Using Version Information for Concern Inference and Code-Assist

Annie T.T. Ying, Gail C. Murphy, Raymond T. Ng
Department of Computer Science
University of British Columbia
2366 Main Mall
Vancouver BC Canada V6T 1Z4

aying@cs.ubc.ca, murphy@cs.ubc.ca, rng@cs.ubc.ca

Mark C. Chu-Carroll
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne NY USA 10591

mcc@watson.ibm.com

ABSTRACT

We propose an approach to assist software developers perform modification tasks that may involve source code artifacts that crosscut the system's code base. Our hypothesis is that given some initial knowledge about which source code artifacts (e.g., methods or classes in an object-oriented language) may need to change in a modification task at hand, past modification tasks help predict the additional source code artifacts that need to change to complete the desired modification. Our approach is to analyze the past modification tasks using association rule mining. Essentially, association rule mining finds relationships between source code artifacts that tend to change together. We envision a tool that uses this approach to interactively guide software engineers during the implementation of modification tasks. The tool might also help identify concerns since code related to a concern likely changes together.

1. INTRODUCTION

Identifying what source code needs to change for a modification task on an existing software system can be difficult. The existing source code may be well-modularized with respect to a limited number of anticipated modification tasks but will not likely be well-modularized for all tasks. As a result, software developers are often faced with modification tasks that involve changes to source code artifacts (e.g., method or classes in an object-oriented language) that are spread across the code base.

In this paper, we propose an approach to assist software engineers who are performing modifications to an existing system. Our hypothesis is that information about past modification tasks is useful in guiding developers to implement a modification task at hand. We assume that the source code of a software system is stored in a version management system, such as CVS [CVS] or Clear Case [LeBlang94]. In addition, we assume that the implementation of a modification task corresponds to the set of changes to source code artifacts between two source code versions. In other words, the version management system keeps track of the history of completed modification tasks. The general framework of our approach is that given some initial knowledge about the source code artifact changes in a modification task at hand, past modification tasks can be useful in predicting the remaining source code artifacts that need to change to implement the desired modification. We propose to analyze the past modification tasks using association rule mining [Agrawal93]. In essence, association rule mining on the past modification tasks finds relationships between source code artifacts that tend to change together.

We plan to build a tool that uses this approach to assist software developers during the implementation of modification tasks. The tool will be highly integrated into the software implementation phase. The suggested rules found by our tool could be used in a way similar to the code-assist functionality in an integrated development environment, such as Eclipse [Eclipse]. For example, for the Eclipse code-assist, as a programmer is typing part of a method call in a source code editor, Eclipse automatically displays a list of method calls that match what the programmer has typed. Our tool will suggest a list of method calls that might need to be made.

One novelty of our approach is that source code versions stored in a version management system are analyzed for knowledge discovery; this contrasts with the original purpose of using version management systems for archival and retrieval. Another novelty is that our approach does not directly use conventional program analysis techniques to infer information about a system's source code.

We believe that in a reasonable number of cases the source code artifacts involved in a modification task are part of a concern that possibly crosscuts the system. When performing a modification task, software developers can use the source code artifacts suggested by our tool to examine a concern that may relate to the desired modification task. Our approach is likely applicable for such concern exploration when the system has rich modification history but does not have an appropriate implemented mechanism to support separation of crosscutting concerns.

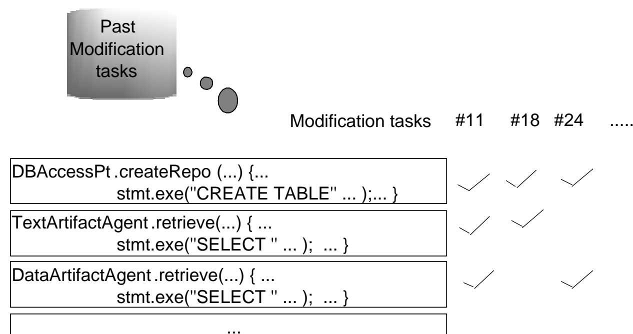

Figure 1: Past history that involves the concern of storage code of a system containing JDBC database table creation and accesses.

2. SCENARIOS

To demonstrate our approach, we consider two modification tasks that involve crosscutting concerns.

