it harder to find. Existing documentation in the form of Javadoc is inadequate in providing a global view of the working of the software.

Panorama, a java based Eclipse plug-in, was developed to facilitate maintenance activities by providing mechanisms to document and to view expert knowledge and relevant code in the form of a concern. Some features of Panorama are: a code tracing feature that allows the expert to quickly find (so he can document it) lines of code executed in carrying out a function, a concern management feature that allows the expert to create and organize concern information in a hierarchical manner, a concern visualization and context management feature that helps the maintainer to handle information overload by allowing him to switch between contexts, an enhanced user-interface that helps the maintainer to easily navigate between relevant contexts and codes. Panorama also provides a Javadoc -like documentation of cross-cutting concerns that supplement existing Javadoc documentation to provide comprehensive information about the software.

In a case study done to validate the usefulness of our tool, Panorama was used to document the SAVER software (a VB.NET based fairly large GIS software with 26,704 executable lines of code that is being actively used by the Iowa Department of Transportation to analyze automobile crashes over a period of time). SAVER has been undergoing continual bug-fixes and enhancement activities – and preliminary studies indicate that the supplementary documentation provided by Panorama has proven beneficial.

Chapter 1. Introduction

1.1. The problem

Maintenance tasks can be expensive depending on the size and complexity of the target software. For instance, past research has shown that a software engineer spends around 50 to 70 percent of his time in making changes to mission critical software [30, 12]. *"In a survey conducted, SEs said that they spend 57% of their time in fixing bugs and 35% in making enhancements to system [32]"*.

Most of the programmer's time in maintenance efforts is spent in gathering information [5] and understanding what an existing system does – when and where it performs its operations. This is generally because the developer fixing or enhancing software may not be the one who actually developed it. Another issue that affects maintenance task is information overload in which, a person faces difficulty in understanding the issue because a lot of information is presented to him. As the complexity of software increases, it becomes difficult to identify the concerns of interest for a maintenance task. A concern can be any conceptual unit a developer can think about. For example, a concern can be the code that implements a particular feature of an application. A concern can also be scattered code that implements non-functional requirements like performance issues, security etc [23]. A simple change to the software can impact single or multiple concerns. In addition to the problem of identifying and locating the code of interest for a maintenance task, it also becomes difficult to understand how a change to one feature or concern impacts other features or concerns.

These problems generate a need for tools to assist in software comprehension to reduce the maintenance time, and thus improve software maintenance productivity. Also, a facility to document the knowledge of software's actual developer is needed to reduce the maintenance time.

1.2. Panorama – a solution approach

Panorama was developed to address the above problems. It is an eclipse plug-in that helps software developers by providing required information to understand and maintain a

piece of functionality without unknowingly affecting other features of the software. Panorama can be considered as a tool to manage a layer of information on top of existing documentation. It is used to document the knowledge and thoughts of developers who originally created the software.

Once the Panorama documentation is generated for an application, a maintainer can use the concern hierarchy provided by Panorama to locate relevant code segments. He can use the visualization feature to understand the behavior of a concern. Also, he can directly navigate to the corresponding code from the concern visualization to understand how it is implemented. In this way, a maintainer can easily identify and understand the code required to perform a maintenance task. There are other useful features provided by Panorama that are described later. To understand the usefulness of Panorama, a case study was conducted where a fairly large application was documented using Panorama. Our experiences from the case study and also comparison with existing tools show that Panorama is uniquely suited to aid developers in performing maintenance tasks.

1.3. Work done

The initial work on Panorama development started in 2008 [42]. This work was continued in 2009 with the implementation of Panorama as a plug-in for the Eclipse environment [23]. Our contribution started with making changes to this existing code base and can be classified into four major areas of effort:

1. Catching up with previous work on Panorama
2. Literature Survey
3. Enhancements
4. Case Study

The work done to accomplish each of the above tasks is explained in the following subsections.

1.3.1. Review of previous work on Panorama

The AspectJ Visualizer plug-in [1] was extended to create the initial version of Panorama. This Visualizer represented information provided to it in the form of bars and

stripes and Panorama extended this to represent concerns and code-segments as bars and stripes respectively [42, 23]. Users could click on a stripe in the "*Visualizer view*" (as shown below) to view its corresponding code segment. There were a few additional helpful features that we do not discuss here.

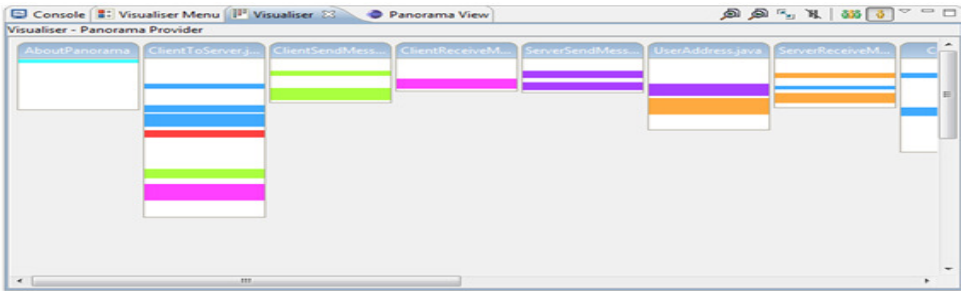

Figure 1: Panorama Visualizer view

An experimental study was conducted to validate this tool and its approach [23]. The study showed that subjects in the experimental group who used the Panorama tool were able to complete more maintenance tasks than the subjects in the control group who didn't have access to the tool. At the end of study, feedback was collected from the experimental group to document the subjects' experience in using Panorama.

As an initial preparation to enhance Panorama, the existing code and design was studied. Based on its results and analysis, various lessons learnt were summarized which provided a direction for the enhancements done to Panorama. Following are some of the key lessons from the previous work:

- Concerns need to be presented in the form of a sequence diagram to help the maintainer understand the execution flow.
- Better navigation facilities need to be developed for both between different views and also within a view.
- Switching views to show only the files that are associated with a chosen concern would be important to reduce information overload.
- The user interface needs to be redesigned so as to make it more easy and intuitive for the maintainer.

1.3.2. Literature survey

To better understand the state of the art in Software maintenance research, we performed a literature survey on the following areas:

- Existing tools for software maintenance,
- Software comprehension theories (on how programmers go about understanding software), and
- Experimental studies on software maintenance.

1.3.3. Enhancements to Panorama

Based on the criteria gathered from the literature survey, enhancements (See Figure 2) were done to Panorama to improve its effectiveness. The major enhancements done to Panorama are - concern visualization, code coverage, offline documentation, maintenance of concern context, and search features. Some of the minor enhancements done to Panorama are – simplifying navigation and user interface, adding new views, and linking code and views. The enhancements are briefly described below.

Major enhancements

- A Graph visualization feature has been added to generate a graph for a chosen concern to help in better understanding its flow. The user can click on a node of the generated graph to view its corresponding code.
- A code instrumentation facility has been added to help an expert developer view the lines of code executed to help him in the concern documentation process. The user can start and stop instrumentation at any time during the execution of the code thus allowing the user to focus on a particular sequence of actions.
- A facility has been added to allow offline documentation of concerns in a format similar to Javadocs used in document Java code. The user can click on links provided in the documentation to view corresponding code.
- A concern context feature has been added to the Panorama Hierarchy view which helps to hiding unnecessary information and shows only the files related to the chosen concern.

- A search facility has been added to Panorama to search for required information from concern documentation.

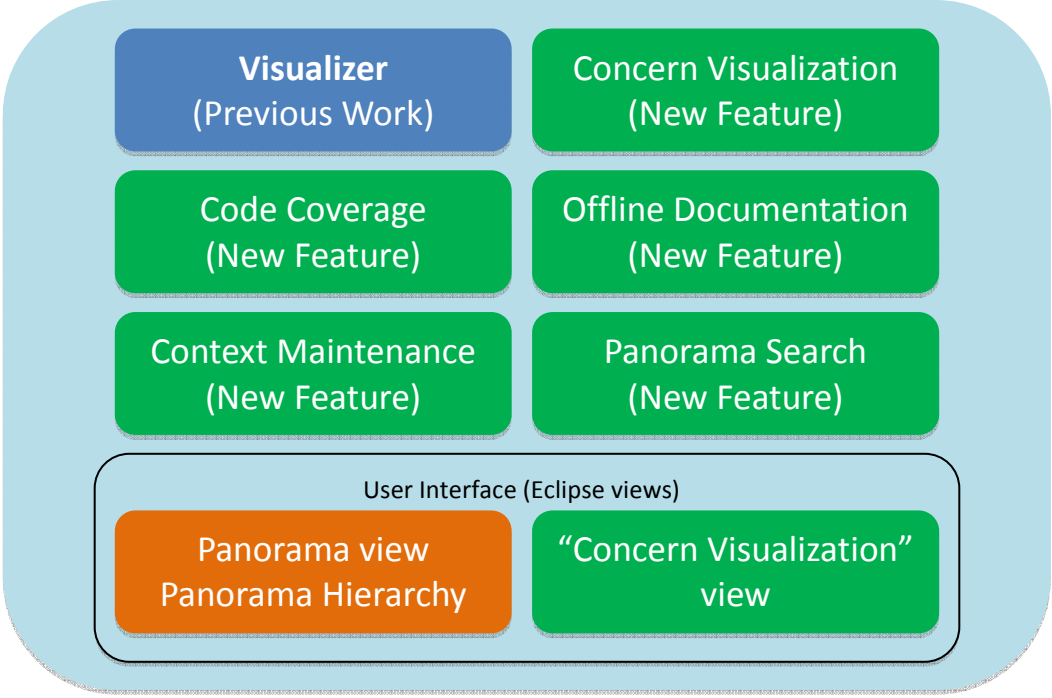

Figure 2: Panorama enhancements at a glance

Other enhancements

- The Panorama user interface has been enhanced to simplify the process of creating, editing, and viewing the concerns and stripes.
- The Panorama Hierarchy view has been improved in a few ways for easy understanding of concerns: for example, a user can now click on a stripe to view its associated code.
- The user is now provided with a facility to navigate between the stripes of a concern without using the Panorama Hierarchy view. The user can click on the markers of a stripe (in the code view) and a quick window pops up with the links to navigate to the connected stripes. This will reduce the complexity for a user in understanding the execution sequence for a particular concern.

To make the enhancements, additional libraries like ZEST, Eclemma and JGraph were included in the Panorama plug-in. We will provide further details on each of these libraries in the "Design" chapter.

1.3.4. Case study

To validate the effectiveness of Panorama, we documented a fairly large application (a VB.NET based application named SAVER with 26,704 executable lines of code)using Panorama. We analyzed the documentation and the maintenance workflows to identify areas of strengths and weaknesses of our current implementation. Preliminary studies indicate that the supplementary documentation provided by Panorama has proven beneficial.

The organization of rest of the document is as follows. In Chapter 2, we discuss existing software maintenance tools, theories on software comprehension, experimental studies on software comprehension, and the criteria for a successful maintenance tool. In Chapter 3, we describe the design of Panorama. In Chapter 4, we validate the effectiveness of Panorama. Finally, in Chapter 5, we present our conclusions and ideas for future work.

Chapter 2. Related Works

There are three areas of research that are relevant to our work. The first area is that of tool development by researchers to support software maintenance (and to simplify the process of understanding programs). The second area deals with theories on software comprehension in which a connection is provided between human psychology and processes used by humans in understanding programs. The third area is experimental studies done to collect data on software maintenance activities and their efficacy.

