Predicting Source Code Changes by Mining Revision History

Annie T.T. Ying*+, Gail C. Murphy*, Raymond Ng*
Dep. of Computer Science, U. of British Columbia*
{aying,murphy,rng}@cs.ubc.ca

Mark C. Chu-Carroll+
IBM T.J. Watson Research Center+
mcc@watson.ibm.com

1 Introduction

Many modification tasks to software systems require software developers to change many different parts of a system's code base [7]. To help identify the relevant parts of the code for a given task, a developer may use a tool that statically or dynamically analyzes dependencies between parts of the source (e.g., [8, 1]). Such analyses can help a developer locate code of interest, but they cannot always identify *all* of the code relevant to the change. For example, these analyses cannot typically identify dependencies between modules written in different programming languages.

To illustrate the difficulty developers sometimes face in finding relevant source code during a modification task, we outline the changes involved in a modification task[1] from the Mozilla development history. Mozilla is a web browser primarily written in C++ and is a large open source project. Modification task #150339, 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 was displayed. As part of a solution[2] for this modification task, a developer added code to limit the font size in the version of Mozilla that uses the `gtk` UI toolkit, but missed a similar required change in the version that uses the UI toolkit `xlib`. The source code in `gtk/nsFontMetricsGTK.cpp` does not reference the code in `xlib/nsFontMetricsXlib.cpp` because the code in the `gtk` version and the code in `xlib` version are used in different configurations of Mozilla. However, an analysis of the CVS revision history for Mozilla indicates that these two files were changed 41 times together in the development of Mozilla.

To augment the existing static and dynmaic analyses to help developers identify relevant code, we have been investigating an approach based on the mining of change patterns—files that have changed together *frequently enough*—from a system's source code versioning information. Mined change patterns can be used to recommend possibly relevant files as a developer performs a 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 [2] to determine the change patterns.

To assess our approach, we evaluate the recommendations our tool can make on two large open source projects, Eclipse[3] and Mozilla[4], based on recommendations' *predictability* and the *interestingness*. Predictability 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 measure the value of correct recommendations.

2 Approach

Our approach consists of three stages. In the first stage, we extract the data from a software configuration management (SCM) system and pre-process the data 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 the first stage, we need to determine which software artifacts—in our case, files—were checked in together. Systems such as CVS, which is used for the systems we targeted in our validation, do not track this information; 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 [6].

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

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

[3] URL at `http://www.eclipse.org/`

[4] URL at `http://www.mozilla.org/`

In the second stage, we use an association rule mining algorithm to extract sets of items that happen frequently enough amongst the transactions in a database. In our context, such sets refer to source files that tend to change together. In our study, we use frequent pattern mining [2], which finds recurring sets of items—or source files in our context—among transactions in a database D. The strength of the pattern $\{s_1, \ldots, s_n\}$ is measured by *support*, which is the number of transactions in D containing $s_1, \ldots,$ and s_n. A frequent pattern describes a set of items that has support greater than a predetermined threshold $min\_support$.

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 $f_s \rightarrow f_R$.

3 Validation

The validation process involved determining if source code recommended from a change pattern was relevant for a given modification task. This validation process required dividing the development history information for a system into training and test data. The training data was used to generate change patterns that were then used to recommend source for the test data.

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[5], which also keeps 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 for a modification task, we use heuristics based on development practices to determine this information. 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.

To recommend possibly 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; we chose this approach because it represents the minimum amount of knowledge a developer would need to generate a recommendation. We evaluate the usefulness of the recommended files f_R in terms of two criteria: predictability and interestingness, described in the rest of Section 3.

[5]URL at `http://www.bugzilla.org/`

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 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 2.

$$precision_{f_R} = \frac{|f_R \cap f_{sol}|}{|f_R|} \quad (1)$$

$$recall_{f_R} = \frac{|f_R \cap f_{sol}|}{|f_{sol} - f_s|} \quad (2)$$

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. These structural hints between two source code fragments in the Java programming language that we consider in our analysis are field accesses, field writes, method calls, instance creation, class checks, header-implementation relationship, inheritance, method declaration ,field declaration, and whether source code fragments are in the same directory.

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, or getting information about a class' fields).

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 enable a user to search for references, declarations, and implementors of direct forward and backward references for a given point in the source (e.g., an integrated development environment).

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— to a method, field or class in f_r, *or*
- a class *that was changed* in f_s has a strong coarse-grained relationship—inheritance, method declaration, field declaraion—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.

