Predicting Source Code Changes by Mining Revision History

by

Annie Tsui Tsui Ying

B.Sc. (Honours), University of British Columbia, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

October 2003

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Oct 10, 2003

DE-6 (2/88)

Abstract

Software developers are often faced with modification tasks that involve source which is spread across a code base. Some dependencies between source, such as the dependencies between platform dependent fragments, cannot be determined by existing static and dynamic analyses. To help developers identify relevant source code during a modification task, we have developed an approach that applies data mining techniques to determine change patterns—files that were changed together frequently in the past—from the revision history of the code base. Our hypothesis is that the change patterns can be used to recommend potentially relevant source code to a developer performing a modification task. We show that this approach can reveal valuable dependencies by applying the approach to the Eclipse and Mozilla open source projects, and by evaluating the predictability and interestingness of the recommendations produced.

Contents

List of Tables

List of Figures

Acknowledgements

I'm so grateful to have Professors Gail Murphy and Raymond Ng as my supervisors. They've constantly guided and encouraged me, even though I sometimes I lost conscious in the direction and did stupid things.

In addition, thank you so much Gail, for everything: introducing me to software engineering research, teaching me to be critical about research, and being amazingly prompt and helpful, especially on commenting the endless number of revisions of the thesis and the paper. I don't think I would even do software engineering research if I didn't know Gail.

I'd like to express special thanks to Mark Chu-Carroll for ideas about the thesis topic and his valuable comments and encouragement throughout my research. I'd also like to thank Rob O'Callahan for his help on the Mozilla validation and his comments about the general validation strategy.

I'm indebted to Professor Kris de Volder and lab-mate Brian de Alwis, who proofread my thesis and provided me helpful feedback, even with the tight thesis submission deadline.

I'd like to thank my colleagues, especially the following people: Davor Cubranic for his help on the Hipikat database; my lab-mate Chris Dutychn who always lends his ears for my whines when I discover bugs in my program or bad results; Carson Leung and Martin Robillard who have been my mentors and role models since my undergraduate years; and Miryung Kim for her encouragement during my most discouraged period. All these nice things really mean a lot to me.

Finally, I'd like to thank my family and friends for their endless support.

ANNIE TSUI TSUI YING

The University of British Columbia
October 2003

Chapter 1

Introduction

Software is consistently changing and evolving to new circumstances. The existing source code may be well-modularized with respect to a limited number of anticipated modifications but is unlikely be well-modularized for all modification tasks [16]. Modifications to software do not always involve changes to a single, well-encapsulated module, so software developers are often faced with modification tasks that involve changes to source code artifacts that are spread across the code base [19].

1.1 Motivation

Developers may use static and dynamic analyses to identify relevant parts of the code in a modification task (e.g., [20, 1]). Such analyses can help a developer with determining the scope of the task, but they cannot always identify *all* of the code relevant to the change.

For example, in a software system with a module whose interface represents multiple platforms but for which there is a different implementation for each plat-

form, whenever a change is made to one platform version, changes are often required in the other platform versions as well. Since different target platforms are involved, static analysis may be hard to apply in the absence of a cross-compilation environment. In addition, dynamic analysis is costly to apply for inferring relationships across versions of different platforms because it can involve running different platform versions and integrating the results from separate runs.

As another example, consider changing code in different programming languages, such as changing Java source code that manipulates textual data in XML format. In static analyses on Java source code, XML data is typically external to the analyses. Tracing-based dynamic analysis can be hard to apply because the tracer needs to generate events with a cross-language event API.

1.2 Thesis statement

When relevant parts of source code are changed together repeatedly, relationships between such parts of source code—including structural dependencies and other dependencies such as cross-platform and cross-language dependencies—may become apparent from analysis of the development history.

We hypothesize that *change patterns*—files that have changed together *frequently enough* in the development history— can be used to recommend potentially relevant source code to a developer performing a modification task. The process of determining change patterns is introduced in the next section, Section 1.3.

1.3 Overview of our approach

Our approach consists of mining *change patterns*—files that have changed together *frequently enough*—from the source code versioning data, and then recommending files from relevant patterns as a developer performs the modification task. Specifically, as a developer starts changing files, denoted by the set f_S, our approach recommends additional files for consideration, denoted by the set f_R. Our initial focus has been on the use of association rule mining to determine the change patterns. In this thesis, we report on our use of two different association rule mining algorithms, each of which has its own notion of "frequently enough": frequent pattern mining [2], which is based on frequency counts, and correlated set mining [5], which is based on the chi-squared test.

To assess the utility of our approach, we evaluate the recommendations our tool can make on two large open source projects, Eclipse[1] and Mozilla[2], based on the *predictability* and the *interestingness* of the recommendations. The predictability criteria measures the coverage and the accuracy of the recommendations against the files actually changed during modification tasks recorded in the development history. The interestingness criteria measures the value of correct recommendations. Our results show that we can provide useful recommendations that complement the information obtained through static or dynamic analyses.

1.4 Motivating Scenarios

To illustrate the difficulty developers sometimes face in finding relevant source code during a modification task, we outline the changes involved in three modification

[1]URL at `http://www.eclipse.org/`
[2]URL at `http://www.mozilla.org/`

tasks[3]: two from the Mozilla revision history, and one from the Eclipse revision history. Mozilla is an open source web browser primarily written in C++. Eclipse is an open source integrated development environment primarily implemented in Java.

1.4.1 Scenario 1: Incomplete change

The Mozilla web browser includes a web content layout engine, called Gecko, that is responsible for rendering text, geometric shapes, and images. Modification task #150339 for Mozilla, entitled "huge font crashes X Windows", reports on a bug that caused the consumption of all available memory when a web page with very large fonts displayed. As part of a solution[4] for this modification task, a developer added code to limit the font size in the code for a module in one layer of the system `gtk`, but missed a similar required change in the code for a dependent module `xlib`. The comments in the modification task report include:

> 2002-06-12 14:14: "This patch [containing the file `gtk/nsFontMetricsGTK.cpp`] limits the size of fonts to twice the display height. I've actually wanted fonts bigger than the display at times, but limiting font size to twice the display height seems reasonable."
>
> 2002-06-12 14:37: "The patch misses the Xlib gfx version." [A patch was later submitted with the correct changes in the X-windows font handling code in the file `xlib/nsFontMetricsXlib.cpp`.]

The source code in `gtk/nsFontMetricsGTK.cpp` does not directly reference the code in `xlib/nsFontMetricsXlib.cpp`. However, an analysis of the CVS revision history for Mozilla indicates that these two files were changed 41 times together

[3] A *modification task* is often referred as a *bug*.

[4] We refer to the files that contribute to an implementation of a modification task as a *solution*.

in the development of Mozilla.

When we applied our approach to Mozilla, we extracted a change pattern with three files: `gtk/nsFontMetricsGTK.cpp`, `xlib/nsFontMetricsXlib.cpp`, and `gtk/nsRenderingContextGTK.cpp`. Changing the `gtk/nsFontMetricsGTK.cpp` triggers a recommendation for the other two files, one of which had been missed in the first patch.

1.4.2 Scenario 2: Reusable code

Modification task #123068 in the Mozilla development involved adding the host-name of a SMTP server in the error message that results from a failed attempt to connect to the server. A developer initially submitted a solution in which the host-name of the SMTP server was constructed in a file representing the message-send operation (`nsMsgSend.cpp`). Another developer later suggested a better way of retrieving the host-name of the server by reusing the code in the SMTP server in the file `nsSmptServer.cpp`.

Changing `nsMsgSend.cpp` could trigger our approach to recommend `nsSmptServer.cpp` in which the first developer did not consider. Our approach also suggested 15 other files that are not in the solution. Had the developer who submitted the initial change used our approach, he might have investigated the file `nsSmptServer.cpp`.

1.4.3 Scenario 3: Narrowing down structural relationships

Eclipse is a platform for building integrated development environments. "Quick fix" is a feature of the Java development support in Eclipse that is intended to help a developer easily fix compilation errors. Modification task #23587 describes

a missing quick fix that should be triggered when a method accesses a non-existent field in an interface.

The solution for the modification task involved the class `ASTResolving`[5] in the text correction package. This class was structurally related to many other classes: It was referenced by 45 classes and references 93 classes; many of these classes were involved in a complicated inheritance hierarchy. This large number of structural dependencies may complicate the determination of which classes should be changed with `ASTResolving` when replying solely on static information.

Having applied our approach, we found that `ASTResolving` had been changed more than ten times in the revision history in conjunction with three classes: `NewMethodCompletionProposal`, `NewVariableCompletionProposal`, and `UnresolvedElementsSubProcessor`. These three classes were part of the solution for the modification task. The other four classes that were part of the solution were not recommended; however, because they had not been modified together a sufficient number of times.

1.5 Outline of the thesis

This chapter presented the motivation, thesis statement, and an overview of our approach. The remaining chapters of the thesis are organized as follows. Chapter 2 explains our approach and implementation issues. Chapter 3 presents an evaluation of our approach on Eclipse and Mozilla. Chapter 4 provides a discussion of outstanding issues. Chapter 5 covers related work. Finally, Chapter 6 concludes, highlighting the contributions of our research.

[5] In Java, the name of a file that contains a publicly accessible class `C` is `C.java`.

Chapter 2

Approach

Our approach consists of three stages (Figure 2.1). In the first stage, data is extracted from the software configuration management system and is pre-processed to be suitable as input to a data mining algorithm. In the second stage, we apply an association rule mining algorithm to form change patterns. In the final stage, we recommend relevant source files as part of a modification task by querying against mined change patterns. Having extracted the change patterns in the first two stages, we do not need to re-generate the change patterns each time we query for a recommendation.

In this chapter, we describe each of the three stages in our approach, covering Sections 2.1 to 2.3. In addition, we present some issues with the implementation of our prototype in Section 2.4.