In this chapter, we discuss recent research activities in each of the above three areas. We also provide a summary of the results from our related research survey, although we leave the bulk of comparison between our work and the closely related work to the discussions in Chapter4.

2.1. Tools

Over the past several years, many tools have been developed to help with program comprehension and to support maintenance tasks. Existing tools can be grouped into two categories. The first category contains tools that help a developer in maintenance or development tasks by providing information related to the current concern context. This information is generated dynamically using code analysis techniques. The second category contains tools that generate documentation from source code. This information is generated statically (one time) from comments and annotations about each program segment that have been entered by developers.

2.1.1. Development tools

Of the tools surveyed, many of them use the concept of “concerns” and provide some or all of the following functionalities:

- the ability to view or explore information related to a concern,
- the ability to display only the relevant information,
- a mechanism to navigate through the presented information, and

- the ability to search for relevant information or to reduce the task context for simplifying search process.

Of these tools, we consider some of them to be more closely related to our work (such as FLAT^3, Mismar, Jasper, and Sextant) than others (such as FEAT, JQuery, and ConcernMapper). There are also other maintenance tools (such as Mylar, JinSight, BLOOM, and Diver) that do not use the concept of "concerns" and we have labeled them as "least related to Panorama". A tabular comparison of the functionalities provided by these tools is provided in Appendix A.

FLAT^3 [28] is an application that allows one to search for locations in code related to a specific feature by combining textual search and dynamic tracing techniques. It provides a suite of tools that helps a developer to find code related to a feature and save it for future use. Its textual feature location technique helps in finding the relevant code by searching code by utilizing the Lucene information retrieval library. Its dynamic feature location uses a tracing tool to bring down the search context to find the relevant code. It also has a visualization feature to provide a global overview of search results across files. This Visualizer shows code segments found in the form of stripes inside bars representing the files they are in. The intensity of shade for a stripe represents the degree of similarity of code to the feature. We consider this tool to be the most similar to Panorama and the differences are elaborated in the "Case Study and Discussion" chapter.

Jasper [3] is an eclipse plug-in that provides a facility to gather information related to a task in the form of working sets. Unlike FEAT and JQuery, it allows selection of arbitrary code segments. It also allows for simultaneous viewing of arbitrary code sections in separate windows. The working sets can be saved for sharing of information or for recovering the context. The goal of Jasper is to help with context management for tasks – where as the goal of Panorama is to provide concern-based documentation. The tool is also different in that it doesn't provide a way to describe or visualize the relationships between various elements of a concern (which in their case is a working set).

SEXTANT [29, 11] is an exploration tool that helps in navigating the software to find the relevant code for a task at hand by searching or browsing. In this tool, the developer first searches for the starting element based on his domain knowledge, and then use this starting element to find further elements that are related to a concern. The information is shown in the form of a graph that shows the relationships between the elements. This tool provides a facility to hide unnecessary information in order to reduce information overload. It is different from Panorama in that it focuses on efficient exploration of information rather than on documentation. In many cases, the elements found could be remotely related to the context (and thus can make it harder for someone to understand the code). It is probably more suited for the developer who already has an in-depth understanding of the code. The differences with Panorama are further elaborated in the "Case Study and Discussion" chapter.

Mismar [6, 7, 8] is a tool that can be used to provide documentation for active concerns where documentation is provided in the form of a guide for the functionality. The guides have a step associated with each type of element – "*For example, if user chooses an XYZ class, an "Extend XYZ class" step is created [6]"*. The tool is also integrated with the eclipse environment and thus if a user performs this step, the "create new class" wizard (customized with the information in the guide) will open. The tool is similar to us in terms of having an ability to provide documentation to developers but is different from Panorama in the way concern information is presented to the user. For example, we could not find any evidence of the availability of offline documentation or interactive graph visualizations. The differences with Panorama are further elaborated in the "Case Study and Discussion" chapter.

FEAT [25] is a tool designed to locate, describe, and analyze the code related to a concern. It provides a facility to maintain a concern hierarchy. User can start with a single class and make queries iteratively to build up a concern graph representing the codes structural dependencies. One way that this tool is different from Panorama is that it doesn't allow the inclusion of arbitrary lines of code to a concern. Also, like SEXTANT, its automatic detection of dependencies can lead to information overload [23].

ConcernMapper[26] is a lightweight eclipse plug-in developed by the same group that developed FEAT. This tool helps in grouping the methods and fields related to a maintenance task in the form of concerns. The created concerns can be used to highlight related code in various views of eclipse IDE. One way that this tool is different from Panorama is that it doesn't provide any way to specify relationships between code segments grouped under a concern. Also, as the tool focuses on grouping the related code rather than on documentation, it doesn't provide any information to the developer on the design rationale or the execution flow.

JQuery [14] is built on top of TyRuBa, an expressive logic query language to combine a hierarchical browser with a query tool. The tool is similar to FEAT in that user can build a tree structure of dependent code by iteratively querying for elements. Once a tree is generated based on a query, the developer can further refine it based on additional criteria. This tool is different from Panorama in that it doesn't allow selecting arbitrary lines of code to create the structure [23] and thus an expert developer's knowledge cannot be utilized in the construction of the tree structure. Also, like SEXTANT and FEAT, its automatic detection of dependencies can lead to information overload [23].

NavTracks [31] is a tool that keeps track of a developer's navigation history and automatically forms associations between related files. As the developer proceeds with his work, the tool recommends potentially related files based on the associations formed based on his previous navigation patterns. It is different from Panorama because it is a tool to support browsing through the software rather than a documentation tool.

Nacin [21] is an eclipse plug-in that records a developer's navigation activity and generates sets of elements related to different features of a current task. The "Recorder" records the activities of developer and the "Inference Engine" groups the elements that are potentially involved. The concerns generated will be approximate and have to be perfected by the developer manually. This tool doesn't provide any information to the developer on the design rationale or the execution flow.

ActiveAspect[40] is a tool to present the crosscutting information effectively based on the aspects as captured by AspectJ. It combines the abstraction of the concern structure with user interaction to allow a developer to increase the context of a diagram as they investigate the presented structure. The tool focuses more on presenting the concern interactions than on documentation. This tool has a very strict interpretation of the notion of a concern (as defined by pointcuts and join points) as opposed to Panorama, which allows arbitrary selection and groupings of code to be called a concern.

CME,ConMan, and FluidAJ are a few tools which are based on the concept of a concern but with different goals than that of Panorama. *"CME comprises of a suite of Eclipse-based tools that aid in identification, encapsulation, extraction, and composition of new and existing concerns in software, and an integrating platform on which AOSD technology providers can build new tools [23].*" The difference between CME and Panorama is that, like ActiveAspect, they use the notion of concern more formally as entities that can be explicitly represented and encapsulated when compared to our notion as a point of interest [23]. "*ConMan is a similar tool that supports multiple, overlapping, and concurrent concerns [23]*". FluidAJ uses pointcuts to search for different joint points and similar code. The goal of the tool is to be able to view all such code sections at the same time and to allow modification at one place to affect all sections. "*Even though the tool helps in identifying such cross cutting sections, it is not possible to identify all types of crosscutting concerns using pointcuts [23]"*.

In addition to the above, there are a few other tools that help a developer during the maintenance task but their goals are different than ours and are not based on the concept of a concern.

- Mylar [17, 18] is an eclipse plug-in that helps a developer to focus on a particular task. It captures the task context based on the programmer's activity using a degree of interest (DOI) model. The tool uses colored shading to present a user's DOI using a Visualizer tool to depict stripes where the color of a stripe darkens with the increase in DOI (This is similar to FLAT^3).

- Jinsight [22] helps to visually analyze a program's runtime behavior and is designed with object-oriented and multi-threaded programs in mind. It identifies performance bottlenecks, deadlocks, shows object and thread communications, and garbage collection activity. It allows programmers to visualize event sequences for performance analysis purposes and supports task-based trace to find details of a selected program task.
- BLOOM [36] is a visualization system that lets the user understand and address specific problems by collecting a variety of program information like program traces, structural information, and semantic information. User can also control the amount and type of data collected.
- Diver [9] is a set of tools that enhances the eclipse IDE to incorporate reverse-engineering technique in every day work. It uses static and dynamic analysis to help a developer understand the software. Some of the features include tracing program execution, filtering eclipse views based on this trace, and also comparing between traces.

2.1.2. Documentation tools

Here we present some tools that generate code documentation. Unlike these tools, Panorama generates documentation for crosscutting concerns based on information provided by the expert developer. Details can be found in the chapter on Panorama "Design".

Javadoc is the document generator for java code. This tool automatically generates HTML files from comments written by developers in the source code. An extension to Javadoc named yDoc can generate high quality UML diagrams and embed them into the documentation. DOC++ is a tool similar to Javadoc but is developed for C++ programming language.

Another similar tool, Doxygen [35], is used to document codes written in C++, C, Java, Objective-C, Python, IDL (Corba and Microsoft flavors), FORTRAN, VHDL, PHP, C#, and to some extent D. It generates reference documentation for software in various formats like HTML, CHM, RTF, PDF, LaTEX, PostScript or man pages. It generates the documentation by extracting source code comments. The documentation is cross-referenced with the

corresponding code so that users can easily traverse to the actual code from the documentation.

progDoc [39] is a language independent program documentation system that maintains consistency between software and its documentation. It can work with any programming language that has code accessible from files and can contain comments. The source code and documentation are independent and are linked by special handles that are placed in the comments of the source code.

In addition to these tools, efforts were made to document information either by using tags in source code comments or by providing syntactic extensions. JTourBus [15] is an eclipse plug-in and uses Javadoc tags to document information that can be used for interactive navigation in source code. There are tools [10] that can used to document design pattern by utilizing additional tags in Javadoc annotations. Elide [2] is a tool that provides syntactic extension to java language to allow developers to introduce new modifiers to capture design concepts and to generate corresponding documentation.

2.2. Comprehension theories

To build useful tools for software maintenance, it is important to understand how developers go about understanding programs i.e. theories on program comprehension. Program comprehension can be defined as the process of *"understanding what a program does and how it does it in order to make functional modifications and extensions to it without introducing errors [4]"*. As a result of research conducted over the past few decades, several cognitive models of program comprehension strategies have been proposed to describe a programmer's behavior [37]. In this section, we will first define some terms related to program comprehension and then discuss the cognitive models proposed by researchers.

2.2.1. Definitions

When understanding programs, a developer, based on observation and inference, forms a mental representation of the program called the ***mental model*** [16]. Depending on the level of abstraction, Pennington (1987) described two models formed by a developer when

understanding a program – program model and domain model. Thus, there are two mental models – the program model and the domain model. ***Program model*** is a low level abstraction that includes coding level constructs whereas the ***domain model*** is a high level abstraction of the application domain. Program model can consist of programming plans and beacons. *"**Programming plans** are generic fragments of code that represent typical scenarios in programming. For example, a sorting program will contain a loop which compares two numbers in each iteration [38]*." ***Beacons*** are recognizable features that help identify the presence of a structure [38, 28].

A ***cognitive model*** describes the cognitive processes and cognitive memory structures used by programmers in forming the mental model. When understanding a program, the approach followed by a programmer is called ***comprehension strategy***. During the comprehension process, developers use a comprehension strategy in order to form a mental model of the program. The strategy followed can depend on the program, domain, and the task at hand.

