

Mining Data from an Automated Grading and Testing System by Adding Rich Reporting Capabilities

Anthony Allevato, Matthew Thornton, Stephen H. Edwards, and Manuel A. Pérez-Quiñones
{allevato, thornptom}@vt.edu, {edwards, perez}@cs.vt.edu
Department of Computer Science, Virginia Tech

Abstract. Programs that perform automated assignment grading can generate a great deal of meaningful data not only for the student, but for the instructor as well. Such tools are often used in computer science courses to assess student programming work. In the process of grading, a large amount of intermediate information is gathered. However, in most cases this information is not used beyond assigning scores, so the potential of learning more about the course is lost. At the same time, continuous collection of data over a large number of submissions across many courses presents an interesting but untapped resource for educational data mining. One solution to this problem is to implement a reporting tool for making use of this intermediate data to create meaningful interpretations. This paper describes how an automated grading system, Web-CAT, has been extended to provide a reporting mechanism that uses the intermediate data that is gathered during assessment of students' programs. Our implementation of the reporting tool makes use of the Business Information Reporting Tool (BIRT) developed for the Eclipse IDE.

1 Introduction

In computer science or information technology courses where students learn to write programs, many assignments focus on this skill. How do student grades improve as a function of how close they submit their assignment to the due date? Is there any correlation between the numbers of test cases that a student writes versus the grade they receive? Is there a statistically significant difference in grades from one assignment to the next? These are typical of questions that instructors may ask in analyzing the success of their lesson plans. Instructors that use some form of automated grading or testing system to help assess student work are at a significant advantage. Such systems should be able to help answer these questions because they provide a great deal of feedback to the instructor about student grades and the assessment of individual assignments.

Unfortunately, such applications usually do not provide a useful way of analyzing this information. Web-CAT, an automated testing system [3], collects a large amount of intermediate data when assessing student projects. Prior to our work outlined here, these data were effectively “trapped” within the tool and were not publicly accessible, thus preventing any analysis. Making the data accessible would require an analysis and reporting tool that would allow complex reports to be generated.

The Business Information Reporting Tools (BIRT) project is an open-source reporting engine based on the Eclipse IDE [2]. BIRT makes it possible to take raw data from some data source and render it using tables, crosstabs, and a number of chart types. It can then generate reports in several formats including PDF and HTML. BIRT is also easily extended through the Eclipse plug-in and extension point paradigms.

Integrating BIRT into Web-CAT provides a powerful resource for data analysis for a student, an assignment, or a course overall. Unfortunately, the integration was not as simple as connecting a data store from Web-CAT to BIRT and generating reports, because data from Web-CAT comes from several disparate sources. It was necessary to create a “bridge” between the Web-CAT data store and the BIRT reporting engine. Additionally, tight integration into the existing Web-CAT user interface was also desired.

This paper details our solution to the problem of analyzing the large volume of data that is acquired in assessing student assignments using an automated grading system. We will look at previous work done in analysis of student assessment data, describe the requirements of our system, and discuss the highlights of its design and implementation. Finally, we will show examples of some in-depth analysis that is now possible because of these enhancements and describe future directions for this tool and its potential to aid in student, assignment, and course assessment from the instructor’s perspective.

2 Background and Related Work

Web-CAT is a web-based system that supports the electronic submission and automated grading of programming assignments [3]. In addition to traditional models of automated grading, it also supports test-driven development by evaluating the correctness and thoroughness of unit tests that students submit with their solution. When students submit, they receive a report detailing their performance on the assignment. This report includes a score derived from such sources as static analysis and style-checking tools, code coverage monitoring tools, and the results of executing reference tests that the instructor provides. Students can also view visualizations of their past performance and their standing among other students [5]. However, a great deal of the raw data that is used to derive these final scores is left inaccessible, buried in internal databases and other file structures. Much of this data could potentially be of use to an instructor who wishes to assess either the abilities of students based on other criteria than their final cumulative scores, or the effectiveness of an assignment by examining how students at different skill and knowledge levels were able to perform on a particular aspect of the assignment.

Marmoset, developed by Spacco, Strecker, et al., is a system that performs analysis of student submissions beyond the computation of a final score for grading purposes [11]. Its notable features include the ability to capture snapshots of a student’s progress on an assignment as the solution is developed over time, and the implementation of an exception checker that can collect data about common failures that occur in student code. Our approach differs from Marmoset’s in the inclusion of a report generation engine, user-defined reports over the entire data model (limited by role-based access control), a WYSIWYG report designer with live data preview capabilities, and support for extending the data model with user-written data collectors via Web-CAT’s plug-in mechanism.

Mierle, Laven, et al. investigate mining CVS repositories used to store students’ code during development [8]. The authors analyze features such as the amount of code written, the extent of changes made between each commit, and the length of time between those commits. They attempt to draw a correlation between these metrics and a student’s performance on an assignment.