2.1 Stage 1: Data pre-processing

Our approach relies on being able to extract information from a software configuration management system that records the history of changes to the source code

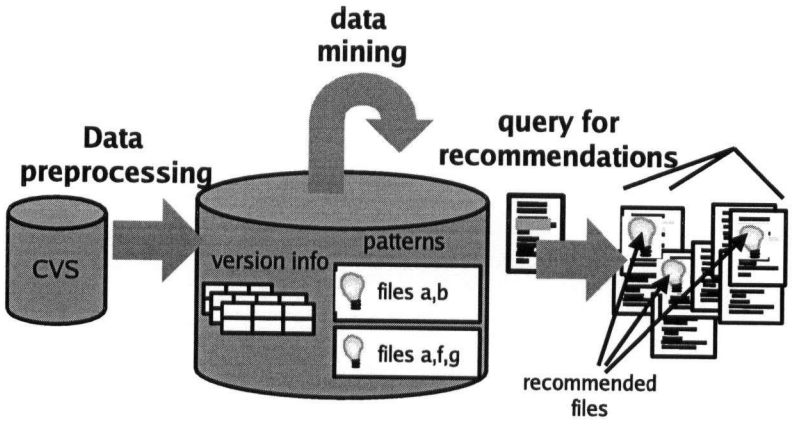

Figure 2.1: Three stages in the approach

base. In addition to the actual changes made to the source code, these systems typically record meta-data about the change, such as the timestamp, author and comments on the change. Most of these systems manage software artifacts using a file as unit. Some support finer-grained artifacts such as classes and methods in an object-oriented programming language (e.g., Coop/Orm [11]). Our initial focus has been on revision histories stored in a CVS[1] repository, which records and manages changes to source files.

Before attempting to find patterns in the change history, we need to ensure the data is divided into a collection of atomic change sets, and we need to filter atomic changes that do not correspond to meaningful tasks.

[1]URL at `http://www.cvshome.org/`

2.1.1 Identifying atomic change sets

The first pre-processing step involves determining which software artifacts—in our case, files—were checked in together. This step is not needed when the history is stored in a software configuration management system that provides atomic change sets, such as ClearCase,[2] in which the concept of an atomic change set is built into the system. However, other systems such as CVS, which is used for the systems we targetted in our evaluation, do not track this information, and as a result, we must process the revision history to attempt to recreate these sets. We form the sets using the following heuristic: an atomic change set consists of file revisions that were checked in by the same author with the same check-in comment close in time. We follow Mockus and colleagues in defining proximity in time of check-ins by the check-in time of adjacent files that differ by less than three minutes [14]. Our proposed architecture in Figure 2.1 works as well even if we use different heuristics for identifying transactions.

2.1.2 Filtering

The second pre-processing step consists of eliminating transactions consisting of more than a certain number of files because these long transactions usually do not correspond to meaningful atomic changes. In our research, we chose 100 as the threshold for the maximum cardinality of a transaction. An example is when an integrated development environment, such as Eclipse, is used to remove unnecessary import declarations in all of the Java files in a project. This organize-import operation changes many files but does not correspond to a meaningful modification task. However, a developer can check in source code that contains meaningful changes in

[2]URL at `http://www.rational.com/index.jsp`

addition to changes resulted from the organize-import operation.

2.2 Stage 2: Association rule mining

Association rule mining finds relationships among items by determining sets of items that occur *frequently enough* amongst the transactions in a database [2]. One well-known use of such associations is marketing. Organizations use purchase associations of the form "when a customer purchases item x, the customer is likely to also purchase item y" to establish strategies for targetting customers. In our context, we apply association rule mining to find sets of source files that tend to change together.

These associations correspond to relationships between items, each with a single, boolean-valued attribute. This describes whether or not an item—or file in our context—occurs in a transaction. There are other extensions to this basic type of associations, including quantitative associations involving numeric attributes [17], multi-dimensional associations involving multiple attributes [10], and multi-level associations that describe associations with different levels of abstraction [18]. Our initial focus has been on associations on related items with a single, boolean-valued attribute. In this thesis, we investigate two algorithms that find such associations: frequent pattern mining [2] which is based on frequency counts, and correlated set mining [5], which is based on the chi-squared test.

2.2.1 Frequent pattern mining

The idea of frequent pattern mining is to find recurring sets of items among transactions in a database D [2]. As mentioned before, the items are source files. The strength of the pattern $\{s_1, \ldots, s_n\}$ is measured by *support*, which is the number of

transactions in D containing $s_1, \ldots, s_n$. A frequent pattern describes a set of items that has support greater than a predetermined threshold *min_support*.

The problem of finding all frequent patterns efficiently is not trivial because the performance can be exponential with respect to the number of items in D when the support threshold *min_support* is low. Efficient algorithms to this problem have been proposed (e.g., [3, 15, 8]).

In the rest of this section, we outline an algorithm called Apriori that uses a breadth-first approach to find frequent patterns [3]. The drawbacks of a breadth-first approach motivates us to consider another algorithm that uses a depth-first approach to mine frequent patterns on a compact tree data structure FP-tree [8]. In our research, we use the algorithm that is based on FP-tree to find change patterns. We also describe the modifications we made to the FP-tree based algorithm.

Breadth-first approach

Breadth-first frequent pattern mining algorithms find all frequent patterns of a certain cardinality k and then use them to find frequent patterns with cardinality $k+1$. The Apriori algorithm is one algorithm [3]. Apriori first finds all frequent items by counting the number of transactions containing each item. The algorithm then finds frequent patterns of cardinality two by first generating candidate patterns using the frequent items and then verifying if the support of candidate patterns is greater than the threshold *min_support*. The algorithm continues until all candidate patterns of cardinality k do not have sufficient support.

The performance of this algorithm can be costly. One reason is that the number of candidate sets can be exponential with respect to the number of items. For example, Eclipse–one of the two projects we investigated–has more than 20,000

files during two years of development, and the number of candidate sets of size 2 alone is exponential to 20,000 in the worst case. Pruning on sets that cannot possibly result in a frequent pattern can reduce the number of candidate sets, for example, as in the Apriori algorithm. However, the amount of reduction on the number of candidate sets depends on whether *min_support* is high enough to prune away sets cannot satisfy the support threshold. In addition, each step of finding frequent patterns of a certain cardinality takes one scan over the whole database; hence the whole process of finding frequent patterns of all cardinality requires k scans of the database if the longest pattern has cardinality k.

Depth-first approach

The algorithm we chose to find frequent patterns uses a compact data structure called FP-tree to encode a database [8]. The idea of the algorithm is to find frequent patterns by a depth-first approach of recursively mining *a pattern* of increasing cardinality from the data structure FP-tree that encodes D, as opposed to a breadth-first approach of finding *all patterns* of the same cardinality before finding patterns of a larger cardinality in the Apriori algorithm. In a FP-tree, each node represents a frequent item in D, except for the root node that represents an empty item. Each path from the root to a node in the FP-tree corresponds to a collection of transactions in D, each of which contains all the items on the path. Items in a path are in descending order of support (and in lexicographical order of item name if two items have the same support). The mining process involves decomposing a FP-tree associated with database D into smaller FP-trees, each of which corresponds to a partition of D. This divide-and-conquer approach allows the determination of frequent patterns to focus on the decomposed database rather than the whole

database.

More specifically, the algorithm for finding all patterns from an FP-tree is as follows. For each frequent item s in D, we construct a FP-tree from transactions in D containing s. To determine whether the pattern $\{s_1, \ldots, s_{k-1}, s_k\}$ has sufficient support in D, we find all transactions containing $\{s_1, \ldots, s_{k-1}\}$, referred to as D', by traversing the FP-tree on D, with the paths in the FP-tree on D representing transactions in D. If s_k have sufficient support in D', then $\{s_1, \ldots, s_{k-1}, s_k\}$ also have sufficient support in D and the mining process continues on D'. We repeat the process until the FP-tree is empty or contains only one single branch. A FP-tree with only a single branch means that all transactions encoded in the FP-tree contain the same patterns. We further illustrate this algorithm with an example in Appendix A.

We have modified the FP-tree mining algorithm so that we do not continue to generate subsets of patterns from a single-branched tree. In addition, we only output maximal patterns[3] with respect to D from the mining process, as opposed to all patterns and their subsets. Storing only the maximal patterns does not affect the recommendations generated by the patterns because recommendations generated by all patterns are contained in the recommendations generated using the maximal patterns. The recommendation step is described in more detail in Section 2.3.

Constructing a FP-tree from a database D requires two scans of the transactions in D, one to determine the frequent items and the second to construct the FP-tree. The mining process requires scanning the FP-tree and constructing new, smaller FP-trees.

Frequent pattern mining uses only *support* to determine the frequency of

[3]A pattern in D is maximal if it is not a subset of any other pattern in D.

items occur *jointly* but does not consider how the joint frequency is different from the frequency of *individual* items [5]. To address this issue, we investigate another algorithm called correlated set mining.

2.2.2 Correlated set mining

The idea of correlated set mining is to find all sets of items with sufficient correlation [5]. As in frequent pattern mining, the items are source files. Correlation among items $\{s_1, \ldots, s_n\}$ is measured by the chi-squared test, which determines if there is a statistically significant difference between the frequency of items occurring (and not occurring) *jointly* and the frequencies of items occurring (and not occurring) *individually.* A set of items are correlated with α significance if the chi-squared value from the test exceeds some threshold, which can be obtained from statistics tables. Our approach uses a 95% significance level with threshold equal to 3.84.

