Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System

Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh
Dept. of Computer Science
University of Maryland
College Park, MD, 20742 USA
{jspacco,strecker,daveho,pugh}@cs.umd.edu

ABSTRACT

Most computer science educators hold strong opinions about the "right" approach to teaching introductory level programming. Unfortunately, we have comparatively little hard evidence about the effectiveness of these various approaches because we generally lack the infrastructure to obtain sufficiently detailed data about novices' programming habits.

To gain insight into students' programming habits, we developed Marmoset, a project snapshot and submission system. Like existing project submission systems, Marmoset allows students to submit versions of their projects to a central server, which automatically tests them and records the results. Unlike existing systems, Marmoset also collects fine-grained code snapshots as students work on projects: each time a student saves her work, it is automatically committed to a CVS repository.

We believe the data collected by Marmoset will be a rich source of insight about learning to program and software evolution in general. To validate the effectiveness of our tool, we performed an experiment which found a statistically significant correlation between warnings reported by a static analysis tool and failed unit tests.

To make fine-grained code evolution data more useful, we present a data schema which allows a variety of useful queries to be more easily formulated and answered.

1. INTRODUCTION

While most computer science educators hold strong opinions about the "right" way to teach introductory level programming, there is comparatively little hard evidence to support these opinions. The lack of evidence is especially frustrating considering the fundamental importance to our discipline of teaching students to program. We believe that the lack of evidence is at least partly attributable to a lack of suitable infrastructure to collect quantitative data about students' programming habits.

To collect the desired data, we have developed Marmoset, an automated project snapshot, submission, and testing system. Like many other project submission and testing systems ([11, 6, 7, 4]), Marmoset allows students to submit versions of their work on course projects and to receive automatic feedback on the extent to which submissions meet the grading criteria for the project. The grading criteria are represented by JUnit [8] tests, which are automatically run against each version of the project submitted by the student. In addition to JUnit tests, Marmoset also supports running the student's code through static analysis tools such as bug finders or style checkers. Currently the only supported static checker is FindBugs [5]; we plan on trying with other static analysis tools such as PMD [10] and CheckStyle [1] in the future.

A novel feature of Marmoset is that in addition to collecting submissions explicitly submitted by students, an Eclipse [3] plugin called the Course Project Manager [13] automatically captures snapshots of a student's code to the student's CVS [2] repository each time she saves her files. These intermediate snapshots provide a detailed view of the evolution of student projects, and constitute the raw data we used as the basis for the experiments described in this paper.

Students can log in to the SubmitServer to view the results of the unit tests and examine any warnings produced by static checkers. The test results and static analysis warnings are divided into four categories:

- Public Tests: The source code for the public tests is made available to students upon their initial checkout of a project, and the results of public tests for submitted projects are always visible to students. (Students should already know these results since they can run these tests themselves).

- Release Tests: Release tests are additional unit tests whose source code is not revealed to students. The outcomes of release tests are only displayed to students if they have passed all of the public tests. Rather than allowing students unlimited access to release test results (as we do with public tests results), we allow limited access as follows. Viewing release tests costs one "release token". Students receive three release tokens for each project and these tokens regenerate every 24 hours. Viewing release results allows the student to see the *number* of release tests passed and failed as well as the *names* of the first two tests failed. For example, for a project requiring the evaluation of various poker hands, a student may discover that they have passed

6 out of 12 release tests and that the first two tests failed were *testThreeOfAKind* and *testFullHouse*. We have tried to make the release test names descriptive enough to give the student some information about what part of their submission was deficient, but vague enough to make the students think seriously about how to go about fixing the problem.

- Secret Tests: Like release tests, the code for secret tests is also kept private. Unlike release tests, the results of secret tests are never displayed to the students. These are equivalent to the private or secondary tests many instructors use for grading purposes. Although our framework supports them, the courses on which we report in this paper did not use any secret tests.

- Static Checker Warnings: We have configured Marmoset to run FindBugs on every submission and make the warnings visible to students. FindBugs warnings are provided solely to help students debug their code and to help us tune FindBugs; the results of FindBugs are not used for grading.

