PROGRAM NAVIGATION ANALYSIS USING MACHINE LEARNING

by

Punit Agrawal

School of Computer Science
McGill University, Montreal

September 2008

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Master of Science

Abstract

Developers invest a large portion of their development time exploring program source code to find task-related code elements and to understand the context of their task. The task context is usually not recorded at the end of the task and is forgotten over time. Similarly, it is not possible to share the task context with other developers working on related tasks. Proposed solutions to automatically record the summary of the code investigation suffer from methodological limitations related to the techniques and the data sources used to generate the summary as well as the granularity at which it is generated.

To overcome these limitations, we investigate the use of machine learning techniques, in particular decision tree learning, to predict automatically the task context from session navigation transcripts obtained from developers performing tasks on the source code. We conducted a user study to collect navigation transcripts from developers engaged in source code exploration tasks. We used the data from the user study to train and test decision tree classifiers. We compared the decision tree algorithm with two existing approaches, and found that it compares positively in most cases. Additionally, we developed an Eclipse plug-in that generates automatically a developer session summary using the decision tree classifier learned from the data collected during the user study. We provide qualitative analysis of the effectiveness of this plug-in.

Résumé

Les développeurs de logiciels investissent une grande partie de leur temps à explorer le code source pour trouver des éléments du code reliés à leurs tâches, et aussi pour mieux comprendre le contexte de leur tâche. Le contexte de leur tâche n'est généralement pas enregistrée à la fin de leur séance d'exploration de code et est oublié au fil du temps. De même, il n'est pas possible de partager le contexte de leur tâche avec d'autres développeurs travaillant sur des tâches reliées. Les solutions proposées pour enregistrer automatiquement le résumé de leur exploration du code souffrent de limitations méthodologiques liées aux techniques et aux sources de données utilisées pour générer le résumé, ainsi qu'à la granularité à laquelle il est généré.

Pour surmonter ces limitations, nous étudions l'emploi de techniques d'apprentissage machine, en particulier l'arbre de décision d'apprentissage, pour prévoir automatiquement le contexte de la tâche à partir des transcriptes de navigation d'une session d'exploration de code du développeur. Nous avons effectué une étude de cas afin de recueillir des transcriptions de navigation générés par des développeurs lors de l'exploration du code source. Nous avons utilisé les données de cette étude pour tester les classifications de l'arbre de décision. Nous avons comparé l'algorithme à arbre à décision avec deux approches existantes, et avons démontré que cette nouvelle approche se compare favorablement dans la plupart des cas. Additionnellement, nous avons développé un plug-in Eclipse qui génère automatiquement un résumé d'une session d'exploration de code par le développeur. Ce plug-in utilise un "classificateur arbre de décision" généré à partir des données collectées au cours de l'étude de cas. Nous fournissons une analyse qualitative de l'efficacité de ce plug-in.

Acknowledgments

I would like to thank, Martin Robillard and Doina Precup, for their supervision and guidance in the research leading up to this thesis. They were always available when I needed them and helped focus my efforts towards the end goal. Without their support, it would not be possible to complete this thesis and the work it represents. I would like to thank my parents and sister for encouraging me to pursue my academic interests and supporting me in every way possible, my wife for the coffee when I worked late and for the times she put my work ahead of us. Additionally, I would like to thank my colleagues from SWEVO (Bart and Jan), from the lab (Xi, Maja, and Dustin), and friends at the university (Imad and Omar) for keeping my spirits high when the going was tough. Last but no the least, I would like to thank Jasmine, Arun, Aniroodh, Gaurav, Abhishek, and Zahid for making me feel at home in Montreal.

Contents

Abstract **i**

Résumé **ii**

Acknowledgments **iii**

Contents **iv**

List of Figures **vi**

List of Tables **vii**

1 Introduction **1**

1.1 Motivation 1

1.2 Contribution 4

2 Related Work **6**

2.1 Empirical Studies 6

2.2 Program Navigation Analysis 8

2.3 Other Techniques 10

2.4 Machine Learning in Software Engineering 11

3 User Study **14**

3.1 Methodology 14

3.2 Target System 16

3.3 Target Concerns 17
3.4 Transcript Description 18
3.5 Data Transformation 20

4 Experiment Methodology and Results **23**
4.1 Machine Learning 23
4.2 Machine Learning for Navigation Analysis 27
4.2.1 Decision Trees 28
4.3 Experimental Methodology 31
4.4 Results 33
4.4.1 Comparison with Nacin 40

5 Tool and Evaluation **43**
5.1 Tool 43
5.2 Evaluation Study 46

6 Conclusion and Future Work **50**
6.1 Future Work 52

List of Figures

3.1 ConcernMapper View 16

4.1 Decision Tree (example) 28
4.2 Weka Explorer 30
4.3 Decision Tree (Count) 33
4.4 Precision vs Recall (%) 38
4.5 minNumObj vs Predicted Concern-Size 40

5.1 Decision Tree Classifier (minNumObj = 20) 44
5.2 TaCoML Plug-in: Toolbar and Predicted Concern 45
5.3 Undo Child Node Creation 47

List of Tables

3.1 Target Project Characteristics 17
3.2 List of Attributes 21

4.1 minNumObj vs Precision (%) 35
4.2 minNumObj vs Recall (%) 37
4.3 minNumObj vs Predicted Concern-Size 39
4.4 Description of Nacin's Configurations (from [17]) 41
4.5 Precision-Recall using Nacin Algorithm 42

5.1 Results of Evaluation Study 48

Chapter 1
Introduction

1.1 Motivation

During the development and maintenance of software projects, developers are responsible for making changes to the source code. These changes are typically to add new features, enhance existing features, re-factor code, or fix bugs. Making changes in the software requires knowledge of the code layout, modularization and the interactions between the different program elements implementing a feature or a concern (e.g., "undo" feature in text editors, a logging facility in web servers, etc.). Henceforth, we use the terms features and concerns interchangeably. Often the source code changes required for a task are themselves spread across program element boundaries (methods, classes and even packages) affecting a cross-section of the code [6].

If developers are unfamiliar with the program source code, they have to learn its layout and structure before the changes can be made. Developers typically search through the source code, using their intuition and experience to locate the code elements relevant to the change task and understand their interactions [22]. This set of elements relevant to the change task is called the *task context* [12]. The task context includes the program elements which do not necessarily change but whose understanding is important to performing the task. Developers continue exploring the code until they are satisfied with their understanding of the context.

After completing the task, developers typically commit the changes to a source code repository, and move on to the next task. Only a subset of the task context, i.e., the changed code elements, are recorded in the repository and can be retrieved later. The knowledge gained about the related but unchanged elements in the task context is not tracked. Over time, the developers' understanding of the context decays as the specifics of the relevant code elements and their interactions are forgotten.

In the future, when developers are required to work on the same or related concern, they have to go again through the process of code exploration to recall the task context. Similarly, if a different developer is assigned to perform a related change task, they will have to spend time to discover the relevant code elements all over again. This repetitive exploration of code for discovering the same elements is a waste of valuable developer resources. A study by Ko et al. found that developers engaged in maintenance tasks spent on average 35% of time navigating dependencies in source code [9].

In order to alleviate the problem of repetitive exploration of source code to discover the task context, it would help if there was a way to automatically discover not only the changed elements but also the unchanged elements belonging to the task context. The task context, when persisted, can help the developers refresh their knowledge about the task when revisiting it later. The task context can also be used to familiarize new developers with the code associated with the same or related programming tasks.

Since the developer navigates through the elements belonging to the task context while performing the task, we believe that the data necessary to find the task context is encoded in the developer's interaction with the source code, as well as the latent structure of the source code. By observing the developer navigation traces and the source code itself, it should be possible to discover the subset of navigated elements that are the essence of the source code exploration, i.e., the task context. A developer navigation trace is the list of all the program elements touched by the developer in the course of performing a task. The navigation trace is henceforth referred to as the navigation transcript or transcript.

Developers vary in their approach to code exploration and performing change tasks. This introduces variance in the composition of task context. But a technique

using navigation transcripts for task context inference should be able to perform well under these conditions since developer variances will also be reflected in the transcripts.

It is usually easy for a developer to examine a program element and determine its membership in the task context. But there is no simple characterization of elements that belong to the task context. A developer will not be able to provide concise criteria to determine task context membership of program elements. In addition, any such criteria will vary between different developers and tasks.

Without a well defined characterization, it is difficult to develop algorithms for inferring the task context from the navigation transcript that perform well across different use case scenarios. Proposed solutions to infer task context from the transcript [4, 8, 17, 21] use ad-hoc algorithms developed based on the researcher's intuition and experience and also on gathered empirical data [15]. In addition, techniques using indirect artifacts other than navigation traces, such as source version history [27], developer communication via mailing lists [3], structure of the source code [7, 14, 18], etc., have also been proposed to help developers find the relevant program elements.

We believe that a developer's navigation transcript is the primary source of information about elements belonging to the task context since it contains the subset of elements in the program source code that the developer has encountered during the task execution. But the proposed solutions which use navigation transcripts face certain methodological limitations such as needing large data sets from multiple individuals [4], using ad-hoc, intuition-based algorithms that require manual fine-tuning for different usage scenarios [8, 17], providing information at coarse granularity [21], etc. We believe that an ideal solution should be able to use the information available in a developer's navigation transcript to generate a summary of the session using automated techniques that do not require manual tuning and provide information at a granularity that helps developers in tracking and communicating contexts of development tasks. In this thesis, we propose an alternative solution with the goal of satisfying these criteria.

1.2 Contribution

We propose to use machine learning algorithms to summarize the essence of a developer's source code exploration. Machine learning is often considered the technique of choice when solving problems in poorly understood domains, in which there is lack of knowledge required to develop effective algorithms [11]. Machine learning algorithms strive to learn the characteristics of the solution and can dynamically adapt to changing conditions. These algorithms analyze the data to find a good solution. Advantages of using learning algorithms include resistance to noise and variance in the input data.

In particular, we propose to use a decision tree learning algorithm [13] for inferring the task context by analyzing developer interactions with the source code, i.e., the navigation transcript. Decision tree learning is a supervised classification algorithm; it uses labeled training data, consisting of a set of attributes and a classification label as input. The outcome of the learning process is a decision tree classifier which tries to mimic the characteristics of the training data. The decision tree is then used to classify program elements in the transcript, as belonging or not to the task context.

In order to obtain training data for the learning algorithm, we conducted a user study in which we asked developers to identify the program elements in an object oriented system that are related to the implementation of a high level concern, for example, the 'undo' feature in a text editor. We recorded the developer interactions with the source code, i.e., the transcript, while they were performing the identification task.

We used the labelled data to train a decision tree classifier. For the learning process, we used the state-of-art C4.5 decision tree algorithm as implemented in Weka [23], a toolkit for machine learning and data mining. The learned decision tree classifier is evaluated using cross-validation. We also compared the precision and recall of the classifier to that of an algorithm based purely on the frequency of element visits, as well as to that of the Nacin algorithm proposed by Robillard et al. [17], which infers concerns from a program navigation trace. When compared to the frequency algorithm, the decision tree classifier has better or comparable performance in most

instances. In a few instances, it performed worse. The tree classifier performs better than all the configurations of the Nacin algorithm.