Various comprehension strategies have been defined depending on the direction and the breadth of comprehension. "*The **direction** of comprehension strategy concerns the programmer's strategic approach to program comprehension whereas **breadth** concerns the approach to scope of comprehension [4]*".With respect to the breadth of comprehension strategy, two approaches were observed – systematic and as-needed approach [16]. In ***systematic approach***, a programmer tries to understand the overall design of the program whereas in ***as-needed approach***, he/she tries to understand minimum amount of code required to finish the task at hand [16,4].

With respect to the direction of comprehension strategy, there are mainly three cognitive models that can be mainly classified as Top down, Bottom up, or a mix of these two [16]. In the following subsections, we describe each of these models.

2.2.2. Top-down comprehension

In the ***top-down approach***, developers first focus on understanding the overall purpose of a program by starting from a general hypothesis. Based on the initial information available

apart from the source code such as title or description, the programmer makes a general hypothesis about how the program works [16,4]. Based on this hypothesis, the programmer expects to find various structures which when confirmed can be used to make more specific hypotheses. Once a specific hypothesis is formed, the programmer tries to verify the hypothesis by searching the code for beacons - an indicator (such as some specific declarations, comments, or programming constructs). If he is successful in finding the beacons, it helps to strengthen or confirm the current hypothesis. If the beacons are not found, he should use the information to modify his hypothesis. This process will continue till the program is understood [4].

2.2.3. Bottom-up comprehension

In a ***bottom-up approach***, programmers start reading code to understand individual lines and tries to recognize patterns at a low level, which are further grouped into meaningful high level structures or **chunks**. Shneiderman and Mayer proposed that the programmer studies individual lines of code and encodes them in short-term memory. This information is further grouped with related ones using a chunking process in order to create a semantic representation of larger unit. Long-term memory will help in forming the chunks by providing semantic and syntactic knowledge. This bottom up construction continues by establishing relationships between already constructed chunks. This entire process will be carried out until the representation is complete for the entire program [Pennington 1987, Shneiderman and Mayer, 1979].

The Pennington model explained that two types of abstractions are done during the comprehension process. The program model is an abstraction at a low-level, close to the program code and control flow, where as the domain model is an abstraction at a high level, with the knowledge of data flow and functional relationships. When a program is new to the programmer, he will first form a program model. The domain model is formed from the knowledge of program model combined with the knowledge of plans related to the program domain [4, Pennington (1987)].

2.2.4. Mixed comprehension

In a ***mixed approach***, programmers switch between top-down and bottom-up based on the situation. These models were proposed by Letovsky, Von Mayrhauser, and Vans [4, 37]. Letovsky considers programmers as opportunistic and that they switch their approach between top-down and bottom-up based on the situation. His model consists of three components – knowledge base, mental model, and the assimilation process. The knowledge base encodes the expertise of a programmer and his knowledge about the application. *"The mental model encodes the programmer's current understanding of the program."* The assimilation process explains how this mental model evolves based on the knowledge base and information about the program being comprehended [37]. Von Mayrhauser and Vans combined the above top-down, bottom-up, and opportunistic model to propose a single meta-model. *"They proposed that understanding is built concurrently at several levels of abstraction by freely switching between the three comprehension strategies [4, 20]"*.

2.3. Experimental studies

The third area of our research survey is about experimental studies that were conducted by researchers to better understand program comprehension theories and to develop better maintenance tools. Here we report on studies that compared top-down and bottom-up strategies, systematic versus as-needed strategies, procedural versus object-oriented, and expert versus novice programmers.

2.3.1. Top-down vs. bottom-up strategy

Koenmann and Robertson (1991) supported top-down approach by arguing that comprehension activities are mostly top-down in large programs. Shaft and Vessey (1995) and von Mayrhauser and Vans (1996) proposed that programmers follow the top-down approach to understand a program if they are working on a familiar domain and can recognize a lot of plans. They tend to use bottom-up approach when they are working on an unfamiliar domain and the code is new to them. However, Mayrhauser and Vans (1998) observed that the selection of top-down or bottom-up approach depends on the task, domain familiarity, and the program at hand. In a large size program, both top-down and bottom-up

approaches will be used because the programmers' familiarity with the code varies in different parts of the program. A programmer uses more top-down approach in adaptive maintenance than in the situations of corrective maintenance. A programmer familiar with the domain takes a more top-down approach than the ones with less domain knowledge. [4]

Similar studies were also conducted to understand the object-oriented comprehension strategies. Burkhardt et al. (1998) studied the comprehension process of an expert in the object-oriented paradigm and found an evidence of top-down behavior. According to them, novices used more bottom-up strategy and were more execution based. In a study by Corritore and Weidenbeck(2000, 2001), they observed that it is more top-down during the initial stages of comprehension and were more bottom-up during the later stages [16].

The studies conducted to understand comprehension direction couldn't completely support one strategy over the other. It was also suggested that with experience, programmer will know which strategy will be most appropriate for the task at hand. Thus, program comprehension tools should support or enhance the programmer's preferred strategies instead of supporting a fixed strategy [37].

2.3.2. Systematic vs. as-needed strategy

In a study by Littman et al (1986), he observed systematic and as-needed strategies regarding the breadth of comprehension. Programmers using systematic approach were more successful in doing the modifications in the study when compared to programmers following the as-needed approach (supported by [24]). But, in a study conducted by Koenemann and Robertson (1991), they found none of the programmers were truly using systematic approach and argued that systematic approach is unrealistic in large programs. However, Corritore and Wiedenbeck (2001) reported that both procedural and object-oriented programmers use a wide breadth of comprehension approaches during the maintenance tasks.

Amela Karahasanovic, Anette Kristin Levine, Richard Thomas (2007) argued that the studies were conducted mostly in procedural paradigm and also the number of lines of code were too less to be considered as large programs (the Koenemann and Robertson used 636 LOC and Corritore and Wiedenbeck used 783 lines for procedural and 822 LOC for OO

programmers). Thus, they conducted a study in a larger setting with the application having 3600 LOC and 38 participants. Based on the results from the study, they supported Littman et al. that programmers using systematic strategy were able to perform the tasks more correctly when compared to the ones using as-needed strategy. Also, they found that 22 participants out of 38 used the systematic strategy and thus supported the results reported by Corritore and Wiedenbeck (2001).

Thus, a program comprehension tool should help the developers following the systematic approach by helping them in understanding the overview of an application. Also, it was found that OO programmers may focus on documentation in the early stages of documentation to understand the domain objects and their relationships [4]. Thus, a developer should be provided with a good high-level documentation in the initial stages of comprehension.

2.3.3. Comprehension process (procedural vs. object oriented programmers)

The study conducted by Corritore and Wiedenbeck (2000) was to analyze the software comprehension activities in repeated maintenance tasks and how they differ between procedural and object oriented programmers. The study tried to examine both the direction as well as the breadth of comprehension. To analyze the direction of comprehension, they used documentation files, header files and implementation files which correspond to abstraction levels from higher to lower respectively. The data collected by a screen capture program was used to analyze the type of files browsed to perform the task at hand.

When the results from all participants and activities of the study were examined together, it showed that programmers used more implementation and header files than that of the documentation, which suggested that, the comprehension direction was bottom-up. However, the direction changed during different phases of the maintenance task. During the initial study phase, OO participants used more documentation files and thus, the comprehension direction was more top-down. However, during the study period, procedural participants used more bottom-up approach when compared to the OO participants. It was also noticed that the participants used less documentation files during the first modification task and this trend

intensified in the second modification task. This suggested a more bottom-up approach during modification tasks.

Based on the results from this study, it was suggested that programmers use more documentation files during the initial phases of comprehension process. Also no one strategy was followed throughout the maintenance process and thus the tools developed to support maintenance tasks should be able to support both top-down and bottom-up comprehension.

2.3.4. Novice vs. expert developers

From the study conducted by Singer (1997), it was observed that software developers spend a considerable amount of time to understand the source code before doing modifications. These modifications could be fixing bugs or adding new features. The main difference between novices and experts is that novice developers are often less focused [41], do not have a clear idea of what to search for, and thus spend more time in studying things that are unrelated to the task at hand. Novice developers do not have an idea of how the system works and thus must learn at both the conceptual level and the detail level. Experts, on the other hand, know the system but still are unable to maintain the complete mental model of the system (as it is large) [32].

From the study conducted by Tegarden and Sheetz (2001) to understand the difficulties faced by developers in OO development, it was found that decomposition was the major contributor for the complexity in OO development [16]. It was also revealed that "*novices get overwhelmed with implementation issues while experts were more focused on project management issues [16]*". The study conducted by Amela Karahasanovic, Anette Kristin Levine, Richard Thomas to understand the problems faced by the developers revealed that major specific difficulties were related to understanding structure of the application and the inheritance of the functionality [16].

Thus, a software comprehension tool should be able to show the relevant information for a task so that novice developers do not get deviated. The amount of information displayed must be controlled to help users from being overwhelmed with information. Also, the tool should provide a way to easily understand or remember the architecture of an application.

2.4. Criteria for a maintenance tool

Our literature survey served to address the following questions:

1. What are the features supported by existing tools dealing with concerns management?
2. What are the underlying theories of software comprehension?
3. Where are the gaps in current tools? How can we use this information and the knowledge of the comprehension theories to provide adequate support to the expert and to the novice?
4. How can we improve Panorama documentation process for an expert developer?

Here is a summary of features that must be supported by a successful maintenance tool:

- Corritore and Wiedenbeck suggested that OO programmers may focus on documentation in the early stages of comprehension to understand the domain objects and their relationships [29]. Thus having a good high-level documentation at the initial stages of comprehension may help in better understanding of the overall design of software ***–CRI 1***.
- Storey at al. observed that the strategy followed depends on various factors like the programmer's experience, task at hand and the program to understand [29, 37]. Also from the result of the study by Corritore and Wiedenbeck, it is showed that object oriented programmers tend to use different strategies during different phases of maintenance task [4, 29]. Thus there is a need to support integrated comprehension and the maintenance tools should provide different approaches for programmers using different techniques [4] ***– CRI 2***.
- Modern software projects contain different types of documents including source code, configuration files, XML descriptors etc. Also, the information stored in these documents is related to each other. Hence this generates a need for the software comprehension system to work with different programming languages [29] ***– CRI 3***.

- Developers explore software to understand which elements are relevant for a given task and how those elements are related to each other [29, 19]. Thus there is a need to show an explicit map of the exploration path for a task at hand [29,13] **– *CRI 4***.
- The map should be provided in such a way that the developer doesn't get lost in the exploration process [29, 32] **– *CRI 5***.
- It is also observed that developers often tend to forget about an area of a system when they move to other parts. Thus they have to re-visit the previously visited elements to recall the knowledge [29]. Thus, the software should help a developer in remembering the previous knowledge gained **– *CRI 6***.
- A software exploration might only give the idea on how software works. Reading of the source code may be required to understand in detail how software works or to confirm a hypothesis about its working [29]. *"Thus, traceability, i.e., the ability to switch seamlessly between the graphical notation and the corresponding source code, is essential for practical use [29, 33]"***– *CRI 7***.
- Based on the data gathered from the work-practices studies, Singer suggested various criteria for software engineering tools [32]. Some of the criteria suggested by Singer are :
    - Providing search capabilities so that a user can search for relevant information **–*CRI 8.***
    - Display all relevant attributes of an item retrieved and all relationships among items **– *CRI 9.***
    - Keeping track of problem-solving sessions **– *CRI 10.***
    - Include ability to work on a source code of very large size **– *CRI 11***

To improve the effectiveness of Panorama, we have implemented all of the above features. These enhancements are described in further detail in the "Design" chapter.