When compared to previous work, we feel Marmoset improves data collection in two major ways. First, by using the Course Project Manager Eclipse plugin, we can gather frequent snapshots of student code automatically and unobtrusively. Prior work on analyzing student version control data [9] focused on data that required the students to manually commit their code. One observation made by Liu et. al. is that students often don't use version control systems in a consistent manner. The Course Project Manager plugin has no such limitation.

Second, by providing the same testing framework for both development and grading, we can quantify the correctness of any snapshot along the development timeline of a project. This allows us to perform statistical analyses of the development history of each student.

2. STUDENT SNAPSHOT DATA

Of the 102 students in the University of Maryland CMSC 132 Fall 2004 course, 73 consented to be part of an IRB approved experimental study of how students learn to develop software. Other than signing a consent form and filling out an optional online survey about demographic data and prior programming experience, students participating in the experiment did not experience the course any differently than other students in the course, as the data collected for this research is routinely used during the semester to provide students with regular backups, automated testing and a distributed file system. From the 73 students who consented to participate in the study, we extracted from their CVS repositories over 51,502 snapshots, of which about 41,333 were compilable. Of the compilable snapshots, 33,015 compiled to a set of classfiles with a unique MD5 sum.

That 20% of the snapshots did not compile is not surprising, as snapshots are triggered by saving. In fact, we were pleasantly surprised that so many of our snapshots did compile.

We tested each unique snapshot on the full suite of unit tests written for that project. In addition, we checked each unique snapshot with the static bug finder FindBugs [5] and stored the results in the database. We also computed the CVS diff of the source of each unique submission with the source of the preceding unique submission, and stored the total number of lines added or changed as well as the net change to the size of the files (we do not track deletes explicitly, though deletes do show up indirectly as net changes to the size of the source files).

students	73
projects	8
student projects	569
snapshots	51,502
compilable	41,333
unique	33,015
total test outcomes	505,423
not implemented	67,650
exception thrown	86,947
assertion failed	115,378
passed	235,448

Table 1: Overall numbers for project snapshots and test outcomes

	Exception			
	yes		no	
	Warning		Warning	
Problem	yes	no	yes	no
ClassCast	362	1,775	1,306	30,878
enhanced	1,047	1,477	n/a	n/a
StackOverflow	279	1,140	2	31,594
enhanced	935	793	n/a	n/a
Null Pointer	267	5,863	382	26,503

Table 2: Correlation between selected warnings and Exceptions

We have performed a number of different kinds of analysis on the data, and continue to generate additional results. Unfortunately, space only allows us to present a small window into our research.

We have looked both at the changes between successive snapshots by an individual student, and at the features of each snapshot in isolation. When looking at changes between successive snapshots, we can examine the change in warnings between successive versions and whether there is any corresponding change in the number of unit test faults between versions. We can also look at the size of changes, and even manually examining the differences between versions where our defect warnings do not seem to correspond to the difference in actual faults (e.g., if a one line change caused a program to stop throwing NullPointerExceptions, but no change occurred in the number of defect warnings generated, is there something missing in our suite of defect detection tools?). While we have some results from this analysis, the complexity of those results makes them hard to present in the space available.

3. CORRELATION BETWEEN WARNINGS AND EXCEPTIONS

In this section, we show the correlation between selected bug detectors and the exceptions that would likely correspond to the faults identified by these detectors. We look at ClassCastExceptions, StackOverflowError and NullPoint-

erExceptions. Before starting work on finding bugs in student code, we didn't have any bug detectors for ClassCastExceptions or StackOverflowErrors. Based on our experience during class and leading up to this paper, we wrote some detectors for each. Table 2 shows the correlation between exceptions and the corresponding bug detectors.

ClassCastExceptions typically arise in student code because of:

- An incorrect cast out of a collection. We believe that many of these would be caught by uses of parameterized collections.
- A collection is downcast to more specific class

```
(Set)Map.values()
```

- A cast to or from an interface that will not succeed in practice, but the compiler cannot rule out since it can't assume new classes will not be introduced. In the example below, although WebPage does not implement Map, we cannot rule out the possibility that a new class could be written that extends WebPage and implements Map:

```
public void crawl(WebPage w) {
  Map crawlMap = (Map)w;
```

- A cast where static analysis dooms the cast, even if additional classes are written, but the programmer has gone to some length to confuse the compiler:

```
public WebPage(URL u) {
  this.webpage = (WebPage)((Object)u);...+
```

We have written detectors to check for the last three casts. Surprising, all three (even the last one) also identify problems in production code; an instance of the the last error occurs in the Apache Xalan library.