While the design and goals of the work done by these two groups may differ from ours, it does give some insight into the type of data that would be useful to make available through the reporting engine so that Web-CAT can support similar types of analyses.

3 Design and Implementation

The first step to designing this project was to select the reporting engine that would be integrated into Web-CAT. The primary requirements were that it be open-source and implemented in Java. The three major candidates were JasperReports [6], JFreeReport [10], and BIRT. After examining each in detail, we chose BIRT for its extensibility, customizability, its rich user-friendly report designer, and other desirable features such as JavaScript support and the ability to generate interactive charts in SVG format.

3.1 Exposing the Web-CAT Data Store

As described earlier, Web-CAT collects a large amount of data that would be useful to compile into a report. A database contains information about assignments such as their names, descriptions, due dates and scoring guidelines. Statistics computed during the evaluation of a student's submission by grading plug-ins are written to a file that holds key/value pairs for properties such as code size, code coverage, execution time, test results, run-time errors, and static analysis results. Thus, it is not adequate simply to connect the reporting engine to the underlying SQL database using one of the standard BIRT data sources, because much of the information is not stored there.

Given this disparity in data storage, it is essential that we provide a uniform method for the reporting engine to query the data from Web-CAT. Further, since Web-CAT's plug-in-based architecture allows instructors to add their own plug-ins that collect additional data, a method with built-in extensibility is critical. The user designing the report should not have to be concerned that the data of interest comes from a database or from a field in a grading properties file. We accomplish this by providing our own extensions to BIRT that define a custom data model that accesses data in a Web-CAT-friendly manner. These extensions integrate fully with the BIRT report designer so that end-users can create report templates for Web-CAT as easily as they would for any conventional data source.

Web-CAT is built on top of Apple's WebObjects framework, which simplified the implementation. Each table in Web-CAT's database is shadowed by a Java class with accessor methods corresponding to the columns in the table. These classes can also be extended to compute dynamic properties as well. Then, values can be identified using a "key-value coding" notation, which uses dot-separated key paths to evaluate properties, similar to how JavaBeans properties are denoted. For example, *submission.user.userName* would call the `getUser()` method on the `submission` variable, and then the `getUserName()` method is called on that result. By adopting this notation when defining reports, we simplify the way that data is captured and reap the benefits from the WebObjects model, so that properties can be accessed in the same manner regardless of whether they are a database-backed property or one that is obtained from another source.

3.2 Extending BIRT to Communicate with Web-CAT

BIRT divides its data model into a two-tiered hierarchy. At the top level are “data sources.” BIRT itself includes support for a handful of data sources, ranging from XML and CSV files to JavaScript and JDBC connections. Properties associated with the data source might be the path to a data file (in the case of XML or CSV), or the server name and login credentials (for JDBC). Then, in a report template, each instance of a data source contains one or more “data sets.” A data set represents a query into that data source and defines the columns that will be retrieved for each row of the result set.

None of BIRT’s built-in data sources are appropriate for interfacing with the Web-CAT object model, so we have extended it by adding a Web-CAT data source and data sets. In this case, the data source is simply a representation of the instance of Web-CAT under which BIRT is running, and has no associated properties of its own. A Web-CAT data set has two principal properties. The first is the type of object in the Web-CAT object model that it expects to receive. Each object in this set will correspond to a row in the result set. Second, the data set defines a list of columns for the report. Each column has three parts: a symbolic name that will be used to refer to it elsewhere in the report, the data type of the column, and the key path to evaluate to retrieve the value for that column. These key paths are evaluated using objects of the data set’s object type as their root; in other words, if *x* is an object being fed into the report and *key.path* is one of the column key paths, then the result is the evaluation of *x.key.path*.

We do not limit the user to simple key paths, however. A column expression can be written in OGNL [1], which is syntactically a proper superset of standard key paths that provides additional features such as enhanced list operations and pseudo-lambda expressions. This added flexibility can be useful when it is necessary to perform additional small calculations, such as aggregations, for which a simple key path would be insufficient. In addition, BIRT uses the Rhino [9] implementation of JavaScript so that scripts can be written in a report to transform the data further as necessary.

3.3 Maintaining a Library of Report Templates

In order for users to gain the most benefit from the Web-CAT reporting engine, they should be encouraged to share the report templates that they create so that other instructors can use the templates to generate reports for their own classes. A similar model is currently in use for Web-CAT’s grading plug-ins, which can either be private to the user who created them or published for anyone’s use, and are annotated with metadata that provides human-readable names, descriptions, and parameter information.