We developed an Eclipse plug-in to automate the summarization of the task context and integrate it into the developer work-flow. The plug-in uses the classifier learned from the training data to classify program elements visited by the developer in the course of a program exploration or change task. The developer can modify the list of recommended program elements and save it for future retrieval. We conducted a qualitative user study to obtain feedback on the plug-in recommendations.

The remaining part of this thesis describes in greater detail the contributions, methodology, and results of the use of decision tree learning for the analysis of program navigation data. In Chapter 2, we describe related prior work in navigation analysis, source code recommendation systems, and a few tools which aid developers in program navigation. The details of the user study from which we obtained training data are presented in Chapter 3. In Chapter 4, we describe the processing of the source code and transcript to derive attributes, and the evaluation of the learned classifier. The Eclipse plug-in, its usage, and the user feedback from a qualitative study of this tool are presented in Chapter 5. We conclude with the lessons learned and possible future directions to explore in Chapter 6.

Chapter 2
Related Work

This thesis is based on the premise that repetitive program exploration to discover the same set of elements is a waste of developers' time and that analyzing navigation traces can provide the necessary insight to eliminate this waste. Here we provide a brief overview of the research in areas related to our work - empirical studies in software engineering, analysis of program navigation traces, code browsing tools, and the use of machine learning techniques in software engineering.

2.1 Empirical Studies

Before building tools for software development, it is important to understand developers' behavior, their thought processes while building programs and the patterns in their work-flow. This understanding will provide insight into how tools can be integrated with the developers' work-flow to make them more effective and reduce resistance to their adoption. Understanding developer behaviour is important to developing all but the most basic and simple tools. In what follows, we highlight previous work in the area of empirical studies aimed at understanding of developer behavior.

In order to understand the factors that generate an effective program investigation behavior, Robillard et. al conducted an exploratory study of developers engaged in performing a change task [15]. They performed a qualitative analysis of the source

code exploration behavior of successful and unsuccessful developers. Developers were asked to perform a multi-part change task, with time alloted for source code investigation prior to the actual study. The resulting solution from each developer was evaluated for success or failure and the developer behavior was analyzed to understand what methods are used in effective program investigations. Based on the observation of differences between successful and unsuccessful developers, they conclude that a methodical investigation of program source code is more effective than random browsing in search of the parts related to the change task.

In another study [10], Ko et al. investigated the effect that differences between individual developers have on the source code investigation of unfamiliar systems. The subjects involved in the study were subjected to an initial battery of psychological tests, given a short introduction to a statistical programming environment and then required to perform a debugging task. The findings indicate that there were differences in the strategies of program comprehension among developers of differing skill levels. In contrast to the findings of the study conducted by Robillard et al. [15], none of the observed strategies were more effective than the others, and the subjects with the most domain knowledge were more successful in accomplishing the task. We believe the findings may be related to the beginner level of the developers involved in the task and their lack of programming expertise.

Ko et al., in a study of code improvement tasks, set out to discover the types of tools required by developers engaged in maintenance tasks [9]. Towards this end, they studied expert Java programmers while performing five maintenance tasks using the Eclipse IDE. Their findings suggest that the developers' activities during maintenance tasks can be grouped into: collecting code elements relevant to the task, navigating among these code elements and performing the changes required for the task. The findings of the study indicate that programmers spent 35% of their time navigating between the dependencies of the elements relevant to the task. Additionally, in their study they also found that developers spent on average 46% of the time navigating code elements not related to the task. If the results of the study are representative of practices at large, then it is necessary to provide developers with tools that support quick retrieval of task-relevant code.

2.2 Program Navigation Analysis

Researchers have proposed various techniques to analyze developer navigation traces.

Navtracks [21], an Eclipse plug-in, tracks the navigation history of a software developer. The navigation history is used to form associations between files visited by the developer, based on the heuristic that files that are part of a navigation cycle (i.e., a sequence of file navigations which start and end at the same file) are related. When detecting a cycle, the tool forms an association between the starting file and all the other files in the cycle. A parameter is used to control the size of the contiguous window in the navigation history that is examined to detect cycles. Similarly, a parameter controls the minimum length of the detected cycles. The values of the parameters are selected in an ad-hoc manner. The developer is presented with a list of files associated with the currently active file. Although useful in discovering associations between files other than those imposed by the hierarchy of the source code organization, the granularity of associations discovered by Navtracks, which is at the level of files, is too coarse. It leaves to the developer the task of searching within the suggested files for the related cross-cutting code elements.

Using a more fine grained approach, Robillard et al. present an automatic technique to infer the important elements (fields and methods) from a program investigation session [17]. Using the developer navigation transcript as input, the algorithm considers factors such as the order of the elements, their method of access (in editor, by scrolling, cross-reference, etc.), and the structural relationships between the examined elements to calculate a suggestion set consisting of methods and fields. The algorithm uses a set of nine configurable parameters. The parameters provide control over: 1) the importance of the ordering of the elements in the transcript, 2) the weights of the elements based on their method of access and, 3) the importance that two transcript elements are actually related in the program source code. The algorithm calculates a correlation metric which is then used to generate the suggestion set. The values of the parameters are chosen based on intuition and experimentation. The authors remark that parameter values need further optimization for different usage patterns to obtain better results.

Instead of using the developer navigation transcript to discover the task structure, Zou et al. use the transcript to detect interaction couplings between program elements [28]. Their approach is based on the hypothesis that if two elements are frequently examined together while performing a task, then the latent relationship between them is relevant to the task. The couplings are then mined to detect patterns that help in understanding program maintenance tasks. Zou et al. propose two patterns that they infer from the interaction couplings, 1) detecting changes to cloned copy of a source code file, and, 2) changes to interface which subsequently lead to changes in implementing classes. Based on their analysis of navigation transcripts and the resulting interaction couplings, Zou et al. suggest that restructuring code is more costly than any other maintenance task.

Kersten et al., in their tool Mylar [8] (now Mylyn), an Eclipse plug-in, use a different approach to model the task context. Each program element is assigned a weight or degree-of-interest, representing the current relative importance of the element. The degree-of-interest associated with a program element increases every time the user selects or edits the element. The degree-of-interest associated with an element decays over time if there is no user interaction with that element. The tool does not make use of any structural relations between the elements. The degree-of-interest is used to filter the elements visible to the user. The visible elements represent the context for the current task, because only the elements with active interaction maintain their degree-of-interest due to the gradual decay. The value of the parameters used to increase the degree-of-interest as well as those associated with the decay function are determined by experimentation.

Rather than relying on the navigation traces of a single developer, Team Tracks [4], leverages the navigation traces of a team of developers working on a common code base. By analyzing team navigation traces, Team Tracks infers two very different relations among program elements. Using the frequency of visits to particular elements, Team Tracks filters the hierarchical Class View to only show elements with visits above a certain threshold. The other elements are available too, but are shown as a collapsed list near the bottom. Secondly, the tool calculates the correlation between visits to elements. This correlation is used to suggest related elements to the

developer based on the currently active code element. The technique of using team navigation traces provides useful insights into the importance of code elements on a larger scale than an individual. But this may not necessarily be useful to a developer working on a task requiring access to infrequently accessed code.

2.3 Other Techniques

In their quest for better tools to help developers to find code related to their tasks, researchers have not restricted their work to the analysis of program navigation traces. Although it is beyond the scope of this document to survey all the different information sources and techniques of analyzing them proposed in the literature, we would like to highlight a few of these tools, because they share the end-goal of helping developers find the code they need. The tools presented below were selected to show the diversity of the techniques and information sources being investigated by researchers in the quest to ease a developer's burden. A more complete and thorough overview of the various tools and techniques is presented by Zeller in his comments on the future of programming environments [24].

Tools developed by directly searching the program source code - ranging from lexical analysis of text to exploiting structural relationships to find related code elements - form a baseline for comparing the rest of the tools highlighted below. A commonly used search tool, Grep [1], uses lexical search to find text. The search string can be specified as a regular expression, allowing for flexible searches. It is commonly used by developers to perform identifier-based searches in source code.

eROSE (previously ROSE), an Eclipse plug-in, developed by Zimmerman et al., mines version archives of software projects to recommend a set of program elements which are change-coupled to the currently active code element [27]. eROSE is based on the premise that fields and methods which have changed together in the past, as observed in the version archive, have a high likelihood of being related. Although able to discover elements related by changes, this technique fails to keep track of or discover unchanged important elements belonging to the task context.

In a more encompassing approach towards gathering information, Hipikat, mines multiple artifacts, to form an implicit group memory of the project [3]. The artifacts analyzed by Hipikat include the bugs database for the project, its version archive, messages from project mailing lists, and other project documentation. The Hipikat tool is provided as an Eclipse plug-in, which can query the central server for developer-driven queries.

A more advanced tool, FEAT [16] [18], facilitates the search and discovery of high level features or concerns in program source code. The discovered concerns are represented as concern graphs, in which the vertices are the program elements (classes, fields and methods) and the directed edges represent the different relationships (calls, reads, writes, superclass, etc) between these elements. A concern graph can be built by adding code elements to the concern and analyzing them for dependencies. The relationships between the elements in the concern can be documented. Using FEAT and the concern graph representation therein it is possible to document the high level knowledge about the source code as well as share this knowledge with other developers.

Another tool, JQuery [7], also implemented as an Eclipse plug-in, allows developers to browse code by combining the advantages of a hierarchical code browser with the flexibility of a query tool. As a code browser, JQuery provides an explicit representation of the exploration path taken by the developer. In addition, at every step of the exploration, it allows for searching on a range of relationships and queries to find an interesting subset of related elements. The rich set of relationship and query search options makes it easy to find the relevant elements. The explicit representation of the search path allows the developers to retrace their exploration and also facilitates back-tracing, in case they are following an uninteresting path.

2.4 Machine Learning in Software Engineering

In recent years, machine learning algorithms have gained increasing popularity in the field of software engineering. Zhang et al. provide an introduction and overview of

the uses of machine learning techniques in software engineering [25, 26] . They also survey the state of the art in the use of machine learning techniques to solve software engineering problems.

In their earlier work, Zhang and Tsai present a balanced view of the state of machine learning use in different areas of software engineering [25]. The authors classify software engineering entities into processes (a collection of activities), products (artifacts produced as a result of the processes), and resources (entities that are required by the processes). They classify software engineering problems into seven categories of tasks related to software engineering entities and provide references to the use of machine learning to solve problems in these categories. Here are some examples of problems in the seven categories:

- Attribute prediction - Software quality prediction, size estimation, cost prediction, defect prediction, etc.
- Attribute discovery - Discovering loop invariants, generating formal models of software system behavior by capturing data from live systems, etc.
- Product transformation - Transforming serial programs into parallel ones while preserving functional behavior, improving modularity of large programs, etc.
- Product synthesis - Test case and test data generation, learning software project management rules, generating project schedules, etc.
- Product and process reuse - Software library reuse suggestions based on source and target feature/term comparison, cost of rework, generalizing program abstractions to increase reuse potential, etc.
- Requirement acquisition - Methods to infer specifications from interaction scenarios, extracting specifications from software, etc.
- Development knowledge extraction - methods for capturing and preserving development knowledge, domain analysis methods, etc.