Chapter 3. Design

In this chapter, we describe the design framework – in this case, the Eclipse development environment and its' architecture, how user interfaces are developed for plug-ins, and the additional libraries needed for our development work. Then, we present the list of added features and their design and implementation details.

3.1. Design framework

3.1.1. Eclipse Plug-in development environment

Eclipse is a software development environment (i.e. an IDE) with an extensible plug-in system. In its default form, the eclipse IDE is used to develop applications in Java using the Java Development Tools (JDT). However, users can add functionalities to their eclipse IDE by installing additional plug-ins. For example, eclipse can be used to develop applications in C, C++, COBOL, Python, Perl, and PHP using appropriate plug-ins.

The Eclipse architecture utilizes the concept of extension points and extensions to allow plug-ins to work with each other. If an existing plug-in wants to provide a facility for other plug-ins to extend or customize its features, it can provide an extension point which declares a contract that extensions (i.e. the other plug-ins) must conform to. These extension point contracts are typically a combination of XML markup and Java interfaces. If a plug-in wants to extend an existing plug-in, it can do so by providing an implementation of one or more of these extension points. Each implementation is called an extension. This design allows Eclipse to achieve loose coupling among various components of the IDE and to extend the existing functionalities in a flexible manner.

3.1.2. Panorama as an Eclipse plug-in

To integrate Panorama with the Eclipse development environment, it has been implemented as a plug-in. It also utilizes the extension points mechanism to modify the default behavior of various existing components.

To develop Panorama as a plug-in, the plug-in development environment (PDE) is used. PDE provides the tools to create, develop, test, debug, and deploy the eclipse plug-ins. Once the plug-in is developed, it can be easily distributed to users by providing an update link.

As mentioned before, if a plug-in wants to extend existing functionality, it can do so by providing an implementation of one or more of the extension points. Aside from implementation of the required interface, the developer must also specify the extension in a plugin.xml file. as per the extension-point schema XML schema file defined in the plugin.xml file. For example, Figure 3 shows the specification in the plugin.xml that defines an extension (the class "panorama.QuickFixer") to the extension-point"org.eclipse.ui.ide.markerResolution".

```
<extension
      point="org.eclipse.ui.ide.markerResolution">
   <markerResolutionGenerator
         class="panorama.QuickFixer">
   </markerResolutionGenerator>
</extension>
```

Figure 3: Panorama Plug-in Extension Point

3.1.3. Additional Plug-ins

In addition to customization of the eclipse IDE by creating extensions, the following external plug-ins were used to implement necessary functionalities.

- Zest: This is an Eclipse Visualization Toolkit developed using SWT/ Draw2D. The library also contains a graph layout package that can provide visualizations in Spring, Tree, Radial, and Grid layouts.
- Eclemma: This is a java code coverage tool for eclipse that helps to identify the lines of code covered during the execution of an application. The tool features include
    - Coverage Overview where the summary of coverage information for java projects is shown down to the method level.

  - Source Highlighting where the tool highlights lines that are fully, partially, and not covered with the colors assigned to each of these categories. These colors can be customized based on user requirements.
  - Coverage Sessions where the tool provides a facility to save the coverage session information to the file system for future use.
- JGraph: This is a open-source, graph drawing program that we used to generate the concern flow graph visualization for offline documentation. These visualizations help in understanding the flow sequence between the stripes for a particular concern.

The plug-ins mentioned above are necessary for Panorama to work and thus are included in the distribution. When an Eclipse IDE tries to install the Panorama plug-in, it will install these additional dependencies if they are not already installed.

3.2. Feature design

The following major features were implemented:

- Concern management
- Concern display
- Code capture
- Offline documentation

In addition, other features (such as Search, Context management, and Distribution) were also developed.

3.2.1. Concern management

Panorama documentation process mainly relies on the wisdom of an application expert who has the knowledge of execution sequence of an application. An expert defines the functionalities of software as “Concerns”, creates “Stripes” defining the pieces of code executed under each concern, and links them in order to create a sequence. A novice developer can use this information to understand the sequence of execution as well as the architecture of an application.

In Panorama, Concerns are maintained in a hierarchical structure. An expert developer can divide a complex functionality into a subsequent set of simple functionalities and document them separately. All these simple functionalities can be shown as children of the complex functionality.

To allow experts to create, modify, delete, and organize concerns, we have implemented two views (Panorama view, Panorama Hierarchy view) that extend the "org.eclipse.ui.views" extension-point. An expert developer can create or edit concerns and stripes using the Panorama view. For the creation of both concerns and stripes, information needs to be provided to define its purpose and parent. During creation of a stripe, it is linked to the corresponding lines of code by selecting them in a code editor. The right pane of the Panorama Hierarchy View shows the relationships between concerns (as in Figure 4).

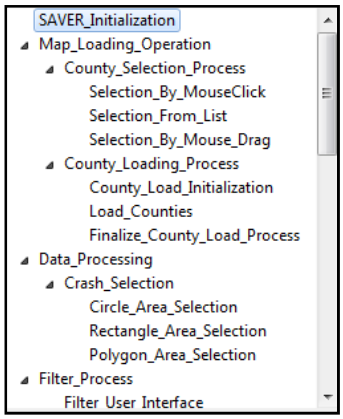

Figure 4: Concern Hierarchy

3.2.2. Concern display

We provide two ways of inspecting a concern – textual and graphical. The Panorama Hierarchy view is used for the text depiction. The Panorama concern visualization view implements the Zest extension point to provide a graphical view.

Textual view

The left pane of the Panorama Hierarchy view shows all the stripes of a selected concern. Stripes are presented in a ScrolledComposite widget provided by the SWT widget library. Once a concern is selected in the Panorama hierarchy view, the stripes in that concern are

displayed with alternating grey and white background which helps in differentiating between the stripes easily (See Figure 5).

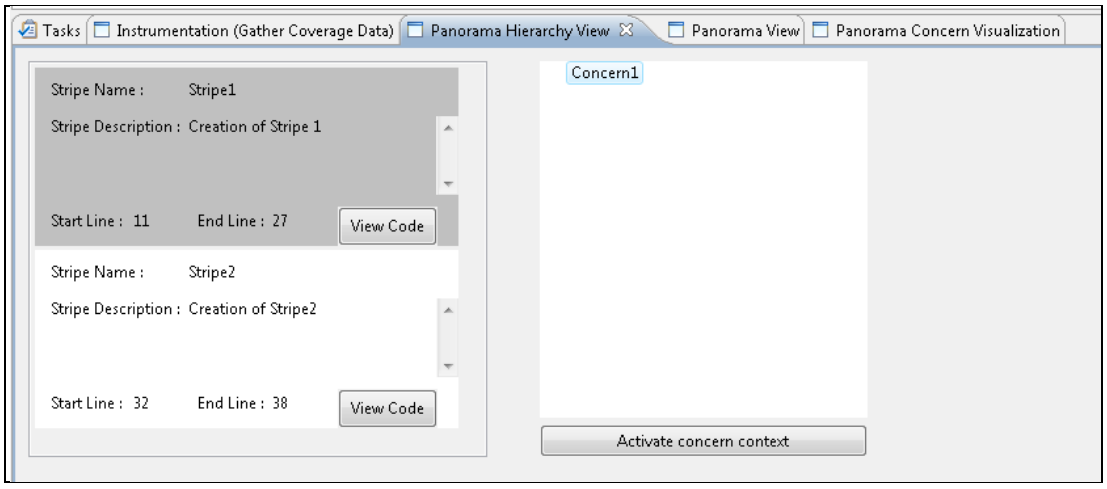

Figure 5: Viewing concern details

Each stripe is directly linked to its corresponding lines of code. To view the code for any stripe, a user can click on the "View Code" button, which will open the corresponding code and adds markers to the editor for easy identification of a stripe (see Figure 6). Once the markers are created, a user can hover the mouse on markers to see the list of stripes that are directly connected to this stripe and navigate between them in a seamless fashion.

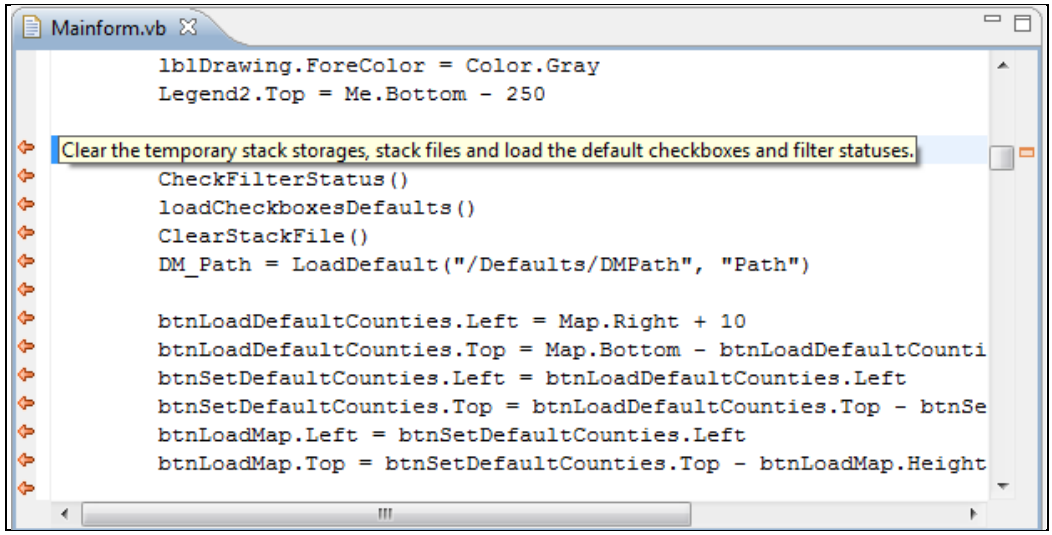

Figure 6: Browsing the code of a stripe

Graphical view

We developed the Panorama Concern visualization view to present the concern information in a graphical format. To do this, we extended the Zest - Eclipse Visualization Toolkit. Once a concern is selected, Panorama will dynamically generate the graph (see Figure 7).

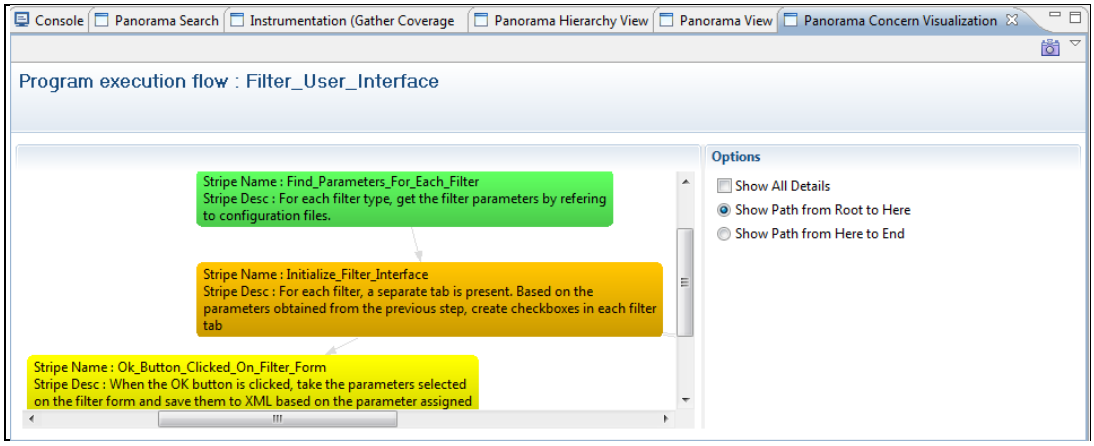

Figure 7: Concern Flowgraph

Users can see concern flow graph in either “Minimal” or “Detailed” mode depending on the amount of information they are willing to view. Here too, every stripe is directly linked to its corresponding piece of code. Double-clicking on a node of the flow graph will open the corresponding code in a code editor and add markers to improve readability (as in Figure 6). Hovering on the markers will provide displays a list of stripes that are directly connected to the selected stripe and allows navigation between stripes in a seamless fashion. There is also a feature to capture snapshots of the current concern flow graph for offline reference.