Equation 2.1 provides a formula for computing the chi-squared value χ^2 between two items in a set $\{s_1, s_2\}$. In the equation, we overload the notation s_1 to denote the condition for a transaction of containing s_1 and the notation $\overline{s_1}$ to denote the condition for a transaction of *not* containing s_1. The notation D_{ij} denotes a set of transactions in D, each of which satisfies conditions i and j. For example, $D_{s_1\overline{s_2}}$ denotes transactions in D, each of which contains item s_1 but not item s_2.

$$\chi^2 = \sum_{i \in \{s_1, \overline{s_1}\}} \sum_{j \in \{s_2, \overline{s_2}\}} \frac{(|D_{ij}| - E(D_{ij}))^2}{E(D_{ij})} \tag{2.1}$$

Intuitively, the chi-squared value χ^2 is the sum of deviation (the ratio inside the double summation) between the *observed frequencies* ($|D_{ij}|$'s) and *expected frequencies* ($E(D_{ij}$'s)) in four correlations. Each of the four correlations represents

	s_1	$\overline{s_1}$	row Σ
s_2	$\lvert D_{s_1 s_2}\rvert$	$\lvert D_{\overline{s_1} s_2}\rvert$	$\lvert D_{s_2}\rvert$
$\overline{s_2}$	$\lvert D_{s_1 \overline{s_2}}\rvert$	$\lvert D_{\overline{s_1} \overline{s_2}}\rvert$	$\lvert D_{\overline{s_2}}\rvert$
col Σ	$\lvert D_{s_1}\rvert$	$\lvert D_{\overline{s_1}}\rvert$	$\lvert D\rvert$

Table 2.1: Transaction counts table used in correlated set mining algorithm

whether each, both, or neither of the items are present in transactions in D; in our context, the four correlations represent whether each, both, or neither of the files are modified together. The observed and expected frequencies depend on the number of transactions containing or not containing the combination of two items, which are listed in Table 2.1. The middle two columns in the table denotes transactions containing or not containing s_1 and the middle two rows denotes transactions containing or not containing s_2. The *observed frequencies* are based on the joint probability of a transaction containing or not containing the two items, represented in the middle four cells of Table 2.1. The *expected frequencies* are based on the assumption that two items are independent of each other, calculated based on the product of the marginal probabilities of a transaction containing or not containing the individual items, given by $E(D_{ij}) = \frac{|D_i||D_j|}{|D|}$ for $i \in \{s_1, \overline{s_1}\}$ and $j \in \{s_2, \overline{s_2}\}$.

Similar to frequent pattern mining, the problem of finding all correlated sets efficiently is a non-trivial problem because the performance can be exponential with respect to the number of items in D. One algorithm [5] follows a similar approach of the Apriori algorithm, which is to use breadth-first approach of finding all correlated sets of a certain cardinality and use them for finding correlated sets of one larger cardinality.

Because correlations must be calculated for each pair of items in a pattern, this algorithm does not scale as the number of items in a pattern increases. In this

thesis, we implemented the correlated sets mining algorithm to find patterns of size two. Another potential weakness of the correlated set mining algorithm is that it is not applicable on data where small frequency counts occur in the transactions in D because the chi-squared values χ^2 of such data can change dramatically given small changes in data [5]. A rule of thumb suggested by statistics texts is that the expected values of the items—calculated from frequencies of items occurring individually—should be greater than 1.

2.3 Stage 3: Query

Applying a data mining algorithm to the pre-processed data results in a collection of change patterns. Each change pattern consists of the names of source files that have been changed together frequently in the past. To provide a recommendation of files relevant to a particular modification task at hand, the developer needs to provide the name of at least one file that is likely involved in the task. The files to recommend are determined by querying the relevant patterns to find those that include the identified starting file(s); we use the notation $f_S \rightarrow f_R$ to denote that the set of files f_S results in the recommendation of the set of files f_R. When the set of starting file has cardinality of one, we use the notation $f_s \rightarrow f_R$.

The relationship $\rightarrow$ is symmetric: since each pattern describes a set of files that change together, the recommendation $f_1 \rightarrow f_2$ implies $f_2 \rightarrow f_1$. However, because recommendations are formed by patterns that contain the initial files f_S in $f_S \rightarrow f_R$, the relation $\rightarrow$ is not necessarily transitive; if $f_1 \rightarrow f_2$ and $f_2 \rightarrow f_3$, it may not be the case that $f_1 \rightarrow f_3$.

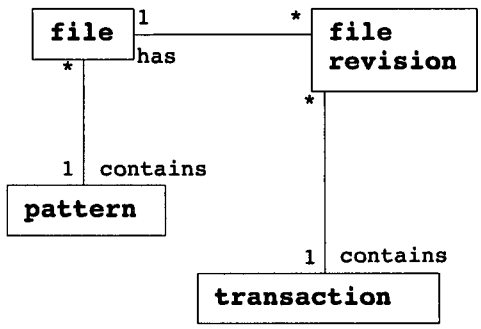

Figure 2.2: Data linkage schema in the mining process

2.4 Data schema and collection

Figure 2.2 shows the schema that represents four types of data related to our change pattern mining process:

- `transaction` represents transaction data on file revisions that are checked into the source code repository together, computed as in the data pre-processing step described in Section 2.1;
- `pattern` represents change pattern data on files modified together repeatedly generated from the data mining process described in Section 2.2;
- `file` represents file data including full path name of the file;
- `file revision` represents file revisions associated with each file.

In our implementation, we store these data in a relational database, with each type of data corresponding to a separate table in the database. We reuse the

database infra-structure from Hipikat [6], which uses MySQL open source database[4] to store data related to CVS file revisions, modification task information, email messages and newsgroup messages. The `file` and `file revision` tables are part of the hipikat database, whereas the `pattern` and `transaction` tables are specific to our change pattern mining process.

Populating the database with the schema illustrated in Figure 2.2 involves retrieving the CVS log files using the `cvs log` command. Each of these log files corresponds to a file in the CVS repository and contains revision information including time, author, and check-in comments. We parse the log files and put the CVS-related information into the `file` and `file revision` tables. Having retrieved the CVS-related information, we apply pre-processing to form transactions and populate transaction information to the `transaction` table. We then apply a data mining algorithm to generate change patterns and populate the `pattern` table.

Storing data in a relational database enforces a consistent data format, enabling different algorithms to be substituted for each of the data pre-processing, data mining, and querying stages. In addition, a relational database provides query capabilities that are useful in the implementation of the data mining algorithms.

Specifically, identifying transactions depends on the check-in time and the check-in comments of the file revisions. However, the log files given by the `cvs log` command is indexed by files, not by time or comments. Caching the information from the CVS repository to our database allows us to query more efficiently. Moreover, using a central repository allows developers in a team to share the change patterns and thus amortizes the cost of populating the database with file revision information and mining for change patterns.

[4]URL at `http://www.mysql.com/`

Chapter 3

Evaluation

In this chapter, we evaluate our hypothesis that change patterns can help recommend relevant source code in a modification task. First, section 3.1 presents our strategy of the evaluation based on the predictability and interestingness criteria. Section 3.2 describes the settings of the evaluation, including data collection, parameters chosen for the algorithms, and statistics about the data. Section 3.3 shows the results based on our evaluation strategy. Section 3.4 provides some performance statistics on our approach. Finally, Section 3.5 summarizes the evaluation.

3.1 Evaluation strategy

To assess the utility of change patterns in a modification task, we need to use realistic tasks so that the evaluation results on the evaluated target projects can be generalized to other projects. To satisfy this requirement, we applied our approach to the development histories of two large open-source systems: Eclipse and Mozilla. We further discuss issues about generalizability of our results in Section 4.3.

The validation process involves taking the code changed in a modification

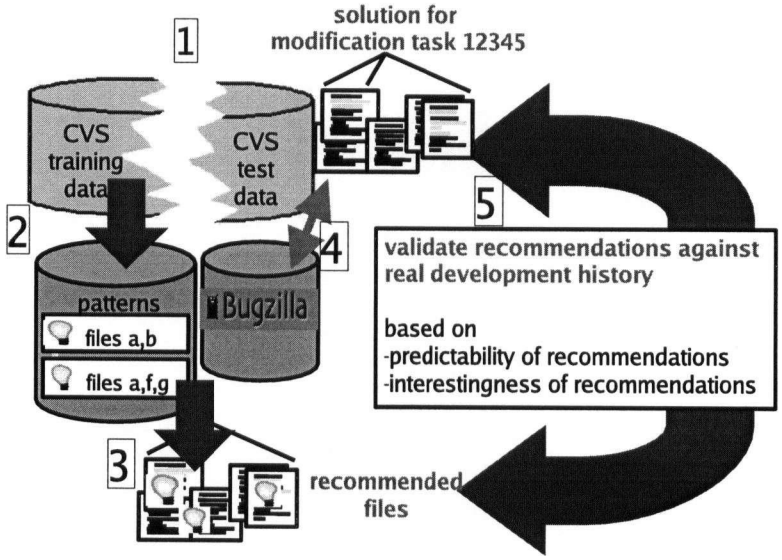

Figure 3.1: Validation process

task and verifying that by the recommended code given by our approach. Figure 3.1 illustrates the validation process. First, we divide the historical information considered into training data and test data (Figure 3.1, label 1). The training data is used to generate change patterns that are then used to recommend source for the test data (Figure 3.1, labels 2 and 3).

To determine if our approach can provide "good" recommendations, we investigated the recommendations in the context of completed modification tasks made to each system. These modification tasks are recorded in each project's Bugzilla bug tracking system, which has evolved to also keep track of enhancement tasks. We refer to both bugs and enhancements as *modification tasks*, and we refer to the files that contribute to an implementation of a modification task as a *solution.*

Since Bugzilla does not record which source files are involved in a solution of

a modification task, we use heuristics based on development practices to determine this information (Figure 3.1, label 4). One common practice is that developers record the identifier of the modification task upon which they are working as part of the CVS check-in comments for a solution of the modification task. Another common practice is that a developer commits the files corresponding to a solution of the modification task into CVS close to the time at which they change the status of the modification task report to "fixed" [6]. We chose tasks for which the files involved in the solution were checked in during the time period identified as the test data and for which at least one file involved in the solution was covered by a change pattern extracted from the training data. We ignore modification tasks for which we could not recover the files associated with the solution using the heuristics.

To recommend relevant files using our approach, at least one file that is likely involved in the solution must be specified by the developer. In our validation, we chose to specify exactly one file f_s to generate a set of recommended files f_R because this represents the minimum amount of knowledge a developer would need to apply our approach. We evaluate the usefulness of the recommended files f_R in terms of two criteria: predictability and interestingness, described in Sections 3.1.1. The evaluation strategy is presented in the rest of this section and is illustrated in Figure 3.1 (label 5).