Similarly, in their later publication, Zhang et al. provide an overview of references to machine learning applications in software engineering data analysis and refinement, applications in software development, developing predictive models for software quality, they describes the state of the art as well as areas of future work [26].

Although, the machine learning applications presented in this section are in the area of software engineering, they are not related to the domain or the particular problem area addressed in this thesis. In light of this fact, we do not present further details of these applications.

Chapter 3
User Study

We performed a user study to observe developers' interactions with the source code while engaged in directed software exploration tasks. More specifically, we asked the developers to map high-level concepts or concerns (e.g., shuffle mode in a music player) to program elements (methods and fields) in the source code, and these mappings were recorded. For each instance of the mapping study, we also recorded the developer's navigation through the program source code. This recording of the navigation elements is called the transcript.

In this chapter, we describe the details of the study as well as the post-processing done on the navigation transcripts in order to obtain a data set to which machine learning algorithms can be applied. The study was performed in the scope of a larger empirical study of the concept assignment problem being pursued by Robillard et al. [19].

3.1 Methodology

As a part of the study, the subjects were required to identify program elements contributing to the implementation of a concern using the Eclipse IDE[1] for Java development.

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

At the start of the study, subjects were provided with a brief description of the system to be investigated, a description of the feature to be identified in source code and instructions on using the tools (ConcernMapper and the transcript recorder plug-in) for the study. The subjects were given as much time as they needed to familiarize themselves with the tools and their usage. Afterwards, they were required to complete the program investigation task in no more than ninety minutes.

While selecting the program elements implementing a concern, the subjects were instructed to use the following criterion as a guideline[19]:

> "it would be useful to know that the element is associated with the concern if I had to modify the implementation of the concern in the future, or if another developer had to modify the implementation of the concern".

The subjects were provided with an Eclipse workspace setup with a working copy of the target system to perform their investigation activities. To encourage the subjects to select the most representative elements implementing a concern, they were instructed to restrict the number of elements to not much more than twenty, although this restriction was not strictly enforced. The rationale for the restriction of concern-mapping size was to prevent the indiscriminate inclusion of all elements related to a concern in favor of only the important ones.

We instructed the subjects to record the elements belonging to the concern using the ConcernMapper plug-in [20]. ConcernMapper is an Eclipse plug-in that allows developers to associate class fields and methods to high level concerns by dragging and dropping them into the ConcernMapper View. The concern mapping is stored and can be retrieved later. Furthermore, the ConcernMapper plug-in provides a programming interface to access the code elements belonging to a concern mapping. We use this interface to obtain information about the elements belonging to a mapping and perform further processing to derive attributes of the transcript elements.

In addition to collecting the explicit mapping produced by the subjects, their interactions with the source code while performing the identification task were also recorded in a transcript. The transcript consists of a sequence of program elements explored by the developer during the program investigation. Additional attributes

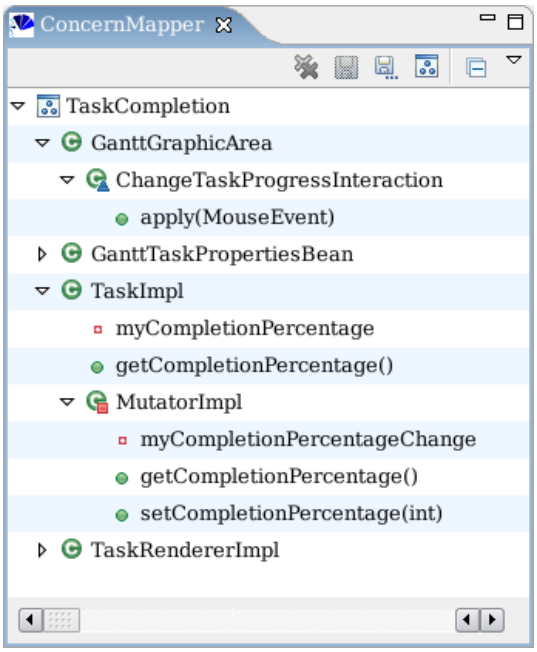

Figure 3.1: ConcernMapper View

such as timestamp, unique element identifier, etc. were also recorded with each program element. We provide more details about the structure of the transcript and additional attributes in Section 3.4.

The subjects chosen for the study, were experienced Java developers and experienced in using the Eclipse IDE. However, they were unfamiliar with the source code of the projects used for the study. The lack of familiarity with the source code was intentional, because we wanted to simulate the scenario of developers exploring unfamiliar source code in search of program elements implementing a particular feature.

Data was collected for 14 investigation sessions spread over 8 different program investigation tasks carried out by 7 different developers.

3.2 Target System

For the purpose of the study, the subjects investigated select features in three medium-sized open-source projects developed using the Java programming language. Each

Project	Version	LOC	Classes	Methods
Gantt Project	2.0.2	43,246	555	3,991
Jajuk	1.2	30,676	227	1,867
JBidWatcher	1.0	22,997	183	1,812

Table 3.1: Target Project Characteristics

system selected for the study consisted of over 20,000 lines of code (LOC) and over 150 type declarations. In addition, the systems have more than 150 reported bugs, more than 70,000 downloads and have been developed for more than 2 years. The systems were selected via the search and filtering interface of Sourceforge[2], an online portal for hosting open-source software projects. The system selection process was carried out as a part of another study [19] and is described in greater detail there. We used the following systems:

- **GanttProject.**[3] An Eclipse application for project-planning using gantt charts.
- **Jajuk.**[4] A music player and organizer supporting a variety of audio file formats such as MP3 and OGG.
- **JBidWatcher.**[5] A tool for tracking, sniping and bidding on auction sites (like eBay, or Yahoo).

Table 3.1 provides the main characteristics of the target systems gathered using the Metrics[6] plug-in for Eclipse and Sourceforge.

3.3 Target Concerns

The concerns used in this study were created by Robillard et. al. [19] manually by searching for high-level concepts in the bugs database, user manual and graphical

[2] `http://sourceforge.net`
[3] `http://gantproject.biz/`
[4] `http://jajuk.info/`
[5] `http://www.jbidwatcher.com/`
[6] `metrics.sourceforge.net`

user interfaces of the system. The authors specifically looked for concerns in the application domain that had a high possibility of being familiar to developers and which would be non-trivial to investigate. An excerpt of the concern descriptions used in this study is presented below [19]. Concerns C1-C4 are defined on the GanttProject, C6-C8 on Jajuk and concern C9 is defined on JBidWatcher. We follow the same numbering schema as used in the original study.

- **C1: Relationships.** The feature allowing users to add a relationship between two tasks.
- **C2: Non-working days.** The feature allowing users to specify the non-working days of the calendar (holidays and weekends) and taking these days into account when scheduling tasks.
- **C3: Completion.** The task completion feature allowing users to specify how much of a task is completed.
- **C4: Undo.** The mechanism allowing users to undo their actions.
- **C6: Shuffle Mode.** The feature allowing users to toggle between listening to tracks in sequential order or in random order.
- **C7: Add Song.** The feature allowing users to add a song to the playlist by dragging and dropping.
- **C8: Sort Collection.** The mechanism allowing users to sort their entire music collection according to different parameters (e.g., genre, artist, etc.).
- **C9: Updating Auctions.** The mechanism that constantly updates the information about auctions of interest (e.g., time left).

3.4 Transcript Description

The transcript is a sequential list of program elements touched by the developer during a program investigation session. The transcript is recorded as a series of interaction

events, with each event representing an interaction with a program element. We consider any selection of the program element with the cursor, keyboard or mouse, in the editor as well as in any of the Eclipse-provided views, as an interaction. The program elements were recorded at the level of the closest enclosing class member; e.g., when the cursor is positioned inside a method body, the enclosing method is recorded. To record the session transcripts, we developed an Eclipse plug-in for the generation and capture of interaction events.

At the beginning of an investigation session, the subjects were required to activate the transcript recorder plug-in by either pressing an icon in the toolbar or selecting the option from the menu. The end of an investigation session was similarly indicated. All the elements touched by the subject during the session are recorded by the plug-in. The plug-in provides an option to store the recorded transcript in the form of a comma-separated-value (CSV) file.

For every interaction event, the plug-in records the following attributes about the event:

- **Handle** - a unique string representation of the Java program element in the Eclipse workspace.
- **Type** - the type of the Java element associated with the event (for example package, type, method, etc.).
- **Timestamp** - the time at which the event occurred.
- **PartId** - a string identifier for the part in which the event was selected (for example Outline view, editor etc).

Upon starting a development session, the plug-in registers with the Eclipse selection mechanism and listens to all the generated selection events. For every selection in the Eclipse IDE workbench, the selection mechanism generates a call-back with the selected object as argument. The plug-in filters all events except those originating from Java program element interactions and records them along with the above-entioned attributes. At the end of the session, the plug-in stops listening to

the selection events and presents the developer with the option to save the transcript to a file.

3.5 Data Transformation

The transcript does not contain any information about the structure and semantics of the program elements within the program (both of which are available to the developer while navigating through the source code). The developer uses the program structure and semantics to relate a particular program element to the implementation of a high level concern. Similarly, within a set of concern-implementing elements, the program structure guides the selection of the important elements.

We wanted to use a developer's navigation transcript in conjunction with the information available in the source code to generate a summary of the important elements in the transcript. In order to capture some of the information to which the developer has access during program exploration, we transform a transcript from a sequence of interaction events to a list of distinct program elements with derived attributes; these attributes encode information related to the program structure and semantics. They are computed by analyzing the transcript and the program source code.

The derived attributes summarize the transcript (repetition of program elements, time spent with a particular element, etc.) and encode structural information (e.g., fan-in for methods and fields) extracted from the source code of the target system. Table 3.2 lists the attributes and their description. An additional binary attribute called the 'class', corresponding to the inclusion of the element in the developer-created concern, is also appended. Thus, if an element belongs to the concern this attribute is set to 'true', and it is 'false' otherwise. The 'class' attribute assignment is based on the hypothesis that the concern mapping produced during the session is the summary of the development session as evaluated by the subject engaged in the discovery of the concern-mapping.

Attribute	Type	Description
handle	nominal	Unique identifier for a Java program element.
type	nominal	Type of the program element (field or method).
totalDuration	numeric	Total time spent (in milliseconds) examining a program element.
noOfSiblings	numeric	The number of siblings of the program element that appear in the transcript.
noOfCalledMethods	numeric	Number of methods in the transcript called by the current program element. This attribute is '0' for field elements.
noOfTCalledMethods	numeric	Number of methods in the transcript calling the current method. This attribute is '0' for field elements.
noOfFieldAccesses	numeric	Number of times a field is accessed by methods within the transcript. This attribute is '0' for methods.
noOfKeywordsInHandle	numeric	The number of the keywords that appear in the handle of the program element. The list of keywords to look for is specified by the user. It is expected that the user enters keywords related to the task.
noOfKeywordsInName	numeric	The number of keywords that appear in the name identifier of the program element. The keyword list is the same as for the previous attribute.
class	nominal	A binary attribute indicating whether the program element has been labelled as part of the concern by the user.