3.2.3. Code capture

This feature allows the expert to capture all the relevant code for a specific functionality (which can then be documented). To implement this feature, we have provided an instrumentation view (an extension of the org.eclipse.ui.views extension point) and also have integrated Panorama with Eclemma(a java code coverage tool). Eclemma provides a coverage view that lists coverage summary for the projects which can be drilled down to the

method level. It also highlights fully, partly, and not-covered lines with colors that are customizable.

Initially, the application is started in Eclemma coverage mode. To collect the coverage data for a particular functionality, the user can click on “Start Coverage” (See Figure 8) when the corresponding code lines are about to execute. This action resets the coverage information and starts collecting data for the specific functionality. Once all the code related to the functionality has finished execution, the user clicks on the “Store Coverage Data” to save the information in a file. This process can be repeated throughout the execution of an application to gather the coverage information of any desired functionality.

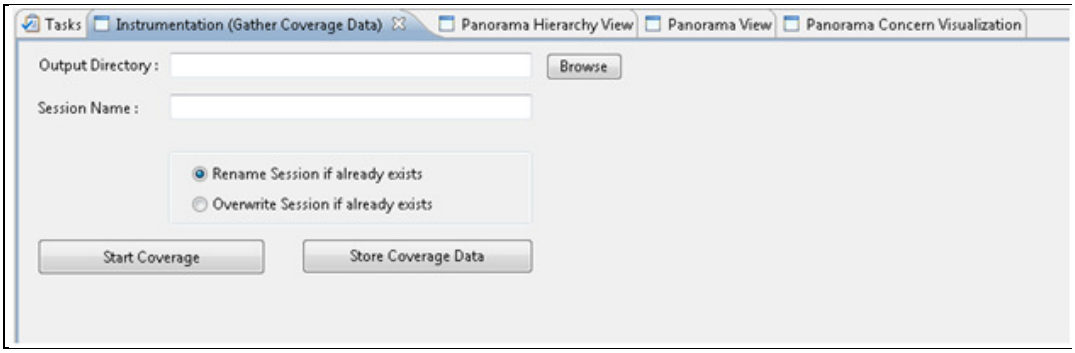

Figure 8: Instrumentation View

Eclemma provides a Coverage view that can be used at a later time to open a saved coverage file and view the corresponding coverage information. Coverage files corresponding to a specific functionality can be opened when desired. The Coverage view provides a facility to drill down to a method level to see the summary of coverage information (See Figure 9).

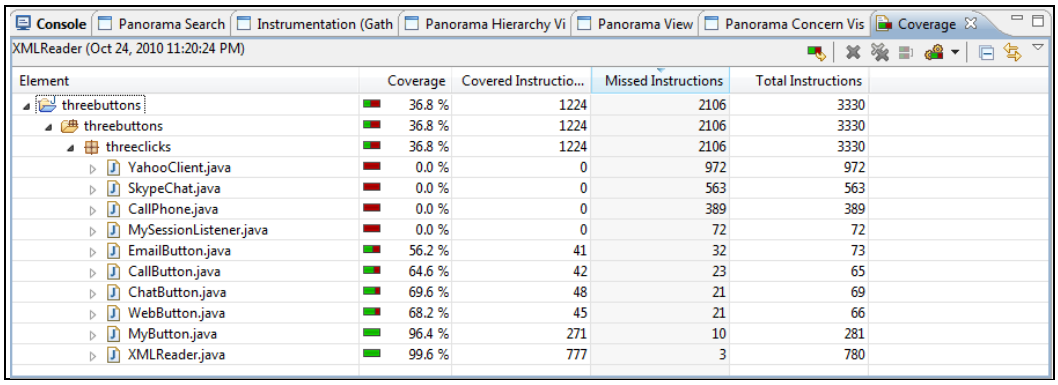

Figure 9: Coverage View

Double clicking on a file or method will open the corresponding java file in an eclipse editor. The editor window will display code with source highlighting (See Figure 10) that distinguishes fully, partially, and not-covered lines. This information can be used to identify the lines of code that should be part of a stripe for a specific concern.

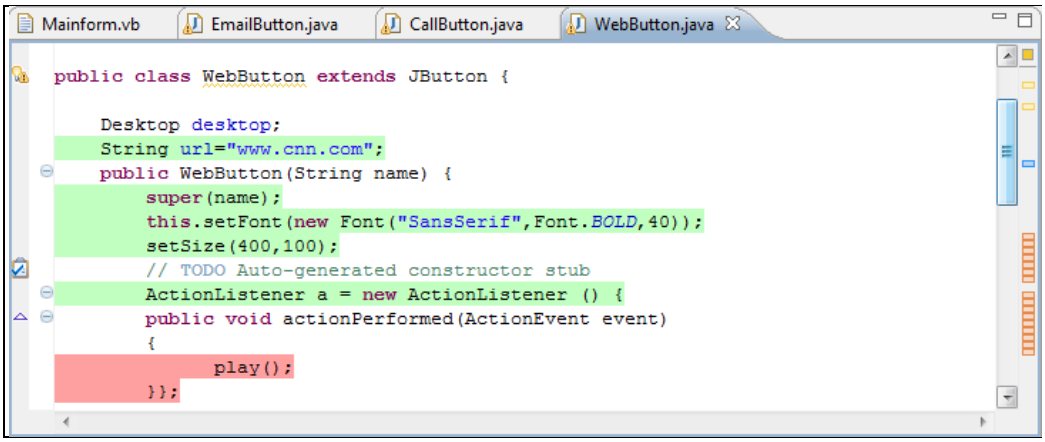

Figure 10: Coverage information highlighting

In this manner, the executed code can be easily converted into stripes and can be linked to create concern flow graphs.

3.2.4. Offline documentation

This feature allows the expert to create javadocs-like documentation for concerns so that existing java developers can easily adapt to it. Essentially, this consisted of generation of

appropriate html files. We used JGraph library to generate the flow graphs for the offline documentation.

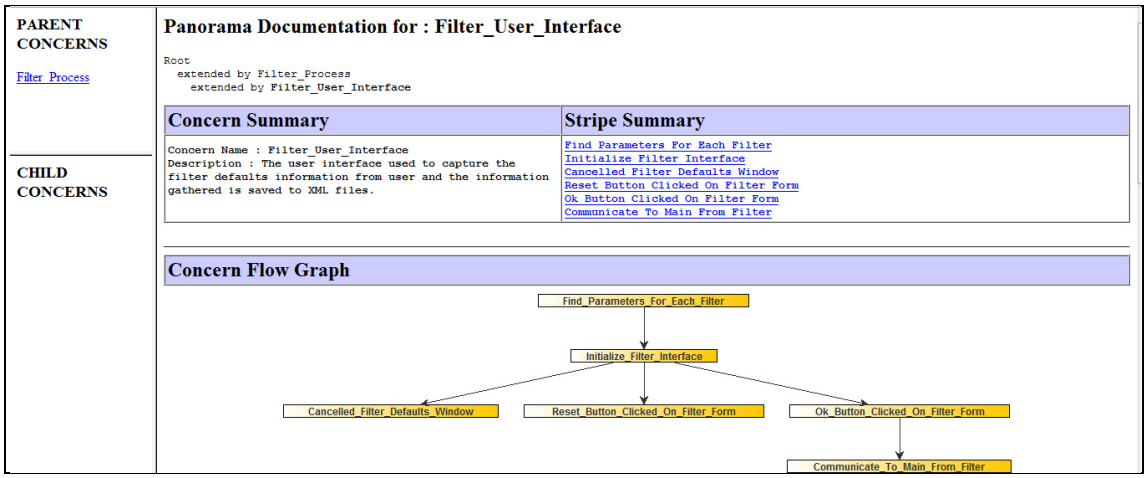

Figure 11: Panorama Documentation Structure

As shown in Figure 11, the Panorama documentation has three sections that are described below:

1. Parent Concerns: This frame on the top-left corner shows the parent(s) of the currently selected concern.

2. Child Concerns: This frame on the bottom-left corner shows the children of the currently selected concern.

3. Documentation window: The frame on the right side shows the details of current concern. This frame has several sections that are described here.

    a. Concern Summary: This section contains the name and description of the concern.

    b. Stripe Summary: This section contains the names of all of the stripes for the selected concern along with hyperlinks to their corresponding stripe details section.

    c. Concern Flow Graph: This section presents the structure of a concern in the form of a flow graph.

d. Stripe Details: This section contains the details of each stripe which includes the stripe description, start line and end line of the stripe, and a hyperlink to the associated code. Users can view a stripe's code by clicking on the "View Code" hyperlink.

3.2.5. Other features

Panorama search

A search view has been implemented to allow users to quickly locate stripes or concerns based on the search criteria. The search view shows a quick overview about the purpose and hierarchy of concerns and stripes. This helps the users in locating relevant information that can be useful during maintenance activities.

Context maintenance

A user can be working on various projects in the eclipse environment and can have many files open at the same time. This will result in information overflow and makes it difficult for the user to focus on a specific task. To avoid this situation, a functionality called the "Concern Context" has been developed to reduce information overload. Once a concern is selected and the concern context feature is activated (See Figure 12), only the files related to the selected concern are kept open and markers are created for the stripes of the concern. All the remaining files are hidden from the user. Once the concern context feature is de-activated, all the windows that were previously open are brought back which allows the user to return to his previous state.

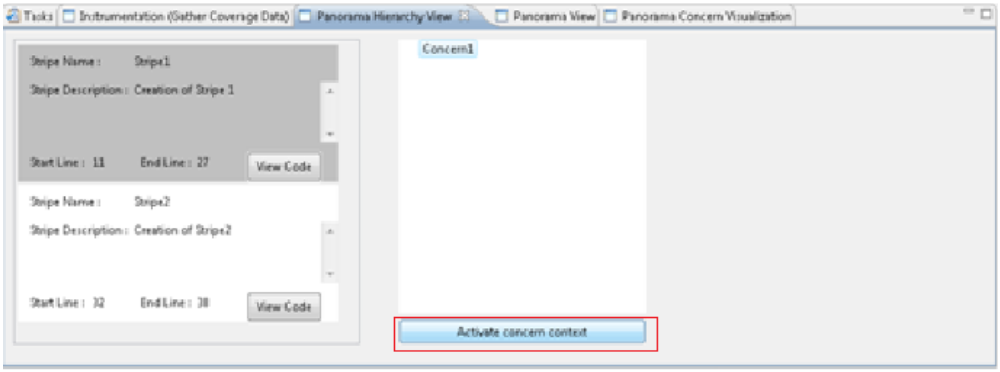

Figure 12: Activating Concern Context

Panorama distribution

The plug-in development environment allows one to create a "feature project" that can be used to distribute a plug-in via an update site. Based on the plug-in information provided in the feature project, eclipse automatically determines the dependencies that need to be installed in the target eclipse environment. A build is created from the feature project and can be hosted at an update site (on network or local folder). Users can point to the update site location provided and follow a simple wizard to install the developed plug-in (See Figure 13).