2.1 Scenario 1 - A Modification Task Involving a Database Storage Concern

We consider a modification task that touches the concern involving the storage code of JDBC database table creation and accesses. In the system's modularization, the Java source code implementing the concern are spread across multiple classes in different packages. The modification task is to change the name of an attribute of a JDBC table in SQL statements embedded in Java source code.

As shown in Figure 1, from the past modification tasks, we can determine that a change to the method issuing the JDBC calls to create a table (method `DBAccessPt.createRepo`) tends to occur with changes to the methods that issue JDBC calls to access that table (methods `TextArtifactAgent.retrieve` and `DataArtifactAgent.retrieve`) from the association rule

$$\{\texttt{DBAccessPt.createRepo}\} \Rightarrow \{\texttt{TextArtifactAgent.retrieve}, \texttt{DataArtifactAgent.retrieve}\}.$$

For example, modification task #11 in Figure 1 involves changes to both the method containing the table creation statement and the two methods containing the table accesses statements. Obviously, the reason behind such an association rule is that when changing the name of an attribute in a table in JDBC, all accesses of the changed attribute in the table must be updated. When a developer makes a change to `DBAccessPt.createRepo`, we envision our tool will suggest that methods `TextArtifactAgent.retrieve` and `DataArtifactAgent.retrieve` are likely to change as well.

If a developer neglected to change the references to some of the attribute names, a compiler would not be able to catch this error because the SQL statements are string literals in the Java source code. Conventional program analysis approaches would require a specific SQL static analyzer in order to give this suggestion. However, in this scenario, our approach just depends on the fact that methods containing the changed string literals have tended to change together in the past.

2.2 Scenario 2 - A Modification Task Involving UI Code

We consider making a modification to the integrated development environment Eclipse [Eclipse]. Eclipse supports team programming using CVS [CVS]. The modification task is to enable a user to specify a default value of the host location of the CVS repository. This modification involves making changes to the preferences configuration UI page and changes to the team programming related UI code (such as entries in a pop-up menu associated with a project icon). We believe that our approach is applicable since adding new features in the UI tends to have a certain pattern. Using our approach, software engineer can identify changes to the UI based on the patterns discovered from the past changes.

3. APPROACH

In this section, we describe our method of analyzing the past modification tasks using association rule mining. We assume that the source code is stored in the repository in a version management system and a modification task corresponds to the difference between two source code versions. The data to be mined is the source code repository.

3.1 Mining Association Rules on Changed Source Code Artifacts

The problem of finding association rules is often referred to the *market basket problem* [Agrawal93]. Intuitively, market basket data corresponds to a database of transactions of customers' purchases D, or market baskets, in retail organizations. In this setting, the goal is to find the *association rules* of the form "when a customer purchases item x, the customer is likely to also purchase item y". More precisely, given basket data D, the goal is to generate all association rules of the form $x \Rightarrow y$ with *support* greater than a user-specified threshold s (i.e., both x and y occur together in at least s% of the market baskets) and *confidence* greater than a user-specified threshold c (i.e., of all baskets containing x, at least c% also contain y). This definition can be generalized to $X \Rightarrow Y$ in which X and Y are disjoint sets of items instead of just a single item. For a formal treatment of the problem, see [Agrawal93].

In our application, the source code repository is a database D of implemented modification tasks; each modification task T is a set of source code artifact changes between two source code versions. In this setting, the association rule $x \Rightarrow y$ means that when the source code artifact change x is present in a modification task, the source code artifact change y is likely to be present in the same task. This definition can be generalized to $X \Rightarrow Y$ in which X and Y are disjoint sets of source code artifact changes.

3.1.1 Dealing with Low Support of Rules

We anticipate that some *interesting* rules in the source code version repository may not occur frequently enough to have support greater than the minimum threshold. Because the number of association rules discovered depends on the user-specified support threshold (and confidence threshold), choosing a good support threshold is tricky. Setting

the support threshold too high would eliminate potentially interesting rules; setting the support threshold value too low would kill the efficiency and would likely return many uninteresting rules, which would cause problems with the acceptance of the approach by developers. We may need to use alternative mechanism other than support and confidence to measure the interestingness of the rules.

3.1.2 Cleaning Negative Data

Source code artifacts that frequently change together do not necessarily correspond to a correct modification and may possibly correspond to bugs. To avoid such misleading suggestions, we need to distinguish such *negative* data by correlating bug fixing reports with modification tasks in the source code repository. We need to exclude this negative data in the computation of association rules.