The last two detectors for ClassCastExceptions were only written shortly before the camera-ready deadline based on manual examination of the snapshots that generated ClassCastExceptions but were not flagged as possibly containing a bad cast. The numbers with those detectors included are reported on the line labeled enhanced. By the camera-ready deadline, we were unable to rerun the detectors on the 30,878 snapshots in which no ClassCastException occurred to see how many false positives they generated.

Many of the StackOverflowErrors are caused by code that obviously implements infinite recursive loops, such as:

```
WebSpider() {
  WebSpider w = new WebSpider(); }
```

We wrote an initial detector based on experience during the fall semester, and that detector also found a number of infinite recursive loops in production code such as Sun's JDK 1.5.0 and Sun's NetBeans IDE.

Based on manual examination of snapshots that threw StackOverflowError but were not flagged as containing recursive infinite loops, we improved the detectors shortly before the camera-ready deadline. The numbers with the enhanced detector are reported on the line labeled enhanced. By the camera-ready deadline, we were unable to rerun the detectors on the 31,596 snapshots in which no StackOverflowError occurred to see how many false positives they generated.

For the NullPointerExceptions, we report just the detectors that perform dataflow analysis to report possible NullPointerExceptions. We also have detectors that look for uninitialized fields containing references (not reported in this table); these are also the source of a number of exceptions, but they have a higher false positive rate than the field based detectors.

4. SCHEMA FOR REPRESENTING PROGRAM EVOLUTION

Our current analysis is somewhat limited, in that we can only easily measure individual snapshots, or changes between successive versions. We can't easily track, for example, which changes are later modified.

We want to be able to integrate code versions, test results, code coverage from each test run, and warnings generated by static analysis tools. In particular, we want to be able to ask questions such as:

- Which methods were modified during the period 6pm-9pm?
- During the period 6pm-9pm, which methods had more than 30 line modifications or deletions?
- Of the changes that modified a strcpy call into a strncpy call, how frequently was the line containing the strncpy call, or some line no more than 5 lines before it, modified in a later version?
- For each warning generated by a static analysis tool, which versions contain that warning?
- Which warnings are fixed shortly after they are presented to students, and which are ignored (and persist across multiple submissions)?

None of these questions can be easily asked using CVS based representations. We developed a schema/abstraction for representing program histories that make answering these questions much easier. A diagram of the schema is shown in Figure 1. Each entity/class is shown as a box, with arrows to other entities it has references to. This schema can be represented in a relational database, and most of the queries we want to ask can be directly formulated as SQL queries.

The schema we have developed is based on recognizing unique lines of a file. For example, we might determine that a particular line, with a unique key of 12638 and the text “ `i++;` ”, first occurs on line 25 of version 3 of the file “Foo.java”, occurs on line 27 in version 4 (because two lines were inserted before it), occurs on line 20 in version 5 (because 7 lines above it were deleted in going from version 4 to version 5) and that version 5 is the last version unique line #12638 occurs.

It is important to understand that unique lines are not based on textual equality. Other occurrences of “ `i++;` ” in the same file or other files would be different unique lines. If a line containing “ `i++;` ” is reinserted in version 11, that is also a different unique line.

So in our database, we have a table that gives, for each line number of each file version, the primary key of the unique line that occurs at that line number.