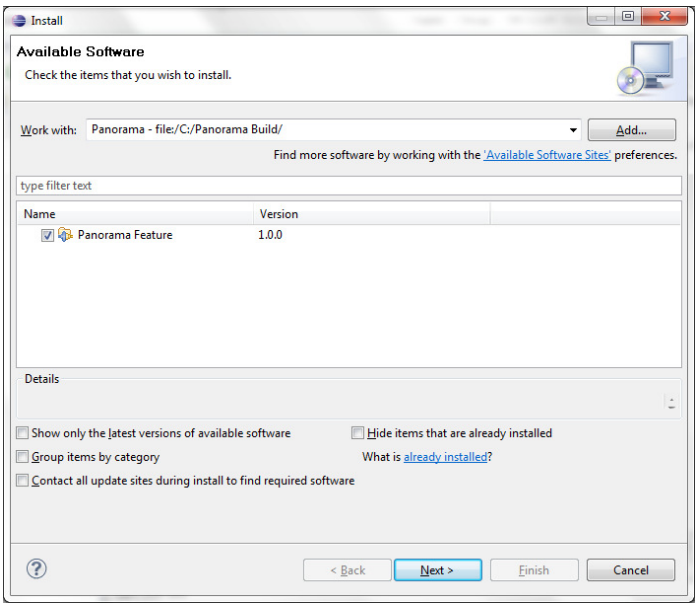

Figure 13: Update site and feature selection

A feature project has been created for Panorama to allow for easy distribution. A local build can be created for distribution to users who would like to use the tool. As shown above, users can point to the local folder and select the features that should be installed.

After implementation and testing of all of the features discussed in this chapter, a case study was conducted to validate the tool. The case study done is explained in detail in the next chapter.

Chapter 4. Case Study and Discussion

In this chapter, we present details of the case study that was done to validate the effectiveness of Panorama. For the case study, we selected a fairly large application that is actively being used and maintained. This application is called SAVER (Statistical Analysis, Visualization, and Exploration Resource) and is a GIS (Geographic Information System) software used by the Iowa Department of Transportation to analyze crashes. SAVER is a VB.NET application with 26,704 executable lines of code and 3,629 comment lines. The case study needed access to an expert developer for documentation purposes and also to developers engaged in day-to-day maintenance activities – and we had access to both for the SAVER project. The author took on the role of expert developer to document SAVER as he had worked on the development of core SAVER functionality for two years.

In the case study, we considered three scenarios – expert knowledge capture, software comprehension in unfamiliar domain, and software comprehension in a familiar domain. For each of these scenarios, we analyzed the workflows and challenges faced by users– and how Panorama fares in comparison to existing tools in resolving these challenges. We have compared Panorama with Mismar, FLAT^3, SEXTANT and Jasper because in our opinion they are the most similar to Panorama in terms of how they help a developer in maintenance tasks (although their goals are different from ours).

In the rest of this chapter, we first describe the details of each scenario and then present our summary of the comparisons.

4.1. Expert knowledge capture

The first scenario considers the workflow of an expert developer who is documenting the code to make it more understandable to the maintainer. In this subsection, we describe the importance and difficulties faced during documentation of cross-cutting concerns, the workflow or process that the expert can follow when using Panorama, the differences with related tools such as Mismar, and finally, additional features that any concern hierarchy management mechanism should incorporate for effective expert knowledge capture.

4.1.1. Importance and challenges

Typically, developers performing maintenance tasks refer to the source code and the associated javadocs documentation to comprehend the working of the software. In some cases, they might even have access to the software's design documentation. However, the focus of most existing documentation is typically an object and its methods – and there is no well-established mechanism by which to document crosscutting behaviors. Thus, existing documentation is somewhat inadequate for maintenance purposes.

Developers make hundreds of decisions during implementation. The reasons for making important decisions are often not documented and these reasons become important when changes have to be made during maintenance activities and for program comprehension. Thus, it is important to provide developers a way to capture their expert knowledge about the application (especially regarding cross-cutting concerns). Capturing this knowledge is important because the person working on a maintenance task may not be the one who actually developed it.

Manual documentation is a laborious and time-consuming process. An expert who has to document a concern's execution sequence should be aware of its entire workflow in order to define the sequence. But even though the expert may know the system, he may be unable to maintain the complete mental model of it [32] – leading to documentation errors. Tools to help with documentation typically provide search mechanisms to help with locating relevant parts of code. There are basically three types of searches: a plain grep-like search, syntax directed search, and semantics based search. Most tools use a grep-like or a syntax directed search. These approaches typically lead to an explosion of information and false positive matches and can actually increase the effort needed for documentation. For example, there may be situations where elements get attached to documentation because they follow a rule or pattern (i.e. syntax directed matches) but are not related to the functionality, which further results in information overload. Also, it is impossible for an automated tool to divine expert's design rationale from the code. For example, it may be possible to obtain a data flow diagram from existing code - but not expert knowledge like design decisions, execution sequence for a cross-cutting concern etc.

Semantics based searches are difficult to automate and our expert-based approach alleviates this problem because the expert knows the details of the code and can quickly locate relevant parts.

4.1.2. The documentation process

Documentation of crosscutting concerns consists of the following tasks:

- Identify crosscutting functionalities and creating associated concerns.
- Searching for existing concerns to avoid duplication.
- Organizing concerns as a hierarchy.
- Finding related code to add to a concern.
- Selection of code and creation of associated stripe.
- Adding, modifying, removing stripes to concerns.
- Linking stripes of a concern (for example as an execution sequence).
- Generating offline documentation for concerns.

Saver documentation proceeded with the creation of a concerns hierarchy (see Figure 14). Important functionalities were chosen and then subdivided into "sub-concerns" and so on. The granularity is left to the users' discretion. This process continued over several sessions.

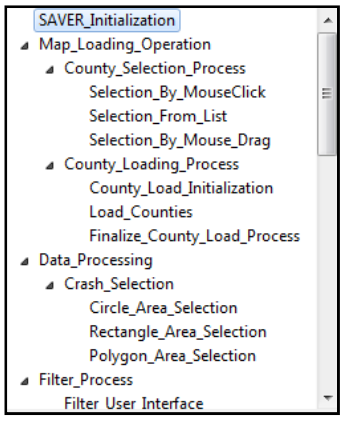

Figure 14: Concern Hierarchy

All the tasks were carried out to create offline documentation (in the form of html files) for SAVER which was then distributed to the SAVER developers.

4.1.3. Comparison with existing tools

Although, Javadocs doesn't provide a way to document cross-cutting concerns, it is the standard way of documenting Java code where annotations by developers are automatically extracted and presented based on class hierarchy. FLAT^3 and SEXTANT cannot be compared with Panorama because their emphasis is more on finding relevant information rather than on documentation of concerns. Similarly, Jasper couldn't be compared in this scenario because its goal is more about grouping information as working sets – rather than documentation.

Mismar is similar to Panorama in that the structure of documentation is manually created by selecting relevant elements and attaching information to these elements. Thus, additional knowledge like design decisions can be easily documented and unnecessary information can be avoided. Mismar allows a strictly sequential linking of elements belonging to a concern. Unlike Mismar, Panorama allows arbitrary linking of stripes to create a flow graph.

As described before, manual documentation is a laborious and time-consuming process. An expert who has to document a concern's execution sequence should be aware of its entire workflow in order to define the sequence. Unlike Mismar, Panorama provides an automated code-capture mechanism that helps to alleviate this problem. This will help the expert to create a mental model (see Chapter 2) at the concern level in order to convert it into concern documentation. Although, a code coverage feature is available in FLAT^3, their goal is to reduce the context for code search rather than for documentation.

4.1.4. Lessons learnt

This scenario helped us identify a few areas that can be further improved.

During the documentation process of SAVER, many stripes and concerns were created. When selecting a stripe, Panorama presents a list of all existing stripes as potentially parents of the current stripe. The expert has to wade through this list to select the parent. The difficulty of documenting concerns increased over time because of information overload. A

context maintenance facility that shows only the existing stripes of the selected concerns would help to alleviate this problem.

We also realized that the current code capture mechanism provides all the code segments that was executed for realization of a specific functionality, but does not provide any information about the sequence in which the code was executed. This information would be useful to allow linking between stripes and understand the concern flow.

To summarize, concern documentation tools should include a context management mechanism to reduce information overload for the documenter and code capture tools should provide information about the sequence in which the code is executed.

4.2. Understanding software when the domain is unfamiliar

The second scenario considers the workflow when a developer from an unfamiliar domain is trying to understand the software to perform maintenance tasks. In this subsection, we will describe the approach he may follow, the challenges faced during this process, the workflow he can follow when using Panorama, and finally the differences with related tools.

4.2.1. Comprehension process

If the developer is from an unfamiliar domain, he may not be able to recognize the plans (See chapter 2) in the software. Thus, he may use bottom-up approach by first understanding individual lines of code and connecting them to create a big picture. To help a developer in this process, existing IDEs have the feature of call hierarchy which when executed on a method/class will provide the information about which methods/classes may call it. This can be used in creating the big picture of the feature/functionality from code level.

4.2.2. Challenges

When a developer is trying to understand an application from a different domain, one of the major challenges he faces is to figure out where to start. Because he doesn't recognize any plans in this domain, finding the starting point for a maintenance task becomes difficult. As mentioned before, he can use the call hierarchy feature provided by the modern IDEs. But the problem with them is that they provide a lot of information. When the developer is trying

to figure out methods/classes related to a maintenance task, the call hierarchy provides methods/classes from the entire project/workspace which can be enormous. This causes the problem of information overload. After the developer figures out where to start, another challenge that is faced is to construct the domain model from program model. As the complexity of software increases, understanding the code segments and connecting them to create a big picture becomes difficult.

To summarize, a developer from unfamiliar domain may need help in the following areas:

- Finding the starting point where he can start understanding the code
- Reducing the information overload
- Helping to create the domain model from program model

4.2.3. Panorama workflow

To help a developer from above problems, Panorama provides the facility to locate the starting point and to construct domain model and program model (See Chapter 2) using the documentation provided. Initially, an expert developer will generate Panorama documentation by recording how the concerns are connected to each other (domain model) and the execution sequence for every concern (program model). A new developer can use the "Panorama search" feature to figure out the starting point. Once he figures out which concern to start with, he can visualize it using "concern visualization" feature to understand how a concern executes and the implementation details (program knowledge). This helps in creating program model. The "concern hierarchy" feature helps in understanding the domain knowledge attached to each concern and how the concerns are connected to each other. This information, along with the program knowledge previously gained, will help in constructing the domain model of the software.

4.2.4. Comparison of existing tools and Panorama

To reduce the overload, user should only be provided the information relevant to the task at hand. Panorama helps in this situation because documentation is based on concerns. For every concern the developer is looking at, he will only be presented the code segments

related to that concern. Panorama also has the facility to hide the unnecessary files when analyzing a concern so that only the relevant ones are opened. Mismar also helps in reducing information because their documentation is also based on concerns. SEXTANT helps by providing the facility to start at one method and to expand the context by selecting the ones that are relevant to the concern. But because it is an exploration tool, understanding if a method is related to the concern is left to the developer and thus cannot help in understanding the code. Jasper reduces the information overload because it groups information in the form of working sets. Thus it helps in seeing only the information relevant to the task at hand. Even though FLAT^3 has a facility to reduce information overload using code coverage, it couldn't be compared in this scenario because it is more of a code search tool and their goal is to reduce the search context but not to help in understanding software.