3.1.1 Criterion 1: Predictability

The predictability of the recommendations is measured in terms of *precision* and *recall*. The precision of a recommendation $f_s \rightarrow f_R$ refers to the accuracy of the recommendations and is measured by the fraction of recommendations f_R that did contribute to the files in the solution (denoted by f_{sol}) of the modification task,

as shown in Equation 3.1. The recall of a recommendation $f_s \rightarrow f_R$ refers to the coverage of the recommendations and is measured by the fraction of files in the solution (denoted by f_{sol}) that are recommended, shown in Equation 3.2.

$$precision_{f_R} = \frac{|f_R| \cap |f_{sol}|}{|f_R|} \tag{3.1}$$

$$recall_{f_R} = \frac{|f_R| \cap |f_{sol}|}{|f_{sol}|} \tag{3.2}$$

3.1.2 Criterion 2: Interestingness

Even if a recommendation is applicable, we have to consider whether or not the recommendation is *interesting*. For example, a recommendation that a developer changing the C source file `foo.h` should consider changing the file `foo.c` would be too obvious to be useful to a developer. To evaluate recommendations in this dimension, we assign a qualitative interestingness value to each recommendation of one of three levels—*surprising*, *neutral*, or *obvious*—based on structural and non-structural information that a developer might easily extract from the source.

Structural information refers to relationships between program elements that are stated in the source using programming language constructs. The "Structural relationship" column of Table 3.1 lists the structural relationships between two source code fragments in the Java programming language that we consider in our criteria. The "Granularity" column indicates the unit of the source code fragments involved in the relationship: "coarse" indicates that the fragments are at a class or interface granularity; "fine" indicates that the granularity of the fragments is finer than class-granularity, e.g., a method-granularity. The last column provides a description for each relationship.

Structural relationship	Granularity	Description
(reads, m, f)	fine	Method m accesses the value of field f.
(writes, m, f)	fine	Method m writes a value to field f.
(calls, m_1, m_2)	fine	Method m_1 contains a method invocation that can bind to m_2.
(creates, m, c)	fine	Method m creates an object of class c.
(checks, m, c)	fine	Method m checks or casts an object to class c.
(implements, c, i)	coarse	A class in file c is an implementation of an interface in file i.
(extends, c_1, c_2)	coarse	A class in file c_1 is a subclass of a class in file c_2.
(declaresMethodReturn-OrParamTypeAs, c_1, c_2)	coarse	A class in file c_1 declares a method such that the object returned or passed as a parameter can be of a class in file c_2.
(declaresFieldAs, c_1, c_2)	coarse	A class in file c_1 declares a field of a class in file c_2, or ancestor of c_2 in the inheritance hierarchy.
(samePackage, c_1, c_2)	coarse	A class in file c_1 is in the same package as a class in file c_2.

Table 3.1: Structural relationships used in our criteria for Java source code

Table 3.2 presents a similar table as Table 3.1, but the structural relationships considered are on C++ source code rather than on Java. For C++, we also consider relationships header and implementation files. We do not consider the "implements" relationship as it is not applicable in C++. Moreover, the "samePackage" relationship is changed to "sameDirectory" relationship. C++ does not provide the notion of packages to group classes. As a convention, C++ programmers use directories to group classes.

Non-structural information refers to relationships between two entities in the source code that are not supported by the programming language. Non-structural information includes information in comments, naming conventions, string literals, data sharing (in which there may not be a shared type), and reflection (e.g., invoking of a method on an object even if the method is not known until runtime,

Structural relationship	Granularity	Description
(reads, m, f)	fine	Method m accesses the value of field f.
(writes, m, f)	fine	Method m writes a value to field f.
(calls, m_1, m_2)	fine	Method m_1 contains a method invocation that can bind to m_2.
(creates, m, c)	fine	Method m creates an object of class c.
(checks, m, c)	fine	Method m checks or casts an object to class c.
(headerOf, h, c)	coarse	File h contains the declaration of a class in file c.
(extends, c_1, c_2)	coarse	A class in file c_1 is a subclass of a class in file c_2.
(declaresMethodReturn-OrParamTypeAs, c_1, c_2)	coarse	A class in file c_1 declares a method such that the object returned or passed as a parameter can be of a class in file c_2.
(declaresFieldAs, c_1, c_2)	coarse	A class in file c_1 declares a field of class c_2, or ancestor of a class in file c_2 in the inheritance hierarchy.
(sameDirectory, c_1, c_2)	coarse	A class in file c_1 is stored in the same directory as a class in file c_2.

Table 3.2: Structural relationships used in our criteria for C++ source code

or getting information about a class's modifiers, fields, methods, constructors, and superclasses).

The interestingness value of a recommendation, f_r where $f_s \rightarrow f_R$ and $f_r \in f_R$, is based on how likely it is that a developer pursuing and analyzing f_s would consider the file f_r as part of the solution of a modification task. We assume that such a developer has access to simple search tools (e.g. `grep`) and basic static analysis tools that provide backward references given a point in the source code (e.g., callees of a given method or methods that write a given field).

We categorize a recommendation $f_s \rightarrow f_r$ as *obvious* when

- a method *that was changed* in f_s has a direct fine-grained reference—reads, writes, calls, creates, checks, declares, as described in Table 3.1 for Java and Table 3.2 for C++— to a method, field or class in f_r, *or*

- a class *that was changed* in f_s has a strong coarse-grained relationship—the coarse-grained relationships described in Table 3.1 for Java and Table 3.2 for C++—as a class in f_r.

We categorize a recommendation as *surprising* when

- f_s has no direct structural relationships with f_r, *or*
- a fragment in f_s contains non-structural information about f_r.

A recommendation is *neutral* when

- a method in f_s, *other than the one that was changed* in f_s, has a direct fine-grained reference to a method, field, or class in f_r, *or*
- a class *that was changed* in f_s has a weak coarse-grained relationship—it indirectly inherits from, is in the same package or directory that has more than 20 files—with a class that was changed in f_r.

If f_s and f_r have more than one relationship, the interestingness value of the recommendation is determined by the interestingness value of the most obvious relationship.

3.2 Evaluation settings

In this section, we first describe the parameters for the data mining algorithms and statistics about the patterns generated from data mining. We then present statistics about file revision data. Finally, we describe the process of retrieving information about modification tasks from Bugzilla and statistics about the tasks.

3.2.1 Patterns

Table 3.3 describes the parameters we use in the data mining algorithms. The first column "Algorithm and parameter" lists the parameters we use for the two algorithms. In the first column, "FP" refers to the frequent pattern mining algorithm, with the support threshold indicated after a comma, and "Corr" refers to the correlated set mining algorithm, with the minimum expected value indicated after a comma. The second column "Target system" indicates whether the data mining algorithm is applied to Eclipse and Mozilla. The "Number of files" column presents the number of files that are generated from the patterns extracted using the algorithm with the specified parameter applied to one of Eclipse and Mozilla. For the frequent pattern algorithm, the value of the support threshold *min_support* was varied so that a reasonably large number of files (over 200) were involved in patterns and the support was not too low (not below 5). We were also careful to choose thresholds that were neither too restrictive nor too relaxed. An overly restrictive threshold results in too few patterns. This situation affects the recall value as the recommendations do not cover the changes needed for a modification task. An overly relaxed threshold affects the precision since too many patterns result in a number of recommendations, only a few of which are correct. For the correlated set mining algorithm, only the minimum expected value was varied because varying other parameters did not affect the results.

Comparing the patterns generated for Eclipse and Mozilla using the frequent pattern algorithm with the same parameter setting (*min_support* equals 20 and 15), Mozilla has more than five times more files involved in the change patterns than Eclipse.

For the correlated set mining algorithm, the minimum expected value thresh-

Algorithm and parameter	Target system	Number of files
FP, *min_support* = 20	Eclipse	134
FP, *min_support* = 15	Eclipse	319
FP, *min_support* = 10	Eclipse	877
FP, *min_support* = 20	Eclipse	3462
FP, *min_support* = 30	Mozilla	598
FP, *min_support* = 25	Mozilla	807
FP, *min_support* = 20	Mozilla	1139
FP, *min_support* = 15	Mozilla	1715
Corr, *min_expected_value* = 1.0	Eclipse	16
Corr, *min_expected_value* = 1.0	Mozilla	227

Table 3.3: Statistics from patterns formed in the training data of Eclipse and Mozilla

old of 1.0 yielded results that involve only 16 files in Eclipse. This result is significant because the threshold is the suggested minimum for the chi-squared test to be applicable. Since so few recommendations could be generated, we decided not to further analyze the correlated set mining because the algorithm is not applicable to the Eclipse data. We also decided not to analyze the correlated set mining algorithm on Mozilla because we could not compare the results to Eclipse.

Pattern statistics

Tables 3.4 and 3.5 show the number of patterns and the number of files covered by patterns of different cardinality. The column name "FP 20" denotes the frequent pattern algorithm with support threshold equal to 20. For each parameter setting on each system, the number of files involved in patterns of a particular cardinality c is much smaller than the product of the cardinality c and the number of patterns, indicating that the patterns have many overlapping files. For example, the number of files covered by patterns of cardinality two for each parameter setting is similar to, but not twice as many as, the number of associated patterns.

	FP 20	FP 15	Fp 10	Fp 05
2	113 (123 files)	265 (269 files)	790 (690 files)	2193 (2013 files)
3	13 (23 files)	46 (71 files)	270 (284 files)	1612 (1446 files)
4	0	7 (27 files)	59 (127 files)	924 (953 files)
5	0	2 (6 files)	11 (39 files)	482 (745 files)
> 5	0	0	5	455
all patterns	126 (134 files)	320 (319 files)	1135 (877 files)	5666 (3462 files)

Table 3.4: Statistics on patterns in Eclipse

pattern size	FP 30	FP 25	Fp 20	Fp 15
2	430 (477 files)	582 (653 files)	876 (889 files)	1293 (1295 files)
3	163 (209 files)	251(288 files)	396 (457 files)	818 (699 files)
4	51 (90 files)	131 (161 files)	213 (238 files)	381 (438 files)
5	26 (52 files)	37 (61 files)	135 (141 files)	251 (264 files)
> 5	20	25	80	332
all patterns	690 (598 files)	1024 (807 files)	1700 (1139 files)	3075 (1715 files)

Table 3.5: Statistics on patterns in Mozilla

3.2.2 Source code revision data

Figure 3.2 presents some metrics about the Eclipse and Mozilla developments and outlines the portions of the development history we considered in our analysis. In both systems, the training data comprised changes on over 20,000 files and over 100,000 revisions to those source files.