We mimic this design by providing a template library to which users can upload the report templates they create. Rather than designing a report with a fixed course, assignment, or other query in mind, the templates are constructed so that they process a set of a particular type of objects in the Web-CAT object model (such as submissions); then, users can visualize data from a course or assignment of their choosing. When a report template is chosen for generation, Web-CAT will ask the user to specify a query that will be used to determine which objects are passed to the report.

3.4 The Life Cycle of a Report

A report on Web-CAT goes through three phases: the report template, the intermediate report document, and the final rendered report. The report template (called a “report design” by BIRT), as one would expect, contains the data set definitions, layout, charts, and tables that make up the report. Report templates can be produced by BIRT’s graphical report designer application. Once designed, the template can then be used by many instructors to create reports for different courses or assignments.

When an instructor requests a report using a given report template, BIRT generates an intermediate file that it refers to as the “report document.” This is an internal representation of the report that contains all of the data retrieved from the data source, but that is not yet rendered in a human-readable format. Our design caches these report documents internally for later use.

In the final phase, the intermediate document is rendered into one of several formats and presented to the user. These formats include HTML, PDF, and Microsoft Excel. Support also is provided to extract the data from a report into CSV files to be analyzed using an external tool if desired.

There are two motivations for caching the intermediate report document as opposed to running the generation and rendering phases as a single task. First, the rendering phase of the report typically requires only a negligible amount of time when compared to the generation phase, which can require seconds or even minutes to complete, depending on the nature of the data being processed. If the user wishes to re-render the report in a different format, either immediately after it is generated or later, starting from the cached report document greatly speeds up this process. Second, a report generated at a particular time should represent a snapshot of the data on Web-CAT at that time. If the report were regenerated each time it was viewed, it would diminish the usefulness of using the reports for comparative analysis.

4 Data Mining Applications

Prior to the implementation of the reporting engine, there were many questions regarding student performance and habits about which we could only make educated guesses based on anecdotal evidence. With the entire data store of Web-CAT now exposed, we are in a much better position to answer these questions with quantifiable evidence. In the sections below, we describe some of these questions, examine how the reporter can be used to mine data from an assignment, and discuss whether the results either reinforce or challenge our original assumptions. All of the reports shown below were generated with data mined from the same assignment in a single course unless indicated otherwise.

4.1 Number of Submissions versus Final Score

Do students who make more submissions receive higher scores than those who make fewer? Web-CAT allows students to make multiple submissions before the deadline, so that students can get feedback early and participate in more feedback/revise/resubmit

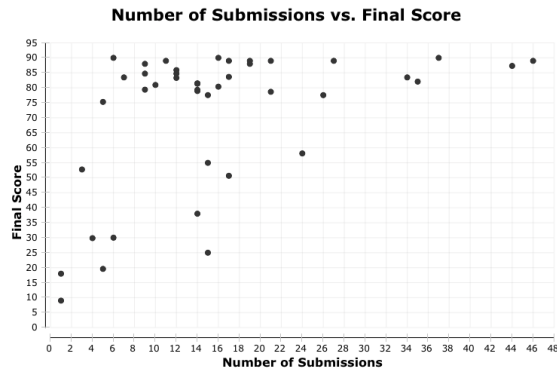


Figure 1.

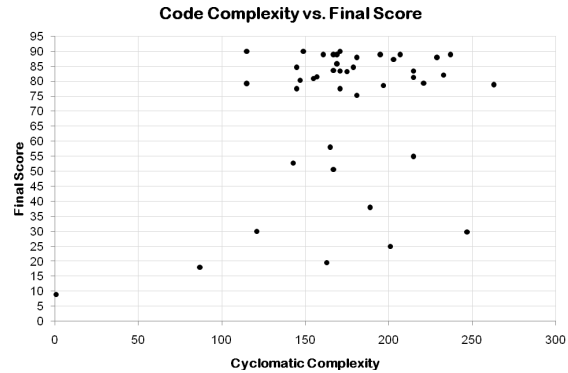


Figure 2.

cycles. Students who make few submissions could be either advanced students who quickly arrive at the correct solution, or poorer students who start late and give up quickly. Students who make an excessive number of submissions may not be thinking about the problem critically, and may just be spinning their wheels.

Figure 1 shows a scatter plot where the horizontal axis is the total number of submissions that a student made and the vertical axis is the final score that each student received on the last submission. Students who made four or fewer submissions scored poorly. Beyond that, grades improved dramatically. Those who made many more submissions were also able to achieve a high score with which they were satisfied. There is a clear partitioning of the student scores at 71, which separates students who passed the assignment (a letter grade of C or better) and those who failed. A chi-square test of number of submissions between the two groups indicates a significant difference ($df = 1$, $\chi^2 = 7.6$, $p = 0.006$, $\alpha = 0.05$), with students failing the assignment making fewer submissions (a mean of 9.5, compared with 18.3 for students who passed).