Table 3.2: List of Attributes

The transformed transcript is saved as an ARFF[7](Attribute-Relation File Format) file. ARFF is an ASCII text file format that is the default input format for Weka[8] - a library of machine learning algorithms.

An ARFF file has two sections - header and data. The header contains information about the name of the dataset, the list of attributes and their type. The data section contains the actual data, with each instance on one line. The attributes of an instance are separated by a comma. We use two types of ARFF attributes: numeric - which can be real or integer numbers, and nominal - which can take one of a specified list of values (e.g., the attribute 'class' can be one of 'true' or 'false'). Further details about Weka and the machine learning analysis are presented in Chapter 4.

[7] http://weka.sourceforge.net/wekadoc/index.php/en:ARFF_(3.5.1)
[8] http://www.cs.waikato.ac.nz/ml/weka/

Chapter 4
Experiment Methodology and Results

We wanted to investigate the effectiveness of machine learning techniques for the classification of program elements to summarize a programmer's development session. In particular, we used decision trees, a supervised learning algorithm, for the classification task. We used precision and recall to evaluate the performance of the algorithm. Furthermore, we compared the performance of the decision tree algorithm with two other algorithms. This chapter introduces the machine learning algorithm used, describes the tools used for the analysis, and explains the results obtained.

4.1 Machine Learning

Machine learning, a sub-field of artificial intelligence, is concerned with the design and development of algorithms and techniques that allow a software program to extract knowledge from input data and learn from experience [11]. Machine learning techniques are well suited to problem domains which are difficult to model. Most relevant to our problem domain are supervised learning algorithms, which work by summarizing a labelled data set into a model. This model is supposed to capture the most important characteristics of the data. The learned model can then be used to predict labels for new problem instances. In many cases the model also provides insight into the nature of the process generating the problem instances. This insight can be further used to develop more specialized algorithms.

Typically, the development of a supervised machine learning solution for a problem contains the following steps:

1. **Attributes Selection** - In supervised learning, an instance consists of a set of attribute-value pairs. Examples of attributes could be the in-degree of a method in a call-graph, the number of accessors of a field, etc. They describe facts about program elements. The choice of attributes used to describe a problem instance reflects the domain expert's belief that the attributes are correlated to the desired label that will be predicted by the learned model. Conversely, the learned model can also provide insight into the relative importance of various attributes used to describe a problem instance. Attribute selection has a very important influence on the performance of the learned model.

   In the case of supervised learning, an instance, in addition to attribute-value pairs, also consists of an assigned label. The label assignment for the training data is typically done by experts or obtained from real world data and represents reliable knowledge available about the problem instances.

   For example, for the classification of fruits, attributes such as size (big, medium, small), shape (round, elongated), color (yellow, red, green), texture (smooth, rough), weight (heavy, light), etc. can be used. The expert chooses the attributes that provide higher benefits than the cost of obtaining the attribute values, e.g., (color, shape, and texture). An instance of training data for the fruit classification problem could then be made of tuples such as (red, round, smooth; apple), (yellow, elongated, smooth; banana) wherein the last term is the label assignment for the particular instance of attribute values.

2. **Model and Algorithm Selection** - Researchers have proposed various models to represent the solution to a machine learning problem, e.g., classification rules, neural networks [11], support vector machines [5], decision trees [13], etc. The models vary in the complexity of their representation, complexity of the learning algorithm and the computation required to compute the output label assignment from the attribute values. Depending on the representation power

and the learning algorithms, different models and algorithms are suited to different problem domains. The choice of an appropriate model (in conjunction with an appropriate set of attributes) determines to a large extent the performance of the machine learning application.

A model in machine learning consists of parameters and an evaluation algorithm which computes the output labels based on the attribute values. The model parameters control the behavior of the evaluation algorithm. An evaluation algorithm specifies the steps needed to transform the input attributes of the problem instance to a value or a set of values representing the output.

A supervised learning algorithm analyzes the input training data to learn patterns useful in the mapping between the attribute values and the associated label. The algorithm tries to capture the statistical relations between the values of the attributes and the corresponding label assignment. The outcome of the learning process is a value for the parameters that, together with the evaluation algorithm, form a solution to the machine learning problem.

For our fruit classification example, we want to select a model which is easy to interpret, and provides a simple representation. We will use *classification rules* to represnt the learned model. Classification rules are composed of tests of attribute values combined together with logical operators. There are different learning algorithms used to learn classification rules such as conjunctive rule learner (which learns a conjunction of clauses), propositional rule learner [2], etc. To keep things simple we will demonstrate a classification rule based model by using the conjunctive rule learner as an example. A conjunctive rule to identify apples could look like: (color = "red") $\wedge$ (shape = "round") $\wedge$ (texture = "smooth") $\Rightarrow$ "apple".

3. **Model Testing and Tuning** - The learned model needs to be evaluated in order to test its effectiveness for the machine learning task. Typically, a large portion of the available labeled data is used for training, with a small fraction left over for testing. Labels for the test data are computed using the learned model

and the predicted output is then compared with the original labels associated with the training instances.

Two commonly used criteria for measuring the effectiveness of machine learning algorithms are precision and recall. Precision is defined as the ratio of the number of correct predictions to the total number of predictions in the output. It is a measure of the accuracy of an algorithm in assigning the correct labels. Recall, for a particular label, is defined as the ratio of the correct label assignments to the total number of test instances belonging to that class in the input. Intuitively, recall measures the ability of the algorithm to find the instances of the target class in a large pool of data.

More precisely, let X be the set of test instances having label $\mathcal{A}$ and let Y be the set of instances predicted to have the label $\mathcal{A}$. Then precision and recall for class $\mathcal{A}$ are calculated as follows:

$$precision = \frac{|X \cap Y|}{|Y|} \tag{4.1}$$

$$recall = \frac{|X \cap Y|}{|X|} \tag{4.2}$$

In an ideal situation, an algorithm has high precision and recall. But, in practice, precision and recall tend to be inversely related. In developing machine learning applications, precision and recall are typically measured across a range of settings for the learning algorithm, for a given set of training and test data. The settings which offers the best trade-off between precision and recall for the given application is used.

Using our choice of attributes (color, shape, texture), the representation model (classification rules) and the learning algorithm (conjunctive rule) for the classification of fruits, we train a model and then evaluate the learned model on the test data. The evaluation will provide quantitative data to compare the performance between different attribute, model and algorithm selection. We can choose to tweak the attributes that we use in the training data by adding

"weight" or dropping "textures". This will allow us to evaluate the relative benefits of the different attributes. If changing the attribute selection does not provide the necessary accuracy, we could choose to adopt a different learning algorithm or even change the model (e.g., decision trees) use to represent the outcome of the learning process.

The development process for a machine learning application iterates over the above stages to fine tune the solution until the required performance characteristics are obtained.

4.2 Machine Learning for Navigation Analysis

In this section we describe the use of a machine learning algorithm for the analysis of developer navigation traces. The description reflects the stages described in the previous section and outlines the choices we made with respect to the input, the model, and the algorithm.

In Chapter 3 we presented a user study from which we obtained mappings from program elements to high-level concerns. We also collected developer navigation traces during the task of discovering these mappings. The data collected in the study consisted of the following artifacts: the concern mappings, the navigation traces, and the program source code. In its original form, this data was unsuitable for use with machine learning algorithms. We processed the artifacts collected in the user study to transform the raw data into a fixed number of attribute-value pairs. In the course of this transformation, we computed attributes from navigation traces, with additional information extracted from the program source code. The label assignment, using the concern mappings as reference, simply indicated whether the element belonged to the concern or not according to the developer.

```
count <= 1.0: false (196.0/27.0)
count > 1.0
|   noOfKeywordsInName <= 0.0
|   |   noOfSiblings <= 2.0: false (84.0/22.0)
|   |   noOfSiblings > 2.0
|   |   |   count <= 3.0
|   |   |   |   noOfSiblings <= 6.0: true (56.0/26.0)
|   |   |   |   noOfSiblings > 6.0: false (70.0/24.0)
|   |   |   count > 3.0: true (112.0/29.0)
|   noOfKeywordsInName > 0.0: true (91.0/18.0)
```

Figure 4.1: Decision Tree (example)

4.2.1 Decision Trees

Due to its advantages, we use decision trees as the machine learning model for the classification of program elements [13]. In a decision tree, each internal node represents a comparison test on the value of an attribute. The branches are directed and are labeled with the outcome of the test. The leaf nodes of the tree are marked with the classification label to be assigned to problem instances associated with the leaf. In a decision tree, the classification label assigned to an instance is based on the outcome of the attribute tests at the internal nodes leading from the root of the tree to a particular leaf node.

For classifying problem instances, their attribute values are tested starting at the root of the tree. Based on the outcome of a test, a particular branch is followed, which may in turn lead to further tests. This process of testing and following a branch is repeated until a leaf node is reached. On reaching a leaf node, the problem instance is assigned its label. The comparison tests at internal nodes are exhaustive on the range of the attributes being tested and the branches leaving the node represent mutually exclusive outcomes of the test. Due to mutual exclusion, there is no ambiguity in the outcome of tests. Although decision trees can have different branching factors for internal nodes, usually binary decision trees are used. A general decision tree can always be converted to a binary decision tree.

Figure 4.1 shows an example of a decision tree learned from a subset of the training data gathered during the user study described in Chapter 3. The attributes and their meaning are described in Table 3.2.

In the representation used in Figure 4.1, the root of the tree is at the outermost level of indentation, with nodes at subsequent levels having progressively more indentation. Each node is represented by two conditions on an attribute, representing the branch to be followed based on the outcome of the test on the attribute value. Note that the attribute tests at a node are mutually exclusive and exhaustive on the range of the attribute. The label assigned to a leaf node is represented by the text after ':'. In this tree, there are two labels - 'true' and 'false'. A true label assignment represents the outcome that a particular program element belongs to the concern. For example, a program element which has been visited 4 times in a development session (count = 4), which does not have any keywords in its string literal (noOfKeywordsInName = 0), and no other sibling elements being visited (noOfSiblings = 0), will be assigned the label 'false'. The numbers at the leaf nodes represent the ratio of the number of training instances which reached a particular leaf and agree with the leaf label, as opposed to those that do not. E.g., the numbers on the right branch of the root node indicate that 196 of the training instances having 'count' less than or equal to '1' had the label false and 27 had the label 'true'.

We highlight some of the advantages of using decision trees for classification [11]:

- Each path from the root of a decision tree to a leaf represents a conjunction of constraints on attribute values. The decision tree as a whole is a disjunction of conjunctive clauses represented by the paths. As such, decision trees provide a concise and general representation for the rules of classification.