Transaction statistics

Table 3.6 shows the number of transactions of different cardinality as well as the total number of transactions. For the period of time we considered as training data, both Eclipse and Mozilla have similar number of transactions. In both systems, transactions of two items have the most counts and the number of transactions of decreases as the cardinality of the transaction increases.

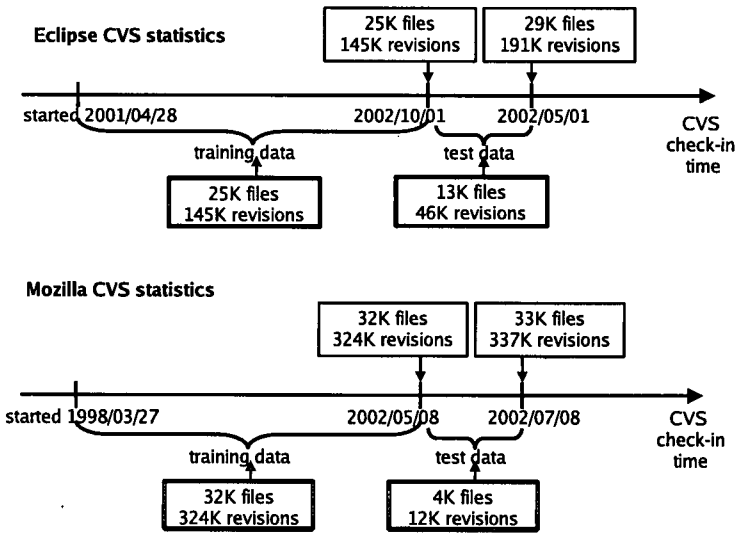

Figure 3.2: Statistics on the Eclipse and Mozilla CVS repositories

cardinality of transactions	Mozilla	Eclipse
2	13,734	13,608
3	5,606	6,053
4	3,605	3,822
5	2,046	2,326
> 5	7,657	9,228
total	32,648	35,037

Table 3.6: Transaction statistics of Eclipse and Mozilla

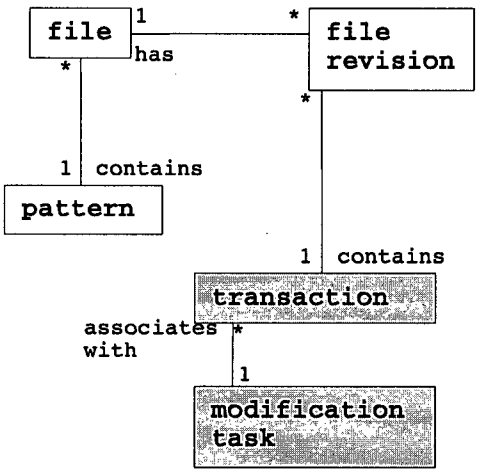

Figure 3.3: Data linkage schema in the mining process

3.2.3 Bugzilla data

Similar to data related to the data mining process, data related to modification tasks for validation is stored in the database described in Section 2.4. Figure 3.3 shows the schema related to the modification task related data—`transaction` and `modification task` tables—highlighted. `modification task` represents modification tasks that happen during period identified as test data.

Populating data in the `modification task` table involves retrieving the data from the Bugzilla HTTP server[1]. This process consists of the following three steps. First, we need to get the identifiers of modification tasks that were fixed within the test period. Second, we need to query Bugzilla for each modification task from the query result from the first step. Third, we need to retrieve information about when the status was changed to "fixed" as well as who changed the status.

After populating the database with information retrieved from Bugzilla just

[1]URL at`http://bugzilla.mozilla.org/`

support threshold	# of modification tasks
5	125
10	105
15	73
20	49

Table 3.7: Statistics about modification tasks in Eclipse

described, we establish an `associates-with` link between each modification task (in the `modification task` table) and files in the solution (in the `transaction` table). This is performed in two steps, each step using one heuristic. We first establish the linkage by applying a heuristic that relies on developers recording the identifier of a modification task upon which they are working as part of the CVS check-in comments for a solution of the modification task. We then apply the second heuristics on the data that are not linked in the first step. The second heuristic relies on a common practice that a developer commits the files corresponding to a solution of the modification task into CVS close to the time at which they change the status of the modification task report to "fixed" [6].

Tables 3.7 and 3.8 show the number of modification tasks whose solutions were checked in during the period of time we identified as training data and whose solutions contained at least one file from a change pattern with different support thresholds. Although Eclipse has three times more modification tasks with "fixed" status (6,053, of which 2702 associate with at least one transaction) than modification tasks in Mozilla (2,212, of which 1268 associates with at least one transaction) in the training data, Eclipse has a fewer number of modification tasks that have at least one file involved in a change pattern for the support threshold we investigated.

support threshold	# of modification tasks
15	487
20	428
25	375
30	337

Table 3.8: Statistics about modification tasks in Mozilla

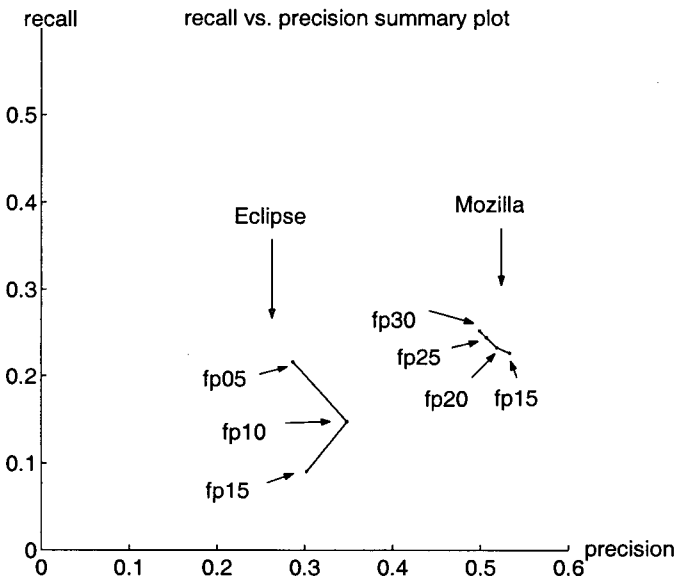

Figure 3.4: Recall versus precision plot showing two algorithms for each of the two target systems Eclipse and Mozilla.

3.3 Results

We analyzed the recommendations on modification tasks whose solutions are from the test data, using the predictability and interestingness criteria described in Section 3.1.

3.3.1 Predictability results

Figure 3.4 shows the precision and recall values that result from applying the frequent pattern algorithm on each system. The lines connecting the data points on the recall versus precision plot show the trade-off between precision and recall as the parameter values are altered. The recall and precision axes are adjusted to only show values up to 0.6 instead of 1.0.

Each data point presents the average precision and recall. To compute the averages, we only include the precision and recall for recommendations from modification tasks M in the test data where each task's solution contains at least one file from a change pattern. We use each file f_s in the solution of a modification task $m \in M$ to generate a set of recommended files f_R and calculate f_R's precision and recall described in 3.1.1. The average precision is the mean of such precision values and analogously for the average recall.

The recall and precision values for the generated change patterns for Mozilla are encouraging; precision is around 0.5 with recall between 0.2 and 0.3 (meaning that, on average, around 50% of the recommended files are in a solution and 20% to 30% of the files in a solution are recommended). The line plot shows a steady trade-off between recall and precision when *min_support* changes. However, the recall and precision values in the Eclipse case are less satisfactory; precision is only around 0.3, recall is around 0.1 to 0.2, and the line plot shows some strange behaviour, namely when *min_support* threshold equals 15 there is a sudden decrease in both precision and recall. The small number of patterns and small number of files covered by patterns may cause this behaviour because so few recommendations can be made.

Although the recall and precision may seem low, the value of the approach must be evaluated based on its ability to provide helpful recommendations; the

scenarios presented in Section 1.4 provide a few such examples. We assess the interestingness of recommendations further below. We argue that recommendations with precision presented in Figure 3.4 are useful as long as the gain to the developer when the recommendation is helpful is greater than the cost to the developer of determining which recommendations are false positives. Our approach can supplement existing approaches that provide recommendations to relevant source code in a modification task.

3.3.2 Interestingness results

To assess the interestingness of the recommendations, we randomly chose 20 modification tasks from each project for the period of time covered by the test data shown in Figure 3.2. For each file (f_s) associated with the check-in for a modification task that was also contained in at least one change pattern, we determined which other files (f_R) would be recommended. For each recommendation, we determined its interestingness level according to the criteria described in Section 3.1.2.

Tables 3.9 and 3.10 summarize the recommendations resulting from change patterns generated with the frequent pattern algorithm where *min_support* threshold equals 20 in Mozilla and *min_support* equals 10 in Eclipse. We chose to investigate the frequent pattern algorithm with these threshold settings because they gave the best results in terms of precision and recall. We categorize recommendations using the interestingness criteria described in Section 3.1.2. We grouped the recommendations with the same interestingness level, as indicated in the "Interestingness" column, into similar cases. We gave each case a descriptive name, as shown in the "Description" column. Recommendations with the same descriptive name and with the same interestingness level can come from different modification tasks; the iden-

tifiers of those tasks are given in last column. The number in parentheses beside each identifier is the number of recommended files.

Mozilla interestingness results

Table 3.9 presents a categorization of recommendations for Mozilla. We were able to generate recommendations for 15 of the 20 selected modification tasks. For five of the tasks (#82534,#126213,#135027,#137246, and #140104), none of the files involved in the solution appeared in any change pattern; therefore, these five tasks are not listed in Table 3.9. The 15 tasks that resulted in recommendations include two cases involving *surprising* recommendations, two cases involving *neutral* recommendations, and five cases involving *obvious* recommendations. We focus the discussion on the cases categorized as *surprising* and *neutral.* Recommendations classified as *obvious* may still be of value to a developer, as shown by the scenario described in Section 1.4.3.

The "cross-language" case in the *surprising* category demonstrates how our approach can reveal interesting dependencies on files written in different languages, and on non-code artifacts, that may not be easily found by a developer.