3.2 Strengths of the Approach

Our approach has at least three strengths. The algorithm we use for finding association rules is scalable, incremental, and able to handle second-hand data.

Scalable

Data mining algorithms are designed to be scalable. There are a series of proposals addressing on scalibility of association rule mining algorithms (e.g., [Agrawal94,Park97]).

Incremental

The algorithm for finding association rules can incrementally adopt the completed modification tasks between source code versions. A non-incremental algorithm would require a recomputation on all the modification tasks in the repository; the performance may not be acceptable if we would like to interactively guide the developer through a modification task. The algorithm would depend on an association rule mining algorithm that handles dynamic data [Cheung98]. Using conventional program analysis techniques to analyze change impact cannot easily handle incremental updates to the source code because such techniques often require global reanalysis of the source code.

Capable of handling second-hand data

We can apply data mining algorithms to data that have been already collected. The original purpose of the source code repository was for managing versions of the system, but we can reuse this data for knowledge discovery. No additional collection or generation of data is required.

4. IMPLEMENTATION

4.1 Issues

There are several implementation issues that we will address in our tool.

Granularity of the source code artifacts

The granularity of versioning in the version management system determines the precision of the association rules inferred. Many version management systems, including CVS, use file-level granualarity. We think that file-granularity is too coarse; many association rules would look redundant because each source code artifact unit may include changes that are irrelevant to each other. We believe our approach will work well at the method-granularity. We plan to build our code-assist tool on a version management system called Stellation [Chu-Carroll02, Stellation], which will support fine-grained (which includes method-grained) versioning shortly.

Lack of data

Because our hypothesis is that past history is useful in predicting the source code artifact changes in the current modification task, we must have a big enough database of source code versions from which to mine rules. Not all software projects have such a history available.

4.2 Exploring Concerns Related to a Modification

Our tool can be integrated into FEAT [Robillard02], a concern exploration tool that shows representation of concerns of the system. A software developer can incrementally build up the concern using suggestions given by our tool. Using FEAT, the developer can store the representation of the concern and further explore the concern.

5. SUMMARY

In this paper, we have described an approach that provides suggestions on what source code artifacts tend to change together in the past. Such suggestions are useful when performing a modification task that involves a crosscutting concern. To evaluate our approach, we will build a tool that assists software developers perform modification tasks.

6. REFERENCES

[Agrawal93] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. *Mining association rules between sets of items in large databases.* In Proc. of the ACM SIGMOD Conference on Management of Data, pages 207-216, May 1993.

[Agrawal94] Rakesh Agrawal and Ramakrishnan Srikant. *Fast Algorithms for Mining Association Rules.* In Proc. of the 20th Int'l Conference on Very Large Databases, pages 487-499, September 1994.

[Aggarwal98] Charu Aggarwal and Philip Yu. *Online generation of association rules.* In Proceedings of 14 th Intl. Conf. on Data Engineering (ICDE'98), pages 402-411, February 1998.

[Cheung96] David W. Cheung, Jiawei Han, Vincent T. Ng, and C.Y. Wong. *Maintenance of Discovered Association Rules in Large Databases: An Incremental Update Technique.* In Proceedings of the 12th International Conference on DataEngineering (ICDE'96), pages 106-114, March 1996.

[Chu-Carroll02] Mark Chu-Carroll, James Wright, and David Shields. *Supporting Aggregation in Fine Grained Software Configuration Management.* To appear in FSE 2002, November 2002.

[CVS] CVS (Concurrent Versions System) web site:
http://ccvs.cvshome.org/servlets/ProjectHome

[Eclipse] Eclipse web site:
http://www.eclipse.org/

[Park97] Jong Soo Park, Ming-Syan Chen and Philip S. Yu. *Using a Hash Based Method with Transaction Trimming for Mining Association Rules.* IEEE Transactions on Knowledge and Data Engineering. Volume 9, Number 5, pages 813-825, September 1997.

[LeBlang] David LeBlang. *The CM Challenge: Configuration Management that works.* In Configuration Management, pages 1-37. John Wiley and Sons, June 1994.

[Robillard02] Martin P. Robillard and Gail C. Murphy. *Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies.* Proceedings of ICSE 2002, pages 406-416, May 2002.

[Stellation] Stellation web site:
http://www.eclipse.org/technology/index.html