- Decision trees are easy to interpret. Simply examining a decision tree can provide insight into the characteristics of the problem. For example, a quick observation of the decision tree in Figure 4.1 informs us that for the particular training dataset used to train this decision tree, an element that has been visited only once has a high likelihood of being unimportant to the development session summary.

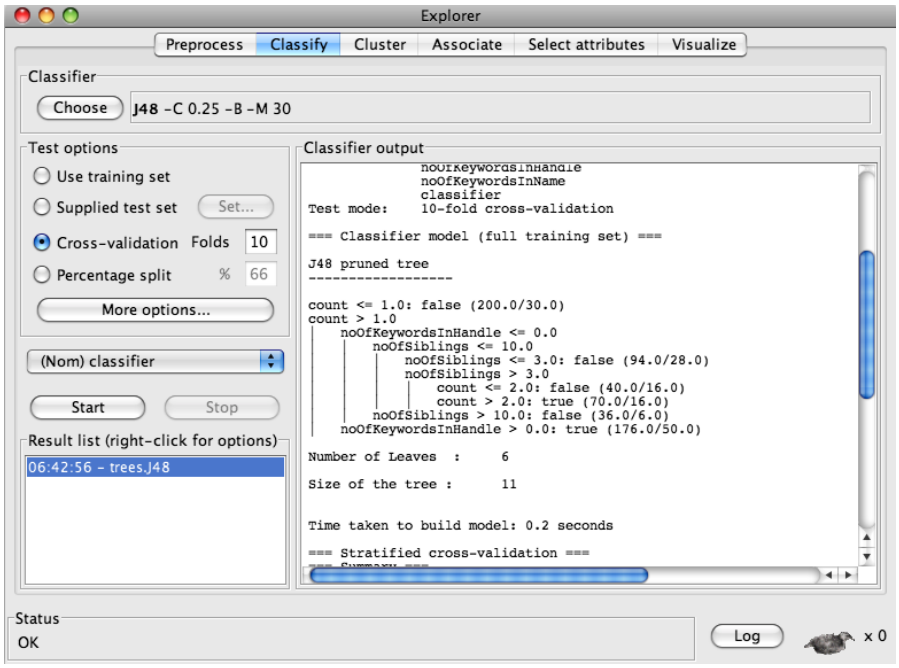

Figure 4.2: Weka Explorer

- Decision tree learning algorithms are fast and can quickly process large datasets.
- Decision tree learning algorithms are to a large extent immune to noise. The tree learning algorithm performs well in the presence of noise in attribute and classifier values in the training data as well as when there are missing values in the attributes and classifier.
- The number of training instances reaching a particular leaf provides a confidence measure to the label assignment for a problem instance. This provides an indicator to the quality of label prediction.
- Compared to most other machine learning techniques, decision trees provide the flexibility of working well with numerical as well as categorical data.

Decision trees have been successfully used in wide ranging applications from classification of stars to medical diagnosis and industrial applications.

To construct the decision trees, we used tools provided by the Weka[1] [23] project. Figure 4.2 shows a screen capture of the Weka application. Weka is an open-source machine learning toolkit that provides implementations of various learning algorithms. Weka tools make it easy to perform various activities associated with the development of machine learning applications. They facilitate data preprocessing and the application of different learning algorithms to the data, support various test strategies and offer the ability to save the learned model. Additionally, Weka also provides an interface to access the learning algorithms from programs written in Java.

For decision tree learning, we used J48 - an implementation of the state-of-art C4.5 [13] decision tree learning algorithm in Weka. The algorithm accepts a set of already classified instances of training data as input and builds a decision tree.

The algorithm uses the concept of information gain to select an attribute to test at a node. It starts with the complete set of training instances and selects an attribute and an associated test that maximizes the information gain. The test splits the input data into two disjoint sets. In each subsequent iteration, the algorithm is called recursively on the resulting subsets from the previous iteration to build a decision tree. The algorithm terminates when all the instances in a subset have the same label, in which case the leaf is assigned that label. Alternate criteria, such as the ratio of the number of instances of a label to the total instances at a leaf node, may also be used to prevent further splitting of the tree nodes. Such criteria can be used to restrict the growth of the decision tree and also, indirectly, the height of the tree.

4.3 Experimental Methodology

Using the pre-processed data derived from the concern mappings and the developer navigation traces described in Chapter 3 and the Weka provided J48 algorithm, we performed a series of experiments in which we built decision tree classifiers and evaluated their performance. We also compared the performance of the decision tree algorithm to that of Nacin [17], and an approach using frequency of program element

[1] `http://www.cs.waikato.ac.nz/ml/weka/`

occurrence in the transcript, henceforth referred to as the count metric, - techniques suggested in the literature for summarizing a developer navigation session.

Corresponding to each of the 14 experiments performed as part of the user study, we created a dataset consisting of training and test instances. The training data for each set was built by concatenating the transformed transcripts of thirteen experiments with the fourteenth experiment being the test set. The training instances were used to train a decision tree using the J48 algorithm.

The J48 implementation of the C4.5 decision tree learning algorithm provides a configurable parameter that indirectly controls the size of the learned decision tree. The numerical parameter, called "minNumObj", is used by the algorithm before it proceeds to split a set of training instances into smaller subsets. The algorithm does not split any set whose cardinality is less than the value of "minNumObj". Thus higher values of "minNumObj" will lead to reduced splitting of the training data set and subsequently fewer levels in the learned decision tree. By controlling the value of the parameter, we can search for the best trade-off between the size of the learned tree and the precision and recall of the tree on test data.

For each run of the learning algorithm, we measured the precision and recall of the learned decision tree on the test data. The test procedure provides a label assignment for each of the program elements belonging to the test data. The assigned label represents a prediction about the inclusion of the program element in the summary of the development session. For all the elements predicted to be part of the session summary, we considered a prediction to be correct if the program element was also part of the concern-mapping produced by the developer for that particular user-study experiment.

Based on the definition of a correct prediction, precision and recall were calculated for each run of the learning algorithm. We used the following formulae to calculate precision and recall:

$$precision = \frac{|P_C \cap A_C|}{|P_C|} \tag{4.3}$$

$$recall = \frac{|P_C \cap A_C|}{|A_C|} \tag{4.4}$$

where P_C is the set of elements predicted to be in the concern and, A_C is the set of elements belonging to the concern.

For the task of program element classification, *precision* is the ratio of program elements belonging to the concern-mapping to the total number of program elements predicted to belong to the concern-mapping by the decision tree. Similarly, *recall* was defined as the ratio of the program elements with correct predictions to the total number of elements belonging to the original concern-mapping.

4.4 Results

Tables 4.1 and 4.2 present the precision and recall calculated using the above formulas for different values of the parameter "minNumObj". In both of the tables, the first column is the value of "minNumObj" passed to the J48 algorithm. As explained above, the value is inversely related to the size of the learned decision tree. This inverse relationship can also be observed from the last column which shows the average number of leaves of the decision tree over the fourteen experiments for each of the values of "minNumObj". The last two rows in Table 4.1 and 4.2 present a comparison of the precision and recall of the decision tree algorithm with those obtained using two variants of the 'Count' metric.

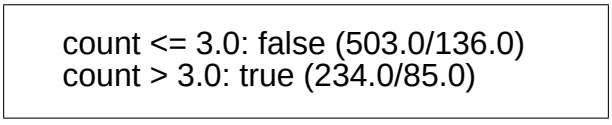

Figure 4.3: Decision Tree (Count)

In the first variant, 'CD', we continued to use the J48 decision tree algorithm but modified the input training data. The new training data consisted of only one attribute, 'count'. The decision tree algorithm, during its training, proceeded to find the value of the attribute which achieves the best split of the input data. The output decision tree contains only one decision node that partitioned the input elements on

the basis of the associated value of 'count' which maximized prediction accuracy. The precision and recall obtained using this variant highlighted the gain in accuracy by using a combination of attributes encoding the program structure as opposed to just the 'count' attribute. Figure 4.3 shows a decision tree learned from training data consisting of only the 'count' attribute.

To obtain the precision and recall by using the 'CN' variant, the predicted elements were selected based on their frequency of occurrence in the transcript. The frequency is the number of times the developer explored a particular code element during the course of their investigation. This variant is based on the hypothesis that the more often the developer encountered a particular element during their investigation, the higher the likelihood of the element belonging to the concern. Using this method we selected a set of elements having the same cardinality as the reference concern-mapping created by the subject for that user-study experiment. This set of predicted elements was then compared with the reference concern-mapping to calculate the precision and recall. It should be noted that it is not possible to know the exact size of the concern set a priori, but for analysis purposes it is still relevant to compare the performance of this technique to that obtained using the decision tree algorithm.

As seen from Table 4.1, the precision varies for different values of the parameter "minNumObj". The best results of 69% average precision are obtained with "minNumObj" set to 5 which has lower average tree size (less than half the leaf nodes) compared to when "minNumObj" is set to one. We attribute this behavior to *overfitting*, a commonly observed phenomenon in machine learning. Overfitting occurs when the representational power of the machine learning model is larger than the information contained in the training data, e.g., a decision tree that has a leaf node corresponding to every instance of the training data. Instead of learning the patterns in the training data, the machine learning algorithm memorizes the training data. Due to overfitting, machine learning models can perform very well on training data but poorly on new inputs. By comparing different model sizes we are able to discover the model size (minNumObj = 5) which performs well not only on training data but also on test data. In further discussion related to precision we restrict ourselves to this particular parameter value, unless otherwise specified.

	Experiment No.															
minNumObj	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Avg	AvgL
1	100	45	55	45	52	36	67	47	69	17	80	74	92	66	60	58
5	100	58	75	56	60	40	55	61	92	22	80	87	91	91	69	21
10	100	55	67	60	62	32	50	61	73	29	80	92	87	68	65	13
20	100	42	80	44	56	47	55	54	75	29	78	86	82	68	64	9
30	100	45	60	44	56	33	55	56	75	29	67	100	83	67	62	7
40	83	45	73	44	44	37	55	63	86	33	67	100	83	67	63	6
50	83	45	73	56	52	37	58	61	86	33	67	100	83	68	65	6
CD	89	42	78	47	46	24	44	50	67	33	73	100	92	94	63	
CN	90	45	73	50	40	44	55	50	62	29	79	73	81	76	61	

Table 4.1: minNumObj vs Precision (%)
Avg = Average; AvgL = Average number of leaves; CD = Count (Decision Tree with 3 nodes); CN = Count (Number of elements = concern size)

Compared to a count variant using decision trees, 'CD', the learned classifier with minNumObj parameter value of five, has better precision in nine instances and is lower by 3% or less in three of the remaining five instances. Only in two instances does the decision tree classifier perform significantly worse than the 'CD' variant. These results indicate that the presence of additional attributes along with a decision tree classifier does indeed lead to comparable or better precision for our task. Additionally, the decision tree classifier has equivalent or better precision than the 'CN' variant in 12 of the 14 instances.