- For Mozilla, a developer specifies the layout of widgets in XUL (XML-based User interface Language), which eases the specification of the UI and provides a common interface for the UI on different platforms. XUL does not solely define the UI; a developer must still provide supporting code in a variety of formats, including XML schema files and Javascript files. This situation occurred in the solution of modification task #150099, which concerned hiding the tab bar by default. The solution involved adding a new menu item for storing the user preference of showing or hiding the tab bar in a XUL file, declaring the

Interestingness	Description	Modification task ids (and number of recommendations)
surprising	cross-language	150099 (6)
surprising	duplicate code bases	92106 (2), 143094 (2), 145560 (2), 145815 (2), 150339 (2)
neutral	distant instance creation and call dependence	123068 (2)
neutral	distant inheritance	74091 (2)
obvious	header-implementation	92106 (2), 99627 (2), 104603 (2), 135267 (8), 144884 (2), 150735 (2), 74091 (2), 123068 (2)
obvious	interface-implementation	135267 (2)
obvious	direct inheritance	74091 (12)
obvious	direct call dependence	99627 (2)
obvious	same package with less than 20 files	135267 (6)

Table 3.9: Mozilla recommendation categorization by interestingness value

menu item in a XML schema file, and initializing the default settings of the menu item as call-back code in a Javascript file. Our approach generated six *surprising* recommendations involving Javascript-XUL, XML schema-XML, and XML schema-Javascript.

The "duplicate code bases" case from the *surprising* category demonstrates how our approach can reveal potentially subtle dependencies between evolving copies of a code base.

- As part of the solution of modification task #92106, two scripts that built different applications in the XML content model module `TransforMiix` needed to be updated. One script was for building an application for transforming XML documents to different data formats, and the other script was for building a benchmarking application for the former application. Much of the two build scripts shared the same text, and changes to one script usually required similar

changes to the other script. In fact, this code duplication problem was later addressed and eliminated (modification task #157142). When either of these build scripts was considered to be modified, our approach was able to provide a recommendation that the developer should consider the other script, resulting in two *surprising* recommendations.

- For the Mac OS X operating system, Mozilla includes two code bases, `Chimera` and `Camino`. The second is a renamed copy of the first code base, but changes were still occurring in parallel to each code base. For modification tasks #143094 and #145815, our approach was able to recommend that a change to a file in one code base should result in a change to the corresponding file in the other code base.

- Modification task #145560 concerned about fixing a typo in a variable named `USE_NSPR_THREADS` in the configuration template file `configure.in`. The shell script `configure`—the other file involved in the solution of the modification task—was generated using `autoconfig` for configuring different platforms using appropriate parameter values specified in the template file `configure.in`. Our approach was able to recommend that a change in file `configure.in` required a change in `configure` because `configure` must be regenerated from `configure.in`.

- Modification task #150339, which was described in Section 1.4.1, presented a situation where a developer missed changing code which was duplicated in two layers of the system `gtk` and `xlib`. Our approach generated recommendations suggesting parallel changes in the code.

The "distant instance creation and call dependence" case in the *neutral* cat-

egory describes dependence between two files, in which a method in one file, other than the one that was changed, creates an instance and invokes a call on the instance of a class in the other file. Such dependence may not be apparent to a developer since the dependence is not in close to the place (not within the same method) that was changed in the solution.

- Modification task #123068 involved adding the host-name of a SMTP server in the error message resulted from failed attempts to connect to the server, as described in the scenario in Section 1.4.2. Two files involved in the solution were `nsMsgSend.cpp` and `nsSmtpService.cpp`. The class `nsMsgComposeAndSend` in the file `nsMsgSend.cpp` represented the operation for composing and sending an email or a newsgroup message, and `nsSmtpService.cpp` contained source code that deals with the network protocal. The relationships between the `nsMsgComposeAndSend` class and the `nsSmtpService.cpp` were that `nsMsgComposeAndSend` created an object of class `nsSmtpService` and invoked a method, not the one that was changed, on the `nsSmtpService` object for sending the message. Our approach was able to recommend one of the files if a developer considered to change the other file.

The "distant inheritance" case in the *neutral* category refers to recommendations that involve an inheritance relationship in which there is some common ancestor in the inheritance hierarchy.

- Modification task #74091 involved a bug in the formatting of the text area when the back button is pressed. Two files involved in the solution, `nsGenericHTMLElement.cpp` and `nsHTMLTextAreaElement.cpp`, were in the same inheritance hierarchy: `nsHTMLTextAreaElement` was a third-

Interestingness	Description	Modification task ids (and number of recommendations)
surprising	cross-platform/XML	24635 (230)
neutral	distant call dependence	21330 (2),24567 (2)
neutral	distant inheritance	25041 (12)
obvious	containment	13907 (2), 23096 (2), 24668 (2)
obvious	framework	21330 (2)
obvious	same package with less than 20 files	21330 (2)
obvious	instance creation/being created	23587 (4)
obvious	method call dependence	21330 (2), 23587 (4), 24657 (2), 25041 (2)
obvious	direct inheritance	25041 (8)
obvious	interface-implementation	24730 (2)

Table 3.10: Eclipse recommendation categorization by interestingness value

level subclass of `nsGenericHTMLElement`. Our approach was able to recommend this dependence when either of the files was changed even though `nsGenericHTMLElement` was the superclass of many other classes.

Eclipse interestingness results

Table 3.10 shows the categorization of recommendations for Eclipse. Fewer of the selected modification tasks for Eclipse resulted in recommendations than for Mozilla. We could not provide recommendations for 11 of the modification tasks (#16817,# 24165, #24406, #24424, #24449, #24594,#24622, #24756, #24828, #25124, and # 25133) because the changed files were not covered by a change pattern. Of the nine tasks that resulted in recommendations, eight of them had solutions involving files in which the classes were structurally dependent. We group these into seven cases involving *obvious* recommendations, two cases involving *neutral* recommendations, and one case involving *surprising* recommendations.

The "cross-platform/XML" case involved recommendations for non-code artifacts, which as we argued above, may not be readily apparent to a developer and may need to be determined by appropriate searches.

- Modification task #24635 involved a missing URL attribute in an XML file that describes which components belong to each platform version. The change involved 29 files that spanned different plug-ins for different platform versions. Our approach generated 230 *surprising* recommendations. This large number of recommendations can be made because the solution for the problem contains 29 files of which 18 match at least one change pattern. Each of these 18 files, when used as the starting file, typically generated recommendations of over 10 files, summing to 230 recommendations.

The "distant call dependence" case in the *neutral* category involved recommendations in which two files have call dependences, but they are not in the methods changed in the solution. In the two modification tasks involved in the case, the call dependences are caused by the larger context of the framework introduces dependencies between them. Such dependencies may be obvious to experienced framework users, but may be more difficult to discern by newcomers.

- Modification task #21330 involved adding a wizard page for renaming a folder in the file exploring view. The solution of the task involved two files with a call dependence because of polymorphism: `InstallWizard` and a class that provides UI functionality `TargetPage` for a wizard page. Our approach was able to recommend to a developer that when changing `TargetPage`, they should also consider the file containing the `InstallWizards` class.

- Modification task #24657 described an error in which Eclipse assumed a par-

tially downloaded feature—a JAR file containing a smallest separately downloadable and installable functionality of Eclipse—as fully downloaded. Two classes involved in the solution of the task were `FeatureContentProvider` and `JarContentReferece`. `FeatureContentProvider` is an abstraction of the internal packaging structure of a feature and manages a mapping between an abstraction for an entry—represented by the class `ContentReference`—in the packaging structure and the reference of an actual file in the file system. The other file involved in the solution `JarContentReference`—a subclass of `ContentReference`—represents a entry in a JAR file that is in the internal packaging structure. The methods that were changed in each of the files `JarContentReference.java` and `FeatureContentProvider.java` did not have direct fine-grained structural reference to methods in the other file, although each of the files have parts that have call relationships to the other file, resulting in two *neutral* recommendations.

The "distant inheritance" case in the *neutral* category involved dependencies between classes that have some common ancestor in an inheritance hierarchy. In some cases, the number of classes involved in the inheritance hierarchy is large and this may complicate the implementation of a solution.

- Modification task #25041 involved changing the classes that represent an abstract syntax tree of a Java program. Nodes in the tree are involved in complicated inheritance hierarchies. The solution of the modification consisted of 33 files, 12 of which were involved in changed patterns. 11 of he 12 files involved in changed patterns were in same the same ancestor `ICompilationUnit`. Although an inheritance relationship existed between the two files, it is not

always easy to uncover the importance of the dependence since there were many other related dependencies.

Conclusion of the interestingness results

We have evaluated the interestingness of recommendations by presenting the relationships between files covered by recommendations produced by change patterns and in the solution of the selected modification tasks in Tables 3.9 and 3.10, as well as analyzing the relationships in Section 3.3.2. We observed that the majority of the relationships found by our approach are structural (relationships in the *neutral* and *obvious* categories) in both Eclipse and Mozilla. This is not surprising because the majority of these systems are implemented using programming language constructs, Java for Eclipse and C++ for Mozilla; therefore, solutions to modifications to these systems involve files that are structurally related. In addition to structural relationships, our approach did reveal valuable non-structural relationships not captured by programming language constructs, as analyzed in Section 3.3.2.

3.4 Performance

The queries that we performed in this validation took a few seconds on a Sun Ultra 60 system with 1280 MB RAM with 2 x 360 MHz UltraSPARC-II processors. The computation time of populating the database is more time-consuming, but is performed less often. Populating the database with file revision data faor Mozilla took over two hours. Transaction computation took 6 minutes on Eclipse and 11 minutes on Mozilla. The computation time of mining change patterns increases support threshold decreases: tables 3.11 and 3.12 show that the performance ranges from 1 minute to 55 minutes for Eclipse and from 1 minute to 3 minutes for Mozilla.

support threshold	time (s)
20	44
15	86
10	439
05	3302

Table 3.11: Statistics on pattern computation on Eclipse

support threshold	time (s)
30	52
25	65
20	102
15	209

Table 3.12: Statistics on pattern computation on Mozilla

3.5 Summary of the evaluation

The evaluation presented in this chapter attempted to validate our hypothesis that recommendations provided by change patterns can help a developer identify relevant source code in a modification task. The predicatbility analysis on precision and recall in Section 3.3.1 provided some quantitative measures on our approach. Although the precision and recall were not high, we have argued that our approach is useful, especially in the cases described in Section 3.3.2 where recommendations cannot be obtained easily from existing analyses.