To help a developer in finding the starting point, Panorama provides the search facility. No evidence was found to support that Mismar has a facility to search in its documentation. Even though FLAT^3 helps in searching code by probability, it may not be helpful in this scenario because developer may be new to the programming language and thus may not know what to search for. In SEXTANT, the starting point is found by search with regular expressions. This can help in finding where to start but may become difficult if the code terminology is different from that of domain. In Jasper, all information related to a concern is presented to user in multiple windows. But no evidence was found to support the existence of a search mechanism to find the right concern for a task at hand.

To help a developer in constructing domain model from program model, Panorama provides concern hierarchy with domain knowledge attached to each concern. Mismar also helps in constructing domain model because it has the concern hierarchy with information attached to concerns. FLAT^3, SEXTANT and Jasper couldn't be compared in this scenario because their goals are different and they do not focus on capturing knowledge.

4.3. Understanding software when the domain is familiar

The third scenario considers the workflow when a developer from a familiar domain is trying to understand the software to perform maintenance tasks. In this subsection, we will

describe the approach he may follow, the challenges faced during this process, the workflow he can follow when using Panorama, and finally the differences with related tools.

4.3.1. Comprehension process

When a developer from familiar domain is trying to understand the software, he has the idea of how things work in this domain. Because of this, he may follow top-down approach where in, he will first understand the high level structure of the software and then understand how it is implemented at code level. To do this, he will first make a hypothesis and try to look for beacons to confirm the hypothesis made (See chapter 2). Once the hypothesis is confirmed, he will make more concrete hypothesis and repeat the process till he understands the software at domain model. Once that is done, he will go to each component separately and understand how it is implemented in code. To help a developer in this process, existing IDEs provide the facility of break points which can be used to confirm the hypothesis. In case of Javadoc, the UML extensions can be used to automatically generate the UML diagrams from the documentation which can be used to understand the high level interactions and data flow.

4.3.2. Challenges

When a developer tries to construct the domain model, he may make a hypothesis and searches for beacons to confirm it. This process of finding beacons may become difficult when the size of code increases or the domain plans (See chapter 2) change. During this process, he may also get confused if a lot of irrelevant information is provided to him. Once the developer tries to understand the implementation of an application, figuring out why the code is written that way can be a challenge because he may not be aware of the development environment or the programming language.

To summarize, a developer from familiar domain may need help in the following areas:

- Finding beacons
- Understanding the code
- Reducing information overload

4.3.3. Panorama workflow

To help a developer from above problems, Panorama provides the facility of concern hierarchy and concern visualization. Because the software is documented in the form of a concern hierarchy, a developer can navigate through the hierarchy and use the knowledge attached to each concern to construct the domain model. Also, a developer can look for beacons in the knowledge attached to the concerns to confirm his hypothesis. If the developer couldn't find information in the concern knowledge, he can use Panorama search facility to extend the search to stripe level knowledge. Once the domain model is constructed, developer can select each concern and visualize its execution flow to understand how it is implemented. Users can directly navigate from the nodes in concern graph to the corresponding code. This simplifies the process of understanding the implementation. During this process, user can hide unnecessary files by activating the concern context facility.

4.3.4. Comparison of existing tools and Panorama

To help a developer in finding beacons, Panorama provides concern hierarchy and search facility. Mismar also provides concern hierarchy but no evidence was found to support the existence of a search facility. In FLAT^3, search feature is used to find the code relevance with a probability. This is useful in finding information at the code level but makes it difficult at high level. In SEXTANT, information can be searched with regular expression but this can become difficult if the terminology used in code is different from the domain. In Jasper, no evidence was found to support the existence of a concern hierarchy and search facility. Text notes can be attached to the working set which can be used to add information about beacons. But the complexity of this technique increases with an increase in the number of working sets and their elements.

To help a developer in understanding the code, Panorama provides concern visualization feature. Mismar allows attaching knowledge to every step in the guide which can help in understanding the code behavior but no evidence was found to support the existence of a sequence diagram to understand the guide. Instead, the guide is displayed as a list of steps [6] which may not always be practical. In Jasper, text notes can be added to the working set which can be used to document the behavior of code. But no evidence was found to support

the existence of a diagrammatic representation of links between the elements of a working set. In SEXTANT, understanding the code is left to the developer and thus cannot help in solving this problem. FLAT^3 cannot be compared in this context because it is more about searching relevant code.

To reduce information overload, Panorama uses concern based documentation technique and provides the context maintenance feature. The details and comparison of the remaining tools remains the same as explained in the previous scenario.

4.4. Results

The following table shows a comparison of features between Panorama and the exiting tools. Based on the following comparison, we can say that Panorama contains most of the features when compared with other tools.

Table 1: Comparison of features provided by tools

Tool	Type of tool	Concern Hierarchy	Doc'n	Visualization	Search	Concern Context	Code Coverage
Jasper	Working Set Maintenance					✓	
FLAT^3	Code Search			✓	Code		✓
Mismar	Documentation	✓	IDE based				
SEXTANT	Exploration	✓		✓	Code	✓	
Panorama	Documentation	✓	IDE & Offline	✓	Doc'n	✓	✓

The following table shows a comparison of the criteria satisfied by Panorama and existing tools. Based on the following comparison, we can see that Panorama is successful in satisfying most of the criteria when compared to other tools. The tools compared here didn't seem to support some of the criteria because of the differences in their goals. Also the way in which the criteria are satisfied by a tool is different from others because of the goals and assumptions made when building it.

Some of the cells in this table were marked "?" because decision couldn't be made either because of unavailability of information or the inability to judge the situation. Mismar is marked "?" for criteria 8 ([Mismar,CRI8]) because no evidence was found to support the existence of a search capability. Same is the reason for criteria 2 ([Mismar,CRI2]) because if the search capability exists, even Mismar can support multiple comprehension strategies. Jasper is marked "?" for criteria 9 ([Jasper,CRI9]) because, though relationships can be provided between the elements of a concern by adding text notes to working set, this process becomes complicated as the number of elements increase in a working set. FLAT^3 and SEXTANT are marked "?" for criteria 10 ([FLAT^3,CRI10] and [SEXTANT,CRI10]) because, even though these tools support the saving of information which can be imported back for analysis, no evidence was found if they explicitly keep track of previous sessions for immediate usage. Even Panorama is marked "?" for criteria 10 ([Panorama,CRI10] because it can explicitly keep track of the code coverage sessions only after the saved sessions are manually imported back into the environment.

Table 2: Comparison of criteria satisfied by tools

Tool	Criteria										
	CRI1	CRI2	CRI3	CRI4	CRI5	CRI6	CRI7	CRI8	CRI9	CRI10	CRI11
Jasper			✓						?		✓
FLAT^3			✓	✓	✓			✓		?	✓
Mismar	✓	?	✓	✓	✓	✓		?	✓		✓
SEXTANT		✓	✓	✓	✓	✓	✓	✓	✓	?	✓
Panorama	✓	✓	✓	✓	✓	✓	✓	✓	✓	?	✓

Because FLAT^3 and SEXTANT are software exploration tools, they have code search feature which can be used to find the code matching a pattern. We realized that having this feature in Panorama will be helpful during the search process. Currently, a developer will search for relevant code segments based on the documentation attached to the stripes and use that to understand code. If this feature is present in Panorama, users can find code segments similar to a required structure and identify the code segments based on probability.

As part of case study, the Panorama documentation generated for SAVER was provided to its client and developers who were asked to provide their feedback on Panorama. We got a positive response from them that it would be helpful in maintenance tasks. They also provided suggestions on improving the offline documentation and these improvements will be included in future versions of Panorama.

4.5. Limitations

Because of the large size of SAVER software, a lot of time was devoted to documenting SAVER and understanding the capabilities and difficulties of the process. Because of a limited time factor, a study couldn't be conducted to validate the effectiveness of Panorama using an experimental group. Panorama is validated in this case study by comparing with existing tools in terms of the features provided and criteria satisfied. As Panorama documentation is available for a large application now, we plan to conduct a study using this documentation with a larger study group than the previous Panorama study conducted by our team [23].

Chapter 5. Conclusion and Future Work

Panorama supports software maintenance by presenting concern relevant information in an efficient manner. It helps in understanding software by reducing information overload and presenting information in the form of a sequence diagram. The enhanced user interface makes it easy to navigate between the views and also from stripes to their corresponding code. Panorama also helps in the documentation process by providing code-tracing information, which a developer can convert into concern documentation. The offline documentation generated by Panorama can be used to understand an application without using Panorama user interface or without using the Eclipse development environment.

In a case study done to validate its effectiveness, the documentation generated on SAVER software was given to its client and developers. They provided a feedback that the tool will be helpful in supporting maintenance task. We also comparing Panorama with existing tools in terms of the features provided and the desired features of a successful maintenance tool. The results should be further validated by conducting a study that compares actual use of Panorama versus existing tools on typical maintenance activities.

5.1. Future work

1. A study should be conducted using an experimental group to evaluate the effectiveness of Panorama and its usability. This can provide a direction for further research to improve Panorama.
2. In Eclemma, the entire code tracing information is obtained by a single instance of plug-in. This may lead to confusion when code coverage is done on an application with multiple threads running in parallel. A mechanism to gather coverage information on per-thread-basis is required to understand how the threads are interacting in parallel and to improve software comprehension.
3. In addition to code coverage information, the sequence in which code executes should also be provided to developer.
4. Context maintenance feature should be extended to documentation process to make it easy for an expert developer to document software.

5. When a piece of code is selected, the user should be able to generate a graph that combines all the concerns containing the selected code.
6. Integration with eclipse debug API will improve the abilities of Panorama in understanding the application in runtime. For example, the ability to create breakpoints automatically at the stripe locations will be helpful in debugging the application at concern level.
7. The ability to search code using pattern matching will help an expert developer in finding relevant code segments that may need to be documented.
8. Version controlling the information stored by Panorama will help in tracking the changes done to Panorama documentation and the application through its life cycle.

References

[1] AspectJ Development Tools (AJDT), http://www.eclipse.org/ajdt (accessed: September 8, 2010).

[2] Bryant, A., Catton, A., De Volder, K., and Murphy, G. C., "Explicit programming", In Proceedings of the 1st international Conference on Aspect-Oriented Software Development (Enschede, The Netherlands, April 22 - 26, 2002), AOSD '02, ACM, New York, NY, 10-18. DOI= http://doi.acm.org/10.1145/508386.508389.

[3] Coblenz, M. J., Ko, A. J., and Myers, B. A., "JASPER: an Eclipse plug-in to facilitate software maintenance tasks", In Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology Exchange (Portland, Oregon, October 22 - 23, 2006), eclipse '06, vol 195, ACM, New York, NY, 65-69. DOI=http://doi.acm.org/10.1145/1188835.1188849.

[4] Corritore, C. L., and Wiedenbeck, S., "An exploratory study of program comprehension strategies of procedural and object-oriented programmers", International Journal of Human-Computer Studies, Jan 2001.