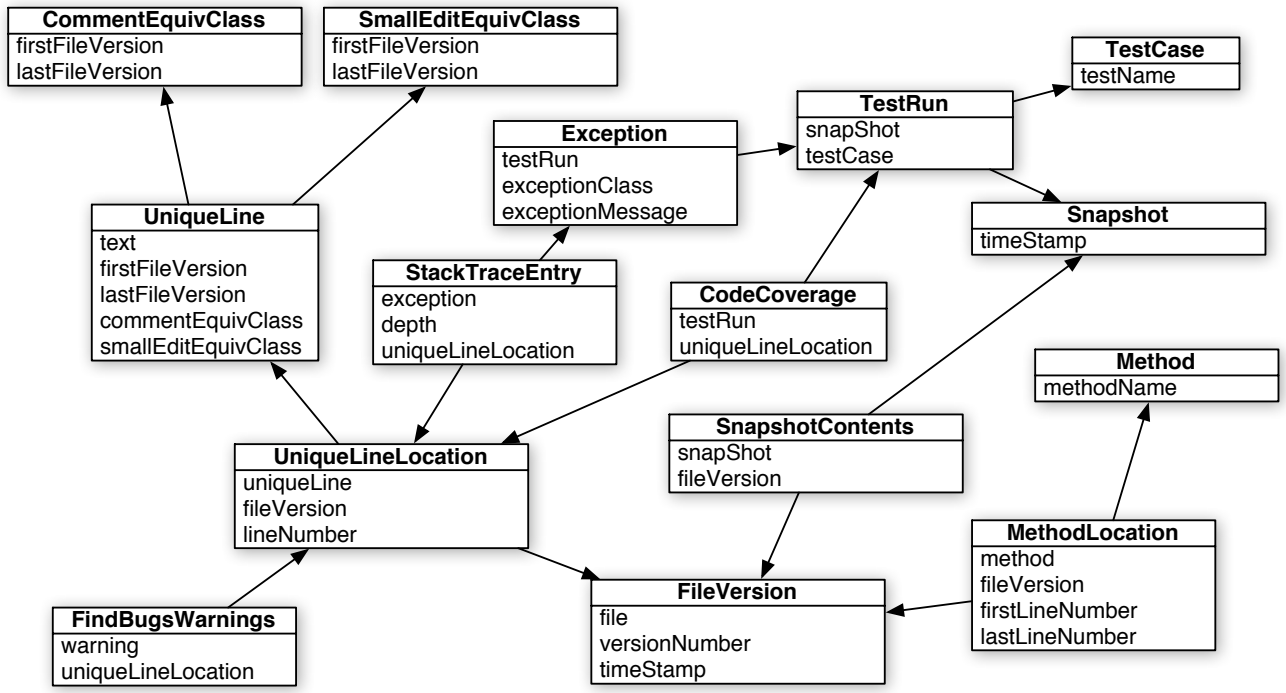

Figure 1: Schema for representing program evolution

4.1 Tracking Lines and Equivalence Classes

As given, two lines are considered identical only if they are textually identical: changing a comment or indentation makes it a different unique line. While we sometimes want to track changes at this granularity, we often want to track lines across versions as their comments are changed or even as small modifications are made.

We handle this by defining equivalence classes over unique lines of text. At the moment, we support the following equivalence relations:

- Identity: The lines are exactly identical.
- Ignore-Whitespace: When whitespace is ignored, the lines are identical.
- Ignore-SmallEdits: When whitespace is ignored, the lines are almost equal; their edit distance is small.
- Ignore-Comments: When whitespace and comments are ignored, the edit distance between the lines is small.

These equivalence relations are ordered from strictest to most relaxed. Thus, the lines " `a = b + c.foo();` " and "`a = b + x.foo(); /* Fixed bug */` " belong to the same Ignore-Comments and Ignore-SmallEdits equivalence classes, but not to the same Ignore-Whitespace and Identity equivalence classes. The equivalence classes are used to track individual lines as they evolve, not to identify textually similar lines of text.

There are various rules associated with identifying these unique lines and equivalence classes in a file:

- No crossings: If a line belonging to equivalence class X occurs before a line belonging to equivalence class Y in version 5 of a file, then in all versions in which lines belonging to equivalence classes X and Y occur, the line belonging to equivalence class X must occur before the line belonging to equivalence class Y.
- Unique representatives: In each version, only one line may belong to any given equivalence class.
- Nested equivalence classes: If two lines are equivalent under one equivalence relation, then they must be equivalent under all more relaxed relations.