Chapter 4

Discussion

This chapter presents some issues about the evaluation and the appraoch. Section 4.1 discusses issues about the predictability criteria and rationales on how we determine our interestingness criteria. Section 4.2 provides observations about the results. Section 4.3 presents issues with generalizing the results obtained from Eclipse and Mozilla to other projects. The remaining two sections provide a discussion on the approach, including the granularity of the source code used in the change pattern mining process (Section 4.4) and the applicability of correlated mining algorithm (Section 4.5).

4.1 Evaluation criteria

4.1.1 Predictability

Our evaluation of recall and precision is conservative in the sense that we measure these values with respect to whether a recommended file f_r was part of the set of files f_{sol} that was checked in as part of the solution. We cannot determine if a recommended file that did not have a revision stored as part of the change might

have been helpful to a developer in understanding the source to make the desired changes. In addition, the recommendation is given using one starting file, which represents the minimum amount of knowledge a developer would need to apply our approach.

4.1.2 Interestingness

In the interestingness criteria described in Section 3.1.2, the determination of what relationships between two fragments of source code we considered and what interestingness values we assigned were subjective. We argue that our criteria is reasonable. We did not choose to consider transitive structural relationships because the number of relationships given by transitive structural relationships can be huge and therefore are not immediately useful to a developers. In addition, we purposely chose to categorize the relationships to only three levels of interestingness (*obvious*, *neutral*, and *surprising*), rather than significantly more levels because of the uncertainties caused by the subjectivity in the categorization. In the future, we can reduce such uncertainties by performing user studies on how developers categorize a relationship.

4.2 Evaluation results

When applied to the two open source systems, our approach resulted in recommendations with higher precision on Mozilla than on Eclipse. Moreover, although the size of the training data was similar between the two projects (as measured by the number of files that were changed shown in Figure 3.2), we had to set a lower support threshold in Eclipse than in Mozilla. One possibility for these differences is in the size of the development histories of the two projects. Mozilla has twice as long the development history as Eclipse has. The change patterns constructed from

Eclipse may be less reliable than the change patterns from Mozilla, as indicated by the lower threshold required in the evaluation on Eclipse.

In both projects, the recall and precision seem low. There are several possibilities. One possibility is that the number of transactions resulted in the projects are too few, as association rule mining assumes a large number of transactions. For example, in the first frequent pattern algorithm literature, the number of transactions used in the experiment is more than 20 times greater than the number of items in a validation of the frequent pattern algorithm [8]. However, in Eclipse and Mozilla, the number of transactions and the number of items—files in our context—are approximately the same because items do not occur in as many transactions as in other applications. This may be one reason that the recall and precision are not high.

The use of CVS by these projects further impacts our approach since historical information is lost by CVS when a file or directory is renamed. Moreover, significant rewrites and refactorings of the code base can affect our approach. Such changes affect the patterns we compute because we do not track the similarity of code across such changes.

4.3 External validity

To increase the likelihood that the approach also applies to other projects, we chose two inherently different projects. The two projects are written in different programming languages: Eclipse is mainly written in Java and Mozilla is mainly written in C++. The two projects also differ in development history: Mozilla has more than six years of development history whereas Eclipse only has three years of development history. In addition, we chose to analyze Eclipse and Mozilla because these

projects involve a lot of developers, reducing the likelihood that peculiar programming practice of a particular programmer dramatically affecting the results.

The two projects have particular properties that may not be generalizable to other projects. One such property is that both Eclipse and Mozilla are open source projects and use CVS and Bugzilla to track modifications. Programming practices of developers that are specific to these practices can affect the generalizability of the results.

4.4 Granularity

Currently, the change associations we find are among files. Applying our approach in the method granularity—where change patterns describe *methods* instead of *files* that change together repeatedly—may provide better results because a smaller unit of a source code may suggest a more similar intention in changing the code. However, refining the granularity weakens the associations (each pattern would have lower support), which may not be well-handled by our current approach.

4.5 Applicability of correlated mining algorithm

When we applied the correlated set algorithm (of computing correlated sets of cardinality two) to the revision data of the target systems, the fact that so few change patterns were generated indicated that that the algorithm is not applicable on the data. This is contrary to what we expected because correlated set mining considers more information than frequent pattern mining algorithm: correlated set mining considers four correlations of combinations of two files changed or not changed together, instead of only associating two files changed together in frequent patterns.

We observed that the *expected frequencies*[1] are much smaller than the *observed frequencies*[2] because the total number of possible files in the system is much larger than the number of times any two files changed together. Data with such distribution does not work well with the chi-squared test used in correlated set mining, which requires expected frequencies not to be too small. One way to improve this situation is to partition the transactions in a meaningful way (so that it would not dramatically weaken the correlations) and apply the correlated set mining algorithm to each partition, so that the number of files in the system is closer to the number of times any two files changed together. Dividing the system into architectural modules is one such meaningful way to partition of the system.

[1] As described in Section 2.2.2, *expected frequencies* are frequency counts of transactions based on the assumption that two items are independent of each other given by the product of marginal probabilities of a transaction containing or not containing *individual* items.

[2] *Observed frequencies* are the number of transactions that contain or not contain both, neither, or either of the items involved in the correlation.

Chapter 5

Related Work

We focus the discussion of related work in three areas. Section 5.1 describes approaches that use development history to determine the "scope"—or what a developer should consider—for a modification task. Such approaches vary in their granularity and types of the generated results, as well as their specificity to a task. Section 5.2 presents analyses that also provide the scope of modification tasks but use program analysis techniques such as static and dynamic analyses. Finally, section 5.3 presents two recommendation systems that can suggest reusable code to a developer performing a modification task.

5.1 Determining the scope of modification tasks from development histories

Zeller and colleagues, independently from us, developed an approach that also uses association rule mining on CVS data for recommending source code that is potentially relevant to a given fragment of source code [23]. The rules determined by their approach can describe change associations between files or methods.

Their approach differs from ours in that they use a particular form of association rule mining, in which rules determined must satisfy some support and confidence. Frequent pattern mining, the algorithm that we use, uses only support to determine the association. The reason that we did not choose to use association mining is because confidence gives misleading association rules in some cases [5]. It was also part of our motivation for considering correlation rule mining, which takes into account how often both files are changing together as well as separately.

As for the validation, both pieces of research work consist of a quantatitive analysis on the predicatability of recommendations with similar results. For the qualitative analysis, they presented some change associations that were generated from their approach, whereas we analyzed the recommendations provided in the context of completed modification tasks. In addition, we evaluated the quality of the correct recommendations provided by the change associations based on the usefulness of recommendations to a programmer, whereas they did not evaluate this aspect.

Zeller and colleagues also apply association rule mining to a different problem: determining evolutionary dependencies amongst program entities for the purpose of justifying a system's architecture [22]. This involves determining the degree of modularity of a system based on analyzing the density of evolutionary dependencies between entities in the source as well as the proportion of inter- versus intra-entity evolutionary coupling.

Lethbridge and colleagues address a similar question to that addressed in this thesis: When a programmer is looking at a piece of code, they want to determine which other files or routines are relevant [9]. They proposed an approach that predict the relevance of any pair of files based on whether the files have been looked at or

changed together. Information about pairs of relevant files are used to learn concepts by building decision trees on attributes, such as the length of the common filename prefix and the number of shared routines (determining the number of shared routines requires simple static analysis). Similar to our approach, their approach can apply across languages and platforms if the attributes do not depend on programming constructs. Our approach differs in that we find files that change together *repeatedly* instead of only changing more than once. Their results show that the error rate of classifying a file, given another file, to one of the three levels of relevance is 27% on average, which is better than the error rate of 67% when the relevance value is assigned randomly. In contrast, our notion of recall and precision is based on whether the recommendations are correct with respect to a given modification, and the error rate of giving recommendations randomly is extremely high because of the huge number of files (over 20,000 in each of Eclipse and Mozilla) that can possibly be recommended.

Hipikat is a tool that provides recommendations about project information a developer should consider during a modification task [6]. Hipikat draws its recommended information from a number of different sources, including the source code versions, modification task reports, newsgroup messages, email messages, and documentations. In contrast to our approach, Hipikat uses a broader set of information sources. This broad base allows Hipikat to be used in several different contexts for recommending a different artifacts for a change task. However, for a change task, Hipikat can only recommend modifications, and their associated file revisions, similar to the modification at hand. Our approach is complementary to Hipikat, as it does not rely upon a similar modification task having occurred in the past.

5.2 Impact analysis

Impact analysis approaches (e.g., [4]) attempt to determine, given a point in the code base involved in a modification task, all other points in the code base that are transitively dependent upon the seed point. This information could help a developer determine what parts of the code base are involved in the modification task. Many of these approaches are based on static slicing (e.g., [7]) and dynamic slicing (e.g., [1]).

Static slicing identifies all the statements in a program that might affect the value of a variable at a given point in the program. This approach relies on static analysis on data-flow and control-flow dependence. This information could help a developer determine what parts of the code base are involved in the modification task [7]. In contrast to these approaches, our data mining approach can work over code written in multiple languages and platform, and scales to use on large systems. On the other hand, slicing approaches can provide fine-grained information about data-flow and control-flow dependencies even should the parts of the code base involved in the change not have changed frequently in the past. We see our approach as complementary to static slicing approaches.

Dynamic slicing finds all parts of source code that affect a variable in an execution for some given input of the program, rather than for all inputs as in static slicing. As with other approaches that use program executions, dynamic slicing relies on an executable program and the availability of appropriate inputs of the program. On the other hand, our approach can work with code that is non-executable, or with code that consists of components running on different platforms. Similar to static slicing, dynamic slicing can provide finer-grained information about code related to a modification task, without relying on the code being changed repeatedly in the

past. Our approach is complementary to dynamic slicing approaches.