[5] Das, S., Lutters, W. G., and Seaman, C. B., "Understanding documentation value in software maintenance", In Proceedings of the 2007 Symposium on Computer Human interaction For the Management of information Technology(Cambridge, Massachusetts, March 30 - 31, 2007), CHIMIT '07, ACM, New York, NY, 2. DOI= http://doi.acm.org/10.1145/1234772.1234790.

[6] Dagenais, B. and Ossher, H. , "Aiding evolution with concern-oriented guides", In Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution (Vancouver, British Columbia, Canada, March 12 - 16, 2007), LATE '07, ACM, New York, NY, 4. DOI= http://doi.acm.org/10.1145/1275672.1275676.

[7] Dagenais, B. and Ossher, H., " Mismar: A New Approach to Developer Documentation", In Companion To the Proceedings of the 29th international

Conference on Software Engineering (May 20 - 26, 2007), International Conference on Software Engineering, IEEE Computer Society, Washington, DC, 47-48. DOI= http://dx.doi.org/10.1109/ICSECOMPANION.2007.51.

[8] Dagenais, B. and Ossher, H., "Guidance through active concerns", In Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology Exchange (Portland, Oregon, October 22 - 23, 2006), eclipse '06, vol. 195. ACM, New York, NY, 60-64. DOI= http://doi.acm.org/10.1145/1188835.1188848.

[9] Diver: Dynamic Interactive Views for Reverse Engineering | The CHISEL Group, http://thechiselgroup.org/diver (accessed: September 8, 2010).

[10] Documenting Pattern Use in Java Programs, In Proceedings of the international Conference on Software Maintenance (Icsm'02) (October 03 - 06, 2002), ICSM, IEEE Computer Society, Washington, DC, 230.

[11] Eichberg, M., Haupt, M., Mezini, M., and Schafer, T., "Comprehensive Software Understanding with SEXTANT", In Proceedings of the 21st IEEE international Conference on Software Maintenance (September 25 - 30, 2005), ICSM, IEEE Computer Society, Washington, DC, 315-324. DOI= http://dx.doi.org/10.1109/ICSM.2005.32.

[12] Erdem, A., Johnson, W., and Marsella, S. , "Task Oriented Software Understanding", In Proceedings of the 13th IEEE international Conference on Automated Software Engineering (October 13 - 16, 1998), Automated Software Engineering, IEEE Computer Society, Washington, DC, 230.

[13] Gross, P. and Kelleher, C. 2010. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. J. Vis. Lang. Comput. 21, 5 (December 2010), 263-276. DOI= http://dx.doi.org/10.1016/j.jvlc.2010.08.002.

[14] Janzen, D. and De Volder, K., "Navigating and querying code without getting lost", In Proceedings of the 2nd international Conference on Aspect-Oriented Software

Development (Boston, Massachusetts, March 17 - 21, 2003), AOSD '03, ACM, New York, NY, 178-187. DOI= http://doi.acm.org/10.1145/643603.643622.

[15] JTourBus – SE – Twiki, http://www.inf.fu-berlin.de/w/SE/JTourBus (accessed: September 8, 2010).

[16] Karahasanović, A., Levine, A. K., and Thomas, R., "Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative study". J. Syst. Softw. 80, 9 (September. 2007), 1541-1559. DOI=http://dx.doi.org/10.1016/j.jss.2006.10.041.

[17] Kersten, M. and Murphy, G. C. , "Mylar: a degree-of-interest model for IDEs", In Proceedings of the 4th international Conference on Aspect-Oriented Software Development (Chicago, Illinois, March 14 - 18, 2005), AOSD '05, ACM, New York, NY, 159-168. DOI= http://doi.acm.org/10.1145/1052898.1052912.

[18] Kersten, M. and Murphy, G. C. , "Using task context to improve programmer productivity", In Proceedings of the 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering (Portland, Oregon, USA, November 05 - 11, 2006), SIGSOFT '06/FSE-14. ACM, New York, NY, 1-11. DOI= http://doi.acm.org/10.1145/1181775.1181777.

[19] Ko, A. J., Aung, H., and Myers, B. A. , "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks", In Proceedings of the 27th international Conference on Software Engineering(St. Louis, MO, USA, May 15 - 21, 2005), ICSE '05, ACM, New York, NY, 126-135. DOI= http://doi.acm.org/10.1145/1062455.1062492.

[20] Mayrhauser, A. v. and Vans, A. M., "Program understanding needs during corrective maintenance of large scale software", In Proceedings of the 21st international Computer Software and Applications Conference (August 11 - 15, 1997), COMPSAC, IEEE Computer Society, Washington, DC, 630-637.

[21] Majid, I. and Robillard, M. P. , “NaCIN: an Eclipse plug-in for program navigation-based concern inference”, In Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology Exchange (San Diego, California, October 16 - 17, 2005), eclipse '05, ACM, New York, NY, 70-74. DOI= http://doi.acm.org/10.1145/1117696.1117711.

[22] Pauw, W. D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., and Yang, J., “Visualizing the Execution of Java Programs”, In Revised Lectures on Software Visualization, international Seminar (May 20 - 25, 2001). S. Diehl, Ed. Lecture Notes In Computer Science, vol. 2269. Springer-Verlag, London, 151-162.

[23] Renish, P., “An experimental study of the effectiveness of Panorama as a maintenance tool”, 2009, Iowa State University.

[24] Robillard, M. P., Coelho, W., and Murphy, G. C., “How Effective Developers Investigate Source Code: An Exploratory Study”, IEEE Trans. Softw. Eng. 30, 12 (December 2004), 889-903. DOI= http://dx.doi.org/10.1109/TSE.2004.101.

[25] Robillard, M. P. and Murphy, G. C., “FEAT: a tool for locating, describing, and analyzing concerns in source code”, In Proceedings of the 25th international Conference on Software Engineering (Portland, Oregon, May 03 - 10, 2003), International Conference on Software Engineering. IEEE Computer Society, Washington, DC, 822-823.

[26] Robillard, M. P. and Weigand-Warr, F., “ConcernMapper: simple view-based separation of scattered concerns”, In Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology Exchange (San Diego, California, October 16 - 17, 2005), eclipse '05, ACM, New York, NY, 65-69. DOI= http://doi.acm.org/10.1145/1117696.1117710.

[27] Ruven Brooks, “Towards a theory of the comprehension of computer programs”, International Journal of Man-Machine Studies, pp 543-554, vol 18, 1983.

[28] Savage, T., Revelle, M., and Poshyvanyk, D., “FLAT^3: feature location and textual tracing tool”, In Proceedings of the 32nd ACM/IEEE international Conference on

Software Engineering - Volume 2 (Cape Town, South Africa, May 01 - 08, 2010), ICSE '10, ACM, New York, NY, 255-258. DOI= http://doi.acm.org/10.1145/1810295.1810345.

[29] Schafer, T., Eichberg, M., Haupt, M., and Mezini, M., "The SEXTANT Software Exploration Tool", IEEE Trans. Softw. Eng. 32, 9 (September 2006), 753-768. DOI= http://dx.doi.org/10.1109/TSE.2006.94.

[30] Sharon, D.,"Meeting the Challenge of Software Maintenance", IEEE Softw. 13, 1 (January 1996), 122-126. DOI= http://dx.doi.org/10.1109/52.476304.

[31] Singer, J., Elves, R., and Storey, M., "NavTracks: Supporting Navigation in Software", In Proceedings of the 13th international Workshop on Program Comprehension (May 15 - 16, 2005), IWPC, IEEE Computer Society, Washington, DC, 173-175. DOI= http://dx.doi.org/10.1109/WPC.2005.25.

[32] Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N., "An examination of software engineering work practices", In Proceedings of the 1997 Conference of the Centre For Advanced Studies on Collaborative Research (Toronto, Ontario, Canada, November 10 - 13, 1997). J. H. Johnson, Ed. IBM Centre for Advanced Studies Conference. IBM Press, 21.

[33] Software Visualization Tools: Survey and Analysis, In Proceedings of the 9th international Workshop on Program Comprehension (May 12 - 13, 2001), IWPC, IEEE Computer Society, Washington, DC, 7.

[34] Soloway, E. and Ehrlich, K., "Empirical studies of programming knowledge", IEEE Transactions on Software Engineering, pp 595-609, SE-10(5), September 1984.

[35] Spinellis, D., "Code Documentation", IEEE Software. 27, 4 (July 2010), 18-19. DOI= http://dx.doi.org/10.1109/MS.2010.95.

[36] Steven P. Reiss and Manos Renieris, "The BLOOM Software Visualization System" in Software Visualization - From Theory to Practice, MIT Press 2003.

[37] Storey, M. D., Wong, K., and Muller, H. A., “How Do Program Understanding Tools Affect How Programmers Understand Programs”, In Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97) (October 06 - 08, 1997), WCRE, IEEE Computer Society, Washington, DC, 12.

[38] Storey, M., “Theories, Methods and Tools in Program Comprehension: Past, Present and Future”, In Proceedings of the 13th international Workshop on Program Comprehension (May 15 - 16, 2005), IWPC, IEEE Computer Society, Washington, DC, 181-191. DOI= http://dx.doi.org/10.1109/WPC.2005.38.

[39] Volker S., and Roland W., “ProgDOC -- A New Program Documentation System”, In Proc. Perspectives of System Informatics, Lecture Notes in Computer Science volume 2890, pp. 438--449, 2003.

[40] Workshop on Modeling and Analysis of Concerns in Software (MACS 2005) 16 May 2005, St. Louis, MO, USA.

[41] Zou, L. and Godfrey, M. W., “Understanding interaction differences between newcomer and expert programmers”, In Proceedings of the 2008 international Workshop on Recommendation Systems For Software Engineering (Atlanta, Georgia, November 09, 2008), RSSE '08, ACM, New York, NY, 26-29. DOI= http://doi.acm.org/10.1145/1454247.1454256.

[42] Dmitry Kirilov, “PANORAMA – a tool to deal with multiple decompositions of a software system”, 2008, Iowa State University.

Appendix: Tabular Comparison of Related Tools *

Functionality	FLATTT	Mismar	Jasper	SEXTANT	FEAT	JQuery	Panorama
Finding information for concern creation							
Facility to pattern search or Query the code	✓			✓		✓	
Syntax analysis (dependency analysis)				✓	✓	✓	
Expert user defined		✓	✓				✓
Code instrumentation	✓						✓
Granularity							
Class / Method	Irrelevant (Search tool)	✓	✓	✓	✓	✓	✓
Arbitrary segments of code		✓	✓				✓
Control							
Ability to add or edit elements	Irrelevant (Search tool)	✓	✓	✓	✓	✓	✓
Ability to add extra information		✓	✓				✓
Comparison at concern level							
Organization structure possible	Unknown	Hierarchical	Flat	Unknown	Hierarchical	Hierarchical	Hierarchical
Expand/collapse nodes		✓			✓	✓	✓
Global view of stripes	✓						✓
Comparison at code element level							
Organization structure possible	Unknown	Flat only		Graph		Hierarchical	Graph
Expand/collapse nodes in visualization				✓		✓	
Comparison of other functionalities							
Switch seamlessly between visualization and code				✓		✓	✓
Concurrent display of related code segments			✓				✓
Avoid information overload	✓	✓	✓	✓	✓	✓	✓
Ability to search in concern knowledge							✓
Cross Document support	✓	✓	✓	✓		✓	✓
Storage of information	✓	✓	✓	✓	✓	✓	✓

* Some of the functionalities marked as unavailable in the comparison either because of the difference in the tool's goal or unavailability of information to confirm the functionality.