4.2 Code Complexity versus Final Score

Is there a correlation between the complexity of a student’s submission and the score received? Here, we have used McCabe’s cyclomatic complexity measure [7], although other measures are possible. It is even possible for instructors to devise their own measures and write a Web-CAT plug-in to collect custom data as part of the grading process, which then becomes directly accessible in custom reports.

We hypothesize that a graph of complexity vs. score would resemble a bell curve, where low-complexity submissions indicate a student failed to write an adequate solution, while excessively complex solutions indicate a student who may have written too much code without a proper understanding of the problem, perhaps using a “band-aid” approach.

Figure 2 shows a scatter plot where the horizontal axis is the cyclomatic complexity of the student’s final submission and the vertical axis is the final score. Although outliers on both ends tended to score poorly, no clear bell shape emerges and low-scoring submissions are distributed throughout the range. A chi-square test fails to show any significant difference between students who achieved passing scores and those that did

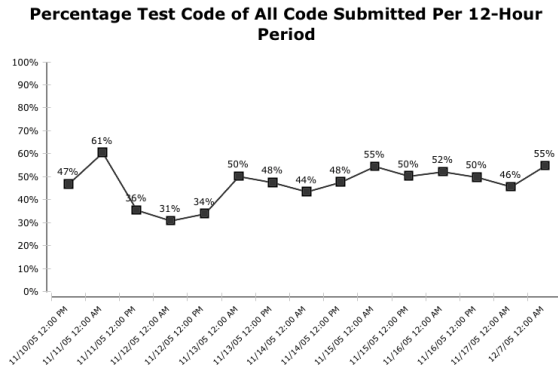


Figure 3.

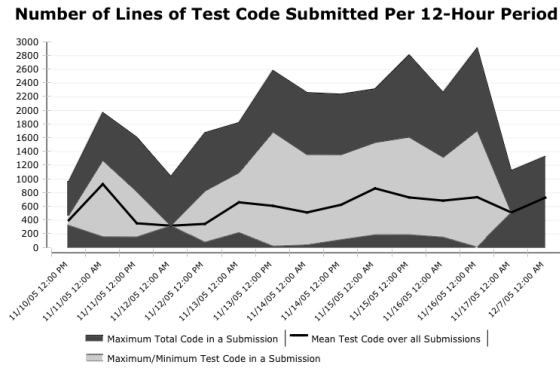


Figure 4.

not. This result suggests re-running the report over multiple assignments—possibly over multiple semesters—might provide a better visualization of any trend that is present.

4.3 Early versus Late Testing: Student Habits

When do students begin writing unit tests for the components in their solutions—incrementally as they write the code, or only after writing the majority of their code? This question interests us because we require students to test thoroughly the code that they write in our CS1 and CS2 courses. To answer this, we can examine the submissions made on each day as an assignment deadline approaches and examine the total number of lines of test code among them versus lines of non-test code.

Figures 3 and 4 show the results. Figure 3 is a line graph that shows, for all submissions occurring in each 12-hour period, the percentage of all code submitted that was unit test code. Figure 4 is a more detailed breakdown of this same information, displaying the minimum, maximum, and average number of lines of test code submitted in each 12-hour block, as well as the maximum number of lines of all code submitted for comparison. Although there is a slight dip approximately five days before the assignment is due—caused by one student, as indicated in Figure 4—these two plots give a strong indication that students overall are testing their work incrementally, rather than waiting until they have finished their solutions before beginning to write their own tests.

We can also examine assignments at the beginning and end of a course offering to see if testing habits improved as time went on. The chart in Figure 3 reflects the final assignment in a course. Figure 5 shows a similar chart for the first assignment given in the same course. The plot for the first assignment indicates that students were writing tests incrementally at the start of the course, as the instructors intended.

4.4 Early versus Late Testing: Affects on Final Score

Do students who begin testing their code early perform better on the assignment overall than those who put off testing until much later? This question arises when instructors consider encouraging students to use test-driven development (TDD), a strategy where they incrementally write tests along with their solution, writing each test just before completing the corresponding part of their implementation. Anecdotal evidence might

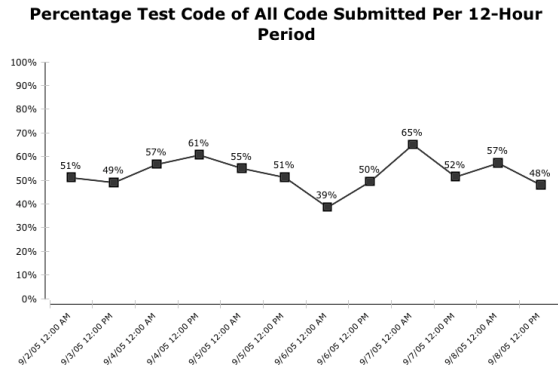


Figure 5.