5.3 Recommendations on reusable code

Association mining has been used for suggesting reusable code. Michail and colleagues used such an approach to find library reuse patterns to aid a developer in building applications with a particular library [12, 13]. The extracted patterns summarize usage information, such as explicating that application classes that inherit from a particular library class often override certain member functions. Our approach differs in both intent and form. In addition to trying to solve a different problem, we are applying the data mining to versioning data, rather than to a single version as in Michail's work.

CodeBroker is a recommendation system that suggests source components in a library that are potentially reusable in the current programming task [21]. To generate a recommendation, the tool uses comments entered in the editor to match library code that contains similar comments using information retrieval methods. The tool can also use refine the matches of the relevant library code by using constraints derived from type information in the source code. In constrast to CodeBroker, our approach does not require proper comments documentation in the library code. However, generating change patterns for recommendations using our approach relies on the availability of a sizable source code repository of the library, which may only be accessible within the development team, not a general user, of the library.

Chapter 6

Conclusion

In this thesis, we have described our approach of mining revision history to help a developer identify pertinent source code for a change task at hand. We have demonstrated that our approach can provide useful recommendations to developers on Eclipse and Mozilla, based on the predictability and interestingness criteria. Although the precision and recall are not high, recommendations reveal valuable dependencies that may not be apparent from other existing analyses.

We have demonstrated our thesis through the development of an approach that helps developers identify relevant source code to a modification task and the demonstration of the validity of the approach. In addition to this major claim, we have developed the interestingness criteria for assessing the interestingness of the recommendations in the validation of the our approach. The interestingness criteria describes the usefulness of a recommendation $f_s \rightarrow f_r$ to a developer based on whether the relationships between f_s and f_r are obvious to the developer. This criteria can be used in qualitative studies of other source code recommendation systems.

Finding change patterns from revision history is a fertile research area. In

our current approach, we only consider a boolean value of whether a file has changed. Taking into account what kind of change occurred may enrich the results. We would also like to perform research on applying static analysis across multiple versions, as many modification tasks involve solution with structural relationships, as seen in Section 3.3.2. Another direction is to address the lack of data problem by using other algorithms, such as concept analysis.

Appendix A

An example of FP-tree based mining algorithm

To demonstrate the FP-tree based mining algorithm, we present an example, which is the example presented in the original literature [8]. The figures are adopted from the corresponding presentation slides[1].

Figure A.1 illustrates transactions in the database and the initial FP-tree generated. In this example, the database contains five transactions (Figure A.1, label 1). The minimum support threshold *min_support* is 0.5, so any frequent pattern must have support greater or equal to 3. The algorithm extracts frequent items—items with support greater or equal to 3—from the transactions, sorts the frequent items by descending order, and stores the sorted frequent items in an auxiliary data structure called the Header Table (Figure A.1, label 2). The five transactions are filtered so that only the frequent items remain and the frequent items are ordered in descreasing support (Figure A.1, label 2). We call these filtered transactions D_f.

[1]The presentation slides are written by Han and can be downloaded from ftp://ftp.fas.sfu.ca/pub/cs/han/slides/almaden00.ppt.

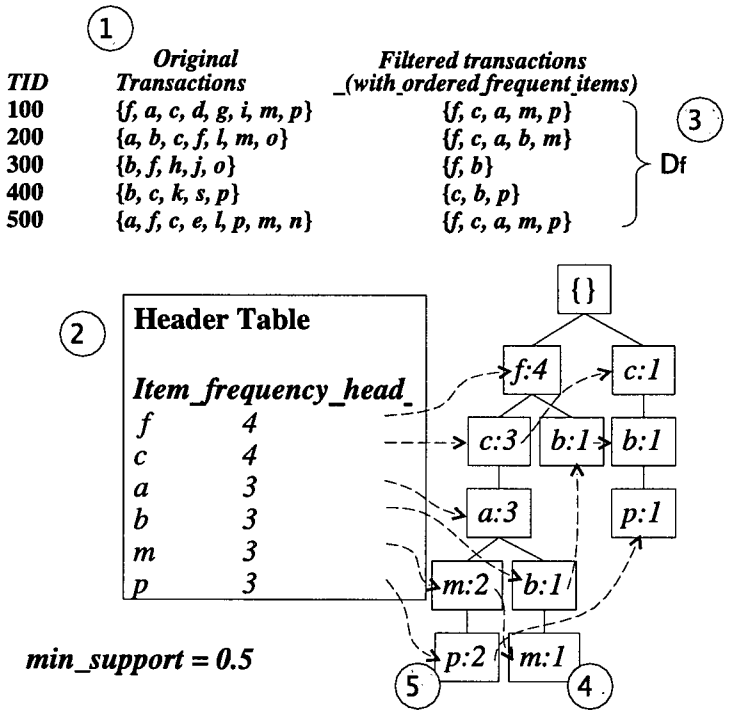

Figure A.1: Initial database and associated FP-tree

The algorithm constructs an FP-tree (Figure A.1, label 4) by inserting each of the filtered transactions in D_f. A node in a FP-tree represents a frequent item and stores its name and support. Each filtered transaction corresponds to a path from the root to a leaf node. For example, the path from the root node to the leaf node $m:1$ (Figure A.1, label 4) corresponds to the filtered transaction $\{f, c, a, b, m\}$, and the path from the root node to the leaf node $p:2$ corresponds to two transactions with the same frequent items $\{f, c, a, m, p\}$. The algorithm also keeps track of an auxiliary data structure called "Header table". Each entry in the Header Table contains the name of a frequent item s, the support of s, and a list (dotted arrows in Figure A.1) of all nodes in the FP-tree that represents s.

The mining process recursively constructs smaller FP-trees, each corresponds to a partition in D_f. For each frequent item s in the Header Table, the algorthm finds all the paths that contains s, starting with frequent items that have the lowest support. This process can be done efficiently using the links stored in the Header Table. For example, for the frequent item m, the paths that contain m as the deepest node represent all transactions in D_f that contain m. Such paths are highlighted in grey in Figure A.2 and we call the set of transactions that corresponds to such paths but with m filtered out D'_f, indicated in label 6 in Figure A.2. To mine a pattern, the algorithm uses the following idea: if m has sufficient support in D'_f, then $\{f, c, a, b, m\}$ also have sufficient support in D_f. The recurisive step of the algorithm is based on such idea, and the mining process continues on the new database D'_f. We repeat the process until the FP-tree is empty or contains only a single branch.

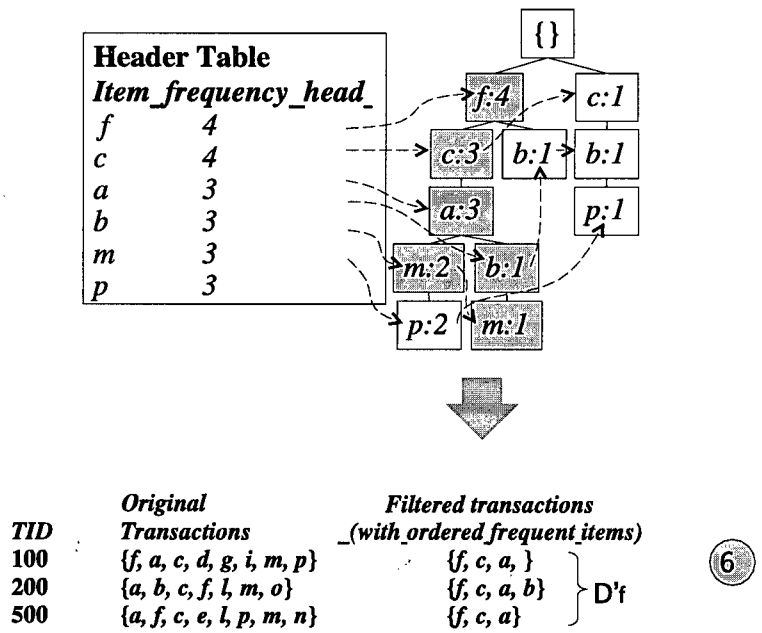

Figure A.2: Illustration of the recursive step in the FP-tree based algorithm

Bibliography

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 246–256, June 1990.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proceedings of the International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[4] R. Arnold and S. Bohner. *Software Change Impact Analysis.* IEEE Computer Society Press, 1996.

[5] S. Brin, R. Motwani, and C. Silverstein. Beyond market baskets: generalizing association rules to correlations. In *Proceedings of the International Conference on Management of Data*, pages 265–276, 1997.

[6] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the International Conference on Software Engineering*, pages 408–418, 2003.

[7] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *Transactions on Software Engineering*, 17(8):751–761, 1991.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *Proceedings of the Internation Conference on Management of Data*, pages 1–12. ACM Press, 05 2000.

[9] S. M. Jelber Sayyad Shirabad, Timothy C. Lethbridge. Supporting maintenance of legacy software with data mining techniques. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, 2000.

[10] M. Kamber, J. Han, and J. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Knowledge Discovery and Data Mining*, pages 207–210, 1997.

[11] B. Magnusson and U. Asklund. Fine grained version control of configurations in coop/orm. In *Proceedings of the International Synposium on System Configuration Management*, pages 31–48, 1996.

[12] A. Michail. Data mining library reuse patterns in user-selected applications. In *Proceedings of the International Conference on Automated Software Engineering*, pages 24–33, 1999.

[13] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the International Conference on Software Engineering*, pages 167–176, 2000.

[14] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[15] J. S. Park, M.-S. Chen, and P. S. Yu. Using a hash-based method with transaction trimming for mining association rules. In *Transactions on Knowledge and Data Engineering*, pages 813–825, 1997.

[16] D. L. Parnas. On the criteria to be used in decomposing systems into module. In *Communications of ACM*, pages 1053–1058, 1972.

[17] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the International Conference on Management of Data*, pages 1–12, Montreal, Quebec, Canada, 4–6 1996.

[18] R. Srikant and R. Agrawal. Mining generalized association rules. *Future Generation Computer Systems*, 13(2–3):161–180, 1997.

[19] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering*, pages 107–119, 1999.

[20] M. Weiser. Program slicing. In *Transactions of Software Engineering*, volume 10, pages 352–357, July 1984.

[21] Y. Ye and G. Fischer. Information delivery in support of learning reusable software components on demand, 2002.

[22] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In *Proceedings of the International Workshop on Principles of Software Evolution*, 2003.

[23] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. 2003.