In Table 4.1 there are a few instances that need further investigation. The precision results for Experiment 10 are the lowest among all the different instances. On further investigation, we discovered that this experiment was performed by an expert developer with high level of proficiency in using the development environment. Another example of poor performance by the classifier can be observed for experiment 6. The subject's comments indicated that the program source code (JBidWatch) was

badly written and hence a majority of the time alloted to the task was spent in exploring random navigation paths in the hope of discovering the concern implementing elements. It was only towards the end of the task that the concern implementing elements were encountered. The poor code quality was corroborated by another subject who performed the same task in Experiment 5. But this subject got a lead early on and hence did not spend a large portion of the time alloted to the task in random explorations.

Based on these observations we hypothesize that the use of decision tree classifier for development task summarization performs poorly in situations in which the developer does not have a clue regarding the elements related to the concern, such as when randomly exploring program code. We believe that the decision tree extracts information from the navigation transcripts by observing patterns which are not immediately obvious. In the above case, due to random code exploration, the attributes for Experiment 6 do not fit the patterns exhibited in the training data leading to poor performance. Similarly, the classifier is unable to recognize the patterns in the transcripts for Experiment 10, but for a very different reason. In this case, the session transcript exhibits patterns of an expert developer comfortable with the tools, while the training data consists of transcripts of less advanced developers. Both of the above cases confirm the findings of Robillard et al. that code exploration patterns vary between developers of different skill level [15]. The differences in exploration patters requires us to develop more specific classifier models based on groupings of developers' expertise to obtain better results. Unfortunately, due to lack of sufficient training data for developers of different skill level we were unable to follow this investigation further. To some extent, similar behavior is observed also in the results of the evaluation study that we present in Chapter 5.

In the recall results presented in Table 4.2, among decision tree classifiers the classifier with minNumObj values of 20 and 30 have amont the highest average recall of 56% and 55.6% respectively. Note that the parameter value which achieves the highest average precision (minNumObj = 5) does not have the best recall. In comparison, the 'CN' technique has the highest average recall of 60.5%. Though it must be noted that this technique is not practical as it assumes that the size of the concern

	Experiment No.														
minNumObj	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Avg
1	70	45	55	28	60	44	55	83	44	24	33	47	52	63	50.2
5	40	64	55	50	60	67	55	94	48	29	33	43	48	61	53.4
10	20	55	55	33	65	78	45	94	44	29	33	37	62	68	51.3
20	60	45	73	39	70	78	55	83	48	29	29	40	67	68	56.0
30	60	45	82	39	70	78	55	83	48	29	25	27	71	66	55.6
40	50	45	73	39	55	78	55	83	48	29	25	33	71	65	53.5
50	50	45	73	50	55	78	64	78	48	29	25	30	71	69	54.6
CD	80	73	64	44	55	44	36	56	32	35	33	10	52	44	47.0
CN	90	45	73	50	40	44	55	50	62	29	79	73	81	76	60.5

Table 4.2: minNumObj vs Recall (%)
CD = Count (Decision Tree with 3 nodes)
CN = Count (Number of elements = concern size)

is known in advance and using a default constant size will affect the recall across different experiments. Commonly, when using statistical classification techniques, a trade-off between precision and recall is observed. Efforts to achieve higher precision by being more selective in the classification process usually lead to lower recall due to the increased possibility of boundary cases being labeled negative. Similarly, the increase in recall by lowering selectivity and allowing more boundary instances to be labeled positive leads to lower precision.

Figure 4.4 compares the average precision and recall obtained when using different techniques (decision trees, Nacin, and, 'CD' and 'CN' variants of count) to classify code elements in a navigation transcript. Both decision trees and Nacin have multiple data points as we tested them for multiple configurations. The points are labelled with the respective identifiers ("minNumObj" for decision trees and configuration labels as in Table 4.5 for Nacin).

In Table 4.3 we present the cardinality of the predicted concern sets for the different experiments for different values of "minNumObj", as well as that of the original

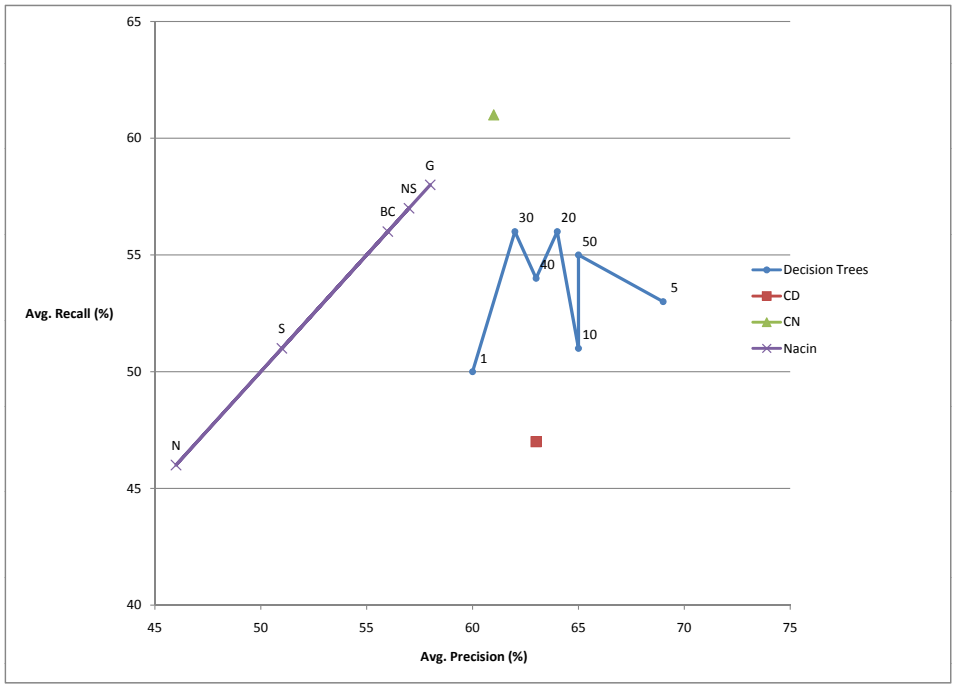

Figure 4.4: Precision vs Recall (%)

reference concern. The size comparison between the predicted and the reference concerns shows the effectiveness of the algorithm in predicting a concern set with a cardinality which matches with the developers' expectation. This comparison is particularly important as the training set used for the decision tree learning did not contain any information regarding the size of the reference concerns. The last row contains the number of distinct elements (methods and fields) that are in the transcript. Comparing the size of the predicted set with the number of distinct transcript elements demonstrates the selectivity of the classifiers.

Another view of the data in Table 4.3 is presented in Figure 4.5. The figure presents a graph of the average concern size when varying the value of the control parameter "minNumObj". For comparison, we also present the average number of elements in the reference concerns as well as the average number of elements in the concerns produced using the 'CD' variant of the count metric. Since the latter two are constant with respect to "minNumObj" they appear as straight lines in the graph. As

	Experiment No.													
minNumObj	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	7	11	11	11	23	11	9	32	16	23	10	19	12	68
5	4	12	8	16	20	15	11	28	13	23	10	15	11	47
10	2	11	9	10	21	22	10	28	15	17	10	12	15	71
20	6	12	10	16	25	15	11	28	16	17	9	14	17	71
30	6	11	15	16	25	21	11	27	16	17	9	8	18	70
40	6	11	11	16	25	19	11	24	14	15	9	10	18	69
50	6	11	11	16	21	19	12	23	14	15	9	9	18	72
CD	9	19	9	17	24	17	9	20	12	18	11	3	12	33
Reference	10	11	11	18	20	9	11	18	25	17	24	30	21	71
NDE	15	45	36	67	69	43	22	79	58	85	45	51	37	107

Table 4.3: minNumObj vs Predicted Concern-Size
CD = Count (Decision Tree with 3 nodes)
NDE = Number of distinct elements (methods and fields only) in the transcript

can be seen from the graph, the size of the concerns using decision trees lies between that of the reference concerns and those produced using the count metric.

From the results presented in Tables 4.1, 4.2, and 4.3, we observe that the classifiers can identify a little more than half of the code elements in the original concerns. Also, in the case of precision, the decision tree classifier predicts on average 7 out of 10 elements correctly. The choice of a particular parameter value to use in an application depends on the preference of the relative trade-off between precision and recall. Additionally, a tool that uses decision trees must allow developers the flexibility to add and delete elements from the predicted concern set after it has been computed. Developers can then further modify the predicted concern set as per their volition. Thus, developers can focus on their development task and at the end of the session make use of the predicted summary, modifying it where necessary, to save high level concern knowledge.

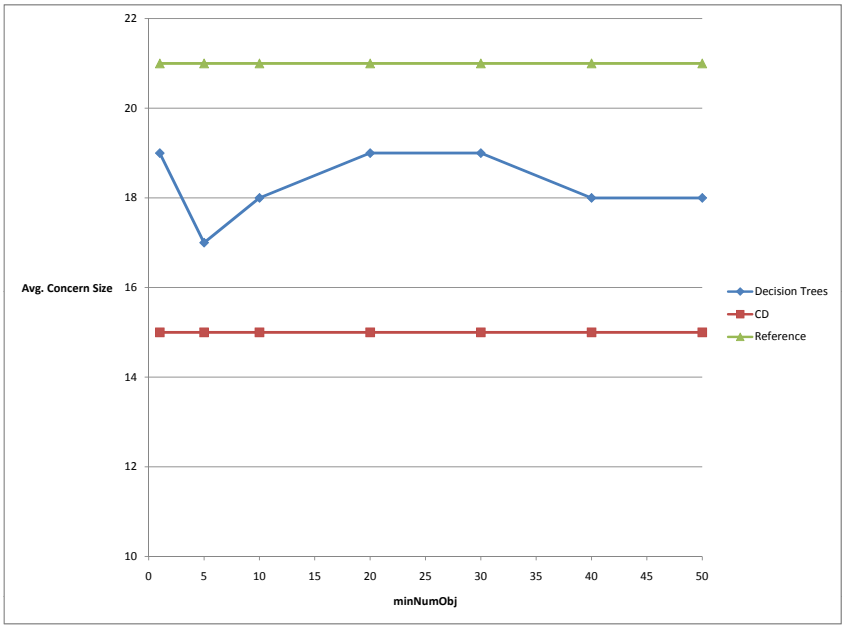

Figure 4.5: minNumObj vs Predicted Concern-Size

4.4.1 Comparison with Nacin

In addition to evaluating the precision and recall of variants of the 'Count' algorithm, we also evaluated the Nacin [17] algorithm using the navigation transcripts from the study and compared its performance to that of the decision trees. The developer navigation transcripts were transformed into a suitable input format. Nacin, using a parameterized heuristic algorithm, ranks the elements in the transcripts in order of importance. For the task of concern prediction, we restricted the output to a set of the highest ranked elements having the same cardinality as the reference concerns from the user study. This particular arrangement led to the precision and recall being equal as the size of the predicted concern and the size of the reference concern was the same. The algorithm uses a set of 9 parameters which influence the heuristic to form different configurations of operation. We evaluated the performance of five different configurations. A brief description of the configurations appears in Table 4.4. Robillard et al. provide additional details describing the configurations [17]. The precision-recall of Nacin for these five configurations is presented in Table 4.5.