3.1 Predictability results

Figure 1 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 label beside each data point indicates the *min_support* threshold used in the frequent pattern mining algorithm. Each data point represents the average precision and recall. In the computation

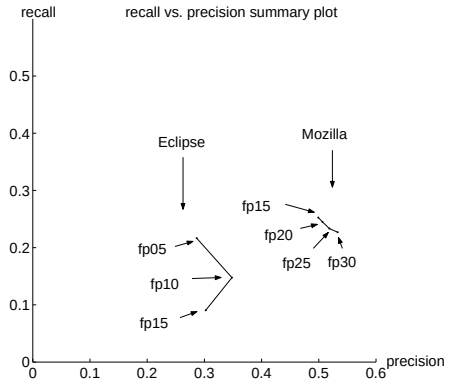

Figure 1. Recall prec. on Eclipse & Mozilla.

of the averages, we included only the precision and recall for recommendations from modification tasks M in the test data where each task's solution contained at least one file from a change pattern. We used each file f_s in the solution of a modification task $m \in M$ to generate a set of recommended files f_R and calculated f_R's precision and recall described in 3. 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 few recommendations can be made.

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. 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.

Table 1 presents a categorization of recommendations for Mozilla. We were able to generate recommendations for 15 of the 20 selected modification tasks. These 15 tasks include two cases involving *surprising* recommendations, two

Int.	Description	Mod. ids (and no. of recmm.)
sur	cross-language	150099(6)
sur	duplicate code bases	92106(2),143094(2),145560(2), 145815(2), 150339(2)
neu	distant instance creation/being created & call dependence	123068 (2)
neu	distant inheritance	74091 2)
obv	header-implementation	92106(2),99627(2),104603(2), 135267(8),144884(2), 150735(2),74091(2),123068(2)
obv	interface-implementation	135267(2)
obv	direct inheritance	74091(12)
obv	direct call dependence	99627(2)
obv	same package with less than 20 files	135267(6)

Table 1. Mozilla recom. by interestingness

Int.	Description	Mod. ids (and no. of recom)
sur	cross-platform/XML	24635(230)
neu	distant call dependence	21330(2), 24567(2)
neu	distant inheritance	25041(12)
obv	containment	13907(2), 23096(2), 24668(2)
obv	framework	21330(2)
obv	same package with less than 20 files	21330(2)
obv	instance creation/being created	23587(4)
obv	method call dependence	21330(2), 23587(4), 24657(2), 25041(2)
obv	direct inheritance	25041(8)
obv	interface-implementation	24730(2)

Table 2. Eclipse recom. by interestingness

cases involving *neutral* recommendations, and five cases involving *obvious* recommendations. We focus the discussion on some of the cases categorized as *surprising*.

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 example, 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 in the web browser by default. The solution involved adding a new menu item for displaying the user's preference of showing or hiding the tab bar in a XUL file, declaring the 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[6], 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. For example, 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.

Table 2 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 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 that we focus on.

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. For example, 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

[6]The terminology "Javascript-XUL" means that given a Javascript file, an XUL was recommended, and given an XUL file, a Javascript file was recommended

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.

4 Related Work

Zimmermann and colleagues, independently from us, have developed an approach that also uses association rule mining on CVS data to recommend source code that is potentially relevant to a given fragment of source code [9]. The rules determined by their approach can describe change associations between files or more fine-grained entities such as methods. Both Zimmermann's approach and ours produce similar quantitative results: precision and recall that measure the predictability of the recommendations are similar in value. The qualitative analyses differ. They present some change associations that were generated from their approach, and argue that these associations are of interest. In contrast, we analyzed the recommendations provided in the context of completed modification tasks, emphasizing when the results would be of value to a developer. We assessed the value of the recommendations using the interestingness criteria that we developed.

Hipikat is a tool that provides recommendations about project information a developer should consider during a modification task [4]. 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. When a recommendation is requested based on a description of a modification task at hand, Hipikat recommends similar modifications completed in the past, with their associated file revisions. Our approach is complementary to Hipikat, as it does not rely upon a similar modification task having occurred in the past.

Impact analysis approaches (e.g., [3]) 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 may 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., [5]) and dynamic slicing (e.g., [1]). 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. In addition, dynamic slicing relies on an executable program and the availability of appropriate inputs of the program, whereas our approach can work with code that is non-executable, or with code that consists of components running on different platforms. On the other hand, slicing approaches can provide finer-grained information about code related to a modification task, without relying on the code being changed repeatedly in the past.

5 Conclusion

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 validated our hypothesis that our approach can provide useful recommendations by applying the approach to two open-source systems, Eclipse and Mozilla, and then evaluating the results based on the predictability and likely interestingness to a developer. Although the precision and recall are not high, recommendations can reveal valuable dependencies that may not be apparent from other existing analyses. In addition to providing evidence for our hypothesis, we have developed a set of interestingness criteria for assessing the utility of recommendations; these criteria can be used in qualitative analyses of source code recommendations provided by other systems.

References

[1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc.of PLDI*, 1990.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proc. of ICMD*, 1993.

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

[4] D. Cubranic and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc.of ICSE*, 2003.

[5] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *TSE*, 17(8), 1991.

[6] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *TOSEM*, 11(3), 2002.

[7] D. L. Parnas. On the criteria to be used in decomposing systems into module. In *CACM*, 1972.

[8] M. Weiser. Program slicing. *TSE*, 10(7), 1984.

[9] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *To appear in Proc.of ICSE*, 2004.