The no crossing rule prevents us from recognizing cut-and-paste operations, in which a block of code is moved from one location to another. Recognizing and representing cut-and-paste (and other refactoring operations) is a tricky issue that we may try to tackle at some future point. However, handling that issue well would also mean handling other tricky issues, such as code duplication.

To calculate which lines belong to the same equivalence class, we have implemented a variation of the "diff" command to discover groups of mismatched lines, or deltas, between two versions. Our diff algorithm recursively computes deltas under increasingly relaxed equivalence relations. First, we find all deltas under the Identity relation, which is the strictest. For each delta, we apply the algorithm recursively, using the next strictest equivalence relation to compare lines. The final result is a "diff" of the versions for each equivalence relation. Because the recursive step of the algorithm only considers those deltas computed under stricter equivalence relations, the algorithm respects the three rules above.

4.2 Methods

Since we will sometimes wish to track which methods are modified by a change or covered by a test case, we also store, for each file version, the first and last line number associated with a method.

4.3 Other information

We represent a number of additional forms of information in our database. A snapshot consists of a set of file versions taken at some moment in time. Usually, a snapshot represents a compilable, runnable and testable snapshot of the system.

Associated with a snapshot we can have test results and code coverage results. Typically, each project will have a dozen or more unit test cases. We run all of the unit tests on each snapshot, and also record which lines are covered by each test case. If a test case terminates by throwing an exception, we record the exception and stack trace in the database. The information we have linking lines in different versions of a file allows us to easily compare code coverage in different versions, or correlate code coverage with static analysis warnings or exceptions generated during test cases.

5. RELATED WORK

Many systems exist to automatically collect and test student submissions: some examples are [11, 6, 7, 4]. Our contribution is to control students' access to information about test results in a way that provides incentives to adopt good programming habits.

In [9], Liu et. al. study CVS histories of students working on a team project to better understand both the behavior of individual students and team interactions. They found that both good and bad coding practices had characteristic ways of manifesting in the CVS history. Our goals for the data we collect with our automatic code snapshot system are similar, although we consider individual students rather than teams. Our system has the advantage of capturing changes at a finer granularity: file modification, rather than explicit commit.

In [12], Schneider et al. advocate using a "shadow repository" to study a developer's fine-grained local interaction history in addition to milestone commits. This approach to collecting and studying snapshots is similar to our work with Marmoset. The principal difference is that we are not focused on large software projects with multiple developers, and so we can use a standard version control system such as CVS to store the local interactions.

In [4], Edwards presents a strong case for making unit testing a fundamental part of the Computer Science curriculum. In particular, he advocates requiring students to develop their own test cases for projects, using project solutions written by instructors (possibly containing known defects) to test the student tests. This idea could easily be incorporated into Marmoset.

6. ACKNOWLEDGMENTS

The second author is supported in part by a fellowship from the National Physical Science Consortium and stipend support from the National Security Agency.

7. REFERENCES

[1] CheckStyle. http://checkstyle.sourceforge.net, 2005.

[2] CVS. http://www.cvshome.org, 2004.

[3] Eclipse.org main page. http://www.eclipse.org, 2004.

[4] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 2003 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, October 2003.

[5] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *Companion of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, BC, October 2004.

[6] D. Jackson and M. Usher. Grading student programs using ASSYST. In *Proceedings of the 1997 SIGCSE Technical Symposium on Computer Science Education*, pages 335–339. ACM Press, 1997.

[7] E. L. Jones. Grading student programs - a software testing approach. In *Proceedings of the fourteenth annual consortium on Small Colleges Southeastern conference*, pages 185–192. The Consortium for Computing in Small Colleges, 2000.

[8] JUnit, testing resources for extreme programming. http://junit.org, 2004.

[9] Y. Liu, E. Stroulia, K. Wong, and D. German. Using CVS historical information to understand how students develop software. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.

[10] PMD. http://pmd.sourceforge.net, 2005.

[11] K. A. Reek. A software infrastructure to support introductory computer science courses. In *Proceedings of the 1996 SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, PA, February 1996.

[12] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.

[13] J. Spacco, D. Hovemeyer, and W. Pugh. An eclipse-based course project snapshot and submission system. In *3rd Eclipse Technology Exchange Workshop (eTX)*, Vancouver, BC, October 24, 2004.