Configuration	Description
Basic	Based on intuition of clues to important elements in program navigation
Neighbors	Taking into account only immediately succeeding elements in program navigation
No Structure	Using developer action as sole basis, ignoring underlying structure in source code
Structure	Emphasizing navigation transitions based on program structure
Guesses	Configuration which places more importance on guesses and browsing

Table 4.4: Description of Nacin's Configurations (from [17])

The last two rows of the table are precision and recall for decision tree classifier with parameter value of five and are copied over from Table 4.1 and 4.2 for comparison.

Of the five Nacin configurations, the best average performance (precision as well as recall) of 58% is achieved for the configuration which places higher importance on guessing and browsing. In comparison, the average precision of the decision tree classifier is significantly higher at 69% with a recall of 53% which is lower than the 'Guesses' configuration by 5%. Based on these results we can say that the decision tree classifier is able to provide more accurate results compared to the Nacin algorithm while having slightly lower recall.

The results in Table 4.5 demonstrate that a model based on machine learning technique performs better than a more ad hoc algorithm based on intuition. We further believe that it is possible to achieve even higher gains in performance by using more complex machine learning algorithms, which model more closely developers' thought processes in forming the task context. The intuition gained by observing developers' behavior will help in choosing the model to use as well as the values of the model parameters.

	Experiment No.														
Configuration	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Avg
BC	90	18	64	28	40	20	46	61	65	35	71	80	86	85	56
N	60	27	55	33	30	11	55	61	38	35	54	70	57	56	46
NS	90	27	64	33	35	22	55	61	65	35	71	80	81	85	57
S	60	18	46	11	40	22	46	61	65	35	71	70	86	85	51
G	90	27	64	28	40	22	55	61	65	35	71	80	86	85	58
DT5(p)	100	58	75	56	60	40	55	61	92	22	80	87	91	91	69
DT5(r)	40	64	55	50	60	67	55	94	48	29	33	43	48	61	53

Table 4.5: Precision-Recall using Nacin Algorithm

BC: Basic Configuration N: Neighbors NS: No Structure S: Structure G: Guesses DT5(p): Precision of Decision Tree Classifier (minNumObj = 5) DT5(r): Recall of Decision Tree Classifier (minNumObj = 5)

Chapter 5
Tool and Evaluation

In Chapter 3, we presented a user study to collect developer navigation traces while engaged in mapping high-level program features to elements in the source code. The data collected was used to train a decision tree classifier. We also reported on the performance of the classifier and analyzed the results of using the decision tree classifier on different training and test data sets. In this chapter we present a tool developed to classify the program elements in a developer's navigation trace and form a summary of the development session. The plug-in uses the decision tree classifier learned from the data collected in the user study to classify the elements. Furthermore, we present a small study performed to evaluate the tool and get user feedback on the generated summary.

5.1 Tool

We developed an Eclipse plug-in, TaCoML(**Ta**sk **Co**ntext using **M**achine **L**earning), to automate the process of deriving a summary of the development session using the navigation trace and integrate it into the developer's work flow. The plug-in monitors the developer's source code navigation within the IDE and records the navigation trace. It also makes note of source code elements modified by the developer. The plug-in then transforms the navigation trace to a form suitable for use with a decision tree classifier. In the transformation, the raw navigation traces are converted into the

```
count <= 1.0: false (200.0/30.0)
count > 1.0
|   count <= 5.0
|   |   noOfSiblings <= 17.0
|   |   |   totalDuration <= 2500.0: true (47.0/7.0)
|   |   |   totalDuration > 2500.0
|   |   |   |   noOfKeywordsInHandle <= 1.0
|   |   |   |   |   count <= 2.0: false (98.0/31.0)
|   |   |   |   |   count > 2.0
|   |   |   |   |   |   noOfCalledMethods <= 0.0: true (85.0/35.0)
|   |   |   |   |   |   noOfCalledMethods > 0.0: false (48.0/13.0)
|   |   |   |   noOfKeywordsInHandle > 1.0: true (24.0/4.0)
|   |   noOfSiblings > 17.0: false (20.0/1.0)
|   count > 5.0: true (94.0/19.0)
```

Figure 5.1: Decision Tree Classifier (minNumObj = 20)

tuple of attributes described in Table 3.2. The transcript elements are classified using the decision tree classifier and the set of elements forming the summary are presented to the user via the ConcernMapper plug-in. For each element in the summary, the plug-in also associates a numerical value between 0 and 100 representing a confidence measure. This value can be used as a filter to limit the list of visible elements in the ConcernMapper view. The changed program elements are included in the summary by default since elements that change during the session are by definition relevant to the task. For the same reason, they are also assigned a confidence value of 100.

The default classifier used in the plug-in is learned from the data collected in the user study described in the previous chapter. We used the transcripts from the fourteen experiments to train the C4.5 decision tree learning algorithm as implemented in the Weka machine learning toolkit. The decision tree was built with the parameter "minNumObj" value of 20. The particular parameter value was chosen because of the higher recall (56%) as well as good precision (64%) obtained using the training data. Of the ten attributes, the classifier uses five attributes, with the "count" attribute being the root of the decision tree. The output decision tree is stored in a file which is loaded at runtime by the plug-in and can be changed by updating the model stored in the file. Figure 5.1 shows the learned decision tree classifier that is used in the

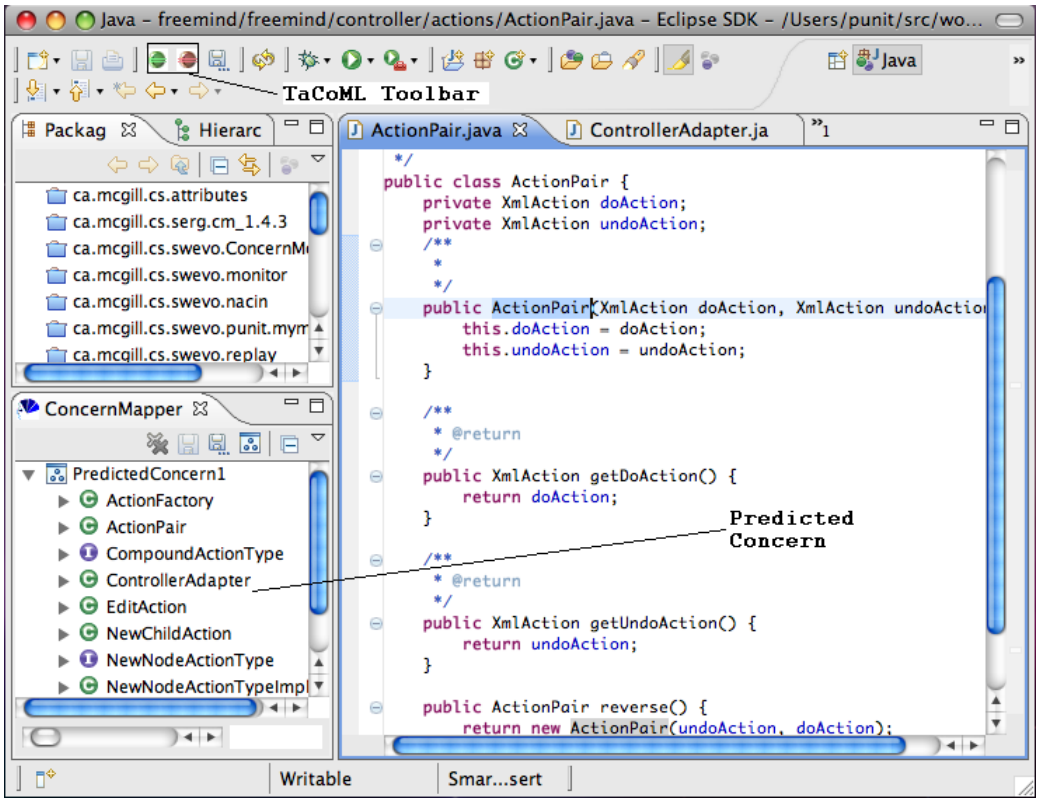

Figure 5.2: TaCoML Plug-in: Toolbar and Predicted Concern

plug-in.

The plug-in integrates into the Eclipse IDE by contributing toolbar items and menu items. For its functionality, the plug-in relies on two external plug-ins - JayFX[1] and ConcernMapper [20].[2] JayFX is used to query the project source code for program element relationships while ConcernMapper is used to present the user with a list of suggested program elements forming the task context. The plug-in toolbar and the predicted concern in the ConcernMapper view are shown in Figure 5.2. The developer indicates the start of a development session by clicking on the green "Start Session" button contributed to the Eclipse toolbar by the plug-in. Pressing the red "Stop Session" button signals the end of the session. These actions can also be accessed through the "Tacoml" menu item. On activation, the plug-in monitors the

[1] http://www.cs.mcgill.ca/~swevo/jayfx/
[2] http://www.cs.mcgill.ca/~martin/cm/

developer's source code navigation and records the list of program elements touched by the developer. When the developer signals the end of a development session, the plug-in builds a transformed transcript containing the attributes described in Table 3.2 for each program element. The transcript is evaluated using the decision tree and the set of elements labeled as belonging to the task context is added to the ConcernMapper as "predicted" concern (see Fig. 5.2). The developer can modify the predicted concern to add or delete elements from it.

5.2 Evaluation Study

We conducted a short user study to evaluate the plug-in and get user feedback on the session summary generated using the decision tree classifier. The format of the individual experiments of the study was structured similarly to the user study described in Chapter 3. The difference in the two studies was in the nature of tasks; we required the users to change the project source code to fix a bug, as opposed to discovering the code of a concern by source code exploration in the previous study. The subjects were unfamiliar with the project source code and hence needed to explore the code to understand the implementation of the feature related to the bug before changes could be made.

For this study we used Freemind[3], a mind mapping software written in Java. A mind map is a diagram used to represent words, ideas, tasks or other items linked to and arranged radially around a central key word or idea. It is used to generate, visualize, structure and classify ideas, and as an aid in study, organization, problem solving, and decision making. We used version 0.8.0 of Freemind which has more than 70k lines of code distributed over 617 types and 5,388 methods. Users were required to investigate, understand and fix a bug related to the "Undo child node creation" feature. The task description is provided in Figure 5.3.

We performed four instances of the task with the first being a pilot. The subjects of the study consisted of four developers experienced in Java development and

[3] `http://freemind.sourceforge.net`

In Freemind, users create a new node that is a child of the selected node by pressing the "insert" key and typing in the title of the node. If this action is undone by pressing Ctrl-Z, only the text of the node is removed, and the link remains, with a node with no text. This node is displayed as only a link. We would like to change this action so that both the node and the link are removed as a single undo action.

Find the code responsible for undo of creation of text node and setting of text. Make the changes necessary for both the actions to be undone by a single undo command.

Figure 5.3: Undo Child Node Creation

comfortable with using the Eclipse IDE. The subjects were provided with an Eclipse workspace setup with a copy of the Freemind project source code. The TaCoML plug-in was installed in the workspace and the developers were asked to activate it before beginning their task. There was no set time limit for the task but all the subjects managed to complete the task within an hour and fifteen minutes. At the end of the task the developers were asked to indicate the end of the task by signalling the end of the develpment task. The plug-in then calculated the set of elements predicted to belong to the task context and presented the results via the ConcernMapper view.

All four developers were successful in fixing the bug, though they differed in their approach and the actual code changes. At the end of the task we asked the developers to provide specific feedback on the elements in the suggested task context - whether they were important to the understanding of the task being performed or not - as well as highlight any important elements which were not included in the suggested list. The feedback from the developers was recorded for future reference.

The results of the evaluation study is presented in Table 5.1. "Unique Elements" is the number of unique methods and field visited by the developer while performing the task. "Edited" elements are the elements that were changed by the subject during the session. "Concern Size" is the size of the concern set predicted by the TaCoML plug-in. The predicted concern automatically included all the edited elements from the

Subject	Unique Elements	Edited	Concern Size	Interesting			
				X	E	R	Total
A	32	2	8	4	1	0	5
B	62	14	17	1	10	0	11
C	36	4	7	3	4	1	8
D	69	15	26	7	11	0	18

Table 5.1: Results of Evaluation Study
X = Explored; E = Edited; R = Random

development session as they definitely belonged to the task context. The developers' feedback on the number of elements of interest in the concern set is presented in the column labelled "Interesting". The interesting elements as indicated by the subject are further split by whether they were explored elements (X), edited elements (E), or were randomly added (R) by the tool for the purpose of the study.

In addition to the result presented in Table 5.1, except for pilot Subject 'A', we had introduced two random elements, selected from the session transcript, in the predicted concern set for each experiment. The addition of the random elements was to verify that the developers understood the code under investigation. All but one of the random elements was marked as not interesting by the subjects. The one random element marked as interesting can be attributed to the method belonging to a class which was core to the understanding and fixing of the bug.

As can be seen from Table 5.1, the edited elements form a large portion of the predicted concern. On further investigation we learnt that the actual code change to fix the bug was localized to one or two methods except in one case. Subject 'D' interpreted a much broader than intended scope for the task and designed a complex fix which required many new methods to be written. Of the remaining developers, subject 'B' chose to place debug prints to understand the code, which were later reverted. There were also a couple of elements where the code change was due to spurious key presses on the part of the developers. But the plug-in looked only at edits to determine changed elements and so failed to detect reverted changes or spurious

edits.

We had not anticipated the numerous edits performed by the developers. Many of the elements targeted by the edits were important for the understanding of the task context. In the plug-in implementation of the decision tree algorithm to predict task context, the edited elements bypassed the decision tree classifier and were automatically included in the predicted set of elements. Because of the implementation, although quite a few elements of interest were present in the predicted concern, they were included because of edit actions performed and not as the outcome of the decision tree classification. Therefore, we cannot use the predicted concerns to draw inferences on the performance of decision tree classifier for predicting concerns. However, we would like to point out the absence of irrelevant elements in the predicted concerns. In future user studies involving code changes, care should be taken to factor in the effects of user edits to the elements belonging to the concern.

In addition to the results presented above, we were also able to observe certain trends in the utility of the prediction technique and its corelation to the expertise of the developers using the tool. We noticed that the plug-in was most useful, i.e., it performed the best for developers who represented the average-case in terms of developer skills. In other words, the predicted concerns were of significantly poor quality for inexperienced as well as expert developers. This is in accordance with the observations from the previous user study and the subsequent analysis of the resulting navigation traces.

On the whole, we received positive feedback about the plug-in and its usage from the developers taking part in the study. Two of the four subjects('A' and 'C') were very impressed with the high signal-to-noise ratio exhibited by the filter algorithm in predicting the concern. The goal of the study, to evaluate the plug-in and get feedback on the predicted concerns, was successfully achieved.

Chapter 6
Conclusion and Future Work

Developers expend a lot of time to understand the context of a development task. Due to lack of good automated tools to record the task context the resources devoted towards understanding the task context are lost over time. Also, it is not easy to share the task context among developers working on related tasks. To overcome this problem, we trained a decision tree classifier, using the data collected during a user study, to automatically generate a summary of the development session. The training data consisted of attributes and a label computed from program navigation transcripts and code-elements to feature mappings generated by the subjects of the user study while engaged in source code exploration tasks. We developed TaCoML, an Eclipse plug-in, to integrate the learned decision tree classifier into the developer's work flow. The plug-in facilitates the recording of the task context by generating a summary of the important elements of the session.

To evaluate the decision tree learning algorithm, we compared its precision and recall with the Nacin algorithm and two variants using the count metric. We found that the decision tree classifiers have better precision than all the configurations of the Nacin algorithm suggested by Robillard et al. [17]. When compared to the variants of the count metric the decision trees have higher precision in most cases. In contrast, when comparing recall, the decision trees performed slightly worse than both Nacin as well as the variants using the count metric. We observed a trade-off between precision and recall as we varied the parameter to control the size of the learned decision tree.

Thus, depending on requirements it is possible to control whether more importance is placed on generating a smaller but more precise summary set, or a larger but possibly less accurate set of elements representing the task context.

We also conducted a small user study to evaluate the TaCoML plug-in. In this study we asked developers to perform a software change task and at the end of the task generated a summary of the elements representing the task context. The developers rated the summary elements on the basis of their usefulness. Due to the large number of edits performed by the subjects, which were automatically included in the generated summary, it was difficult to ascertain the quality of the predicted elements. But the feedback obtained on the tool and its utility were positive. In the future, user study design involving development tasks should take into account the developer behavior of modifying the program source code to understand it.

The learned decision trees also provided insight into the relative importance of the different attributes. Although the structure of the trees varied with training data and configuration parameters some of the attributes were more commonly encountered in the learned tree than others. Particularly, the 'count' attribute was prominent in most resulting trees. This validates the common notion that the more often the developer accesses certain elements, the more important they are. But the presence of additional attributes helped the learning algorithm to further refine the decision tree and obtain better precision. The additional precision obtained suggests that the count attribute alone is insufficient and better results can be obtained by including additional information through other attributes.

We have successfully evaluated the utility of using machine learning algorithms for the task of classification of navigation transcript elements to generate a summary of the development session or the task context. Our results indicate that the decision tree learning algorithm had better precision than other approaches we compared with, but the recall for the machine learning algorithms needs further improvement. It should be noted that machine learning methods (especially, classical supervised learning, such as decision trees) are designed to optimize precision, and *not* recall. As a result, the fact that we obtained very good precision and not that great recall is not suprising and should be considered a good result.

6.1 Future Work

Although, our initial attempt to use machine learning techniques for the task of program navigation analysis have been successful, the results obtained so far suggest further directions for future research. Decision trees are amongst the simpler models that can be used to represent the outcome of a learning process. We believe that a model that more closely represents the process of discovering the task context that a developer follows will lead to improvements in the quality of the generated summary. In the future, we would like to investigate the effectiveness of using more complex machine learning models and algorithms to generate the summary. One of the possibilities is the use of time-series learning methods which would treat the navigation trace as a sequence rather than trying to summarize it in a fixed set of attributes. Also, we would like to investigate additional attributes that better encode the semantics and information considered important by the developers. In conjunction with the work on learning models and attributes, we would like further improve the Eclipse plug-in based on user feedback. Based on the outcome of the research, we would like to further add support for other methods to generate the session summary. Another finding of our investigations was the differences in the program investigation styles and patterns between developers with varying skill levels. We would like to investigate the differences and evaluate the effectiveness of using different classifiers and models based on developer skills. This would allow developers to fine tune the summary generating tool for their development style.

Bibliography

[1] A. V. Aho. *Pattern Matching in Strings*. Academic Press, 1980.

[2] William W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*, pages 115–123. Morgan Kaufmann, 1995.

[3] Davor Cubranic and Gail C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th IEEE International Conference on Software Engineering*, pages 408–418, 2003.

[4] Robert DeLine, Mary Czerwinski, and George Robertson. Easing program comprehension by sharing navigation data. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 241–248, 2005.

[5] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[6] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? Assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, January 2001.

[7] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd ACM International Conference on Aspect-Oriented Software Development*, pages 178–187, 2003.

[8] Mik Kersten and Gail C. Murphy. Mylar: A Degree-Of-Interest model for IDEs. In *Proceedings of the 4th ACM International Conference on Aspect-Oriented Software Development*, pages 159–168, 2005.

[9] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for Maintenance-Oriented IDEs: A detailed study of corrective and perfective maintenance tasks. In *Proceedings of the 27th ACM International Conference on Software Engineering*, pages 126–135, 2005.

[10] Andrew J. Ko and Bob Uttl. Individual differences in program comprehension strategies in unfamiliar programming systems. *International Workshop on Program Comprehension*, 2003.

[11] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997.

[12] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Cubranic. The emergent structure of development tasks. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 33–48. Springer-Verlag, 2005.

[13] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., 1993.

[14] Martin P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, 2005.

[15] Martin P. Robillard, Wesley Coelho, and Gail C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[16] Martin P. Robillard and Gail C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th ACM International Conference on Software Engineering*, pages 406–416, 2002.

[17] Martin P. Robillard and Gail C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 25th IEEE International Conference on Automated Software Engineering*, volume 00, pages 225–234, 2003.

[18] Martin P. Robillard and Gail C. Murphy. FEAT: A tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th ACM International Conference on Software Engineering*, pages 822–823, May 2003.

[19] Martin P. Robillard, David Shepherd, Emily Hill, K. Vijay-Shanker, and Lori Pollock. An empirical study of the concept assignment problem. *Technical Report SOCS-TR-2007.3*, June 2007.

[20] Martin P. Robillard and Frédéric Weigand-Warr. ConcernMapper: Simple view-based separation of scattered concerns. In *Proceedings of the ACM Eclipse Technology Exchange at OOPSLA*, pages 65–69, 2005.

[21] Janice Singer, Robert Elves, and Margaret-Anne Storey. NavTracks: Supporting navigation in software maintenance. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334, 2005.

[22] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 21–35, 1997.

[23] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.

[24] Andreas Zeller. The future of programming environments: Integration, synergy, and assistance. In *Proceedings of the 2007 IEEE Future of Software Engineering*, pages 316–325, 2007.

[25] Du Zhang and Jeffrey J. P. Tsai. *Machine Learning Applications In Software Engineering (Series on Software Engineering and Knowledge Engineering)*. World Scientific Publishing Co. Inc., 2005.

[26] Du Zhang and Jeffrey J. P. Tsai. *Advances in Machine Learning Applications in Software Engineering*. IGI Publishing, 2007.

[27] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, June 2005.

[28] Lijie Zou, Michael W. Godfrey, and Ahmed E. Hassan. Detecting interaction coupling from task interaction histories. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 135–144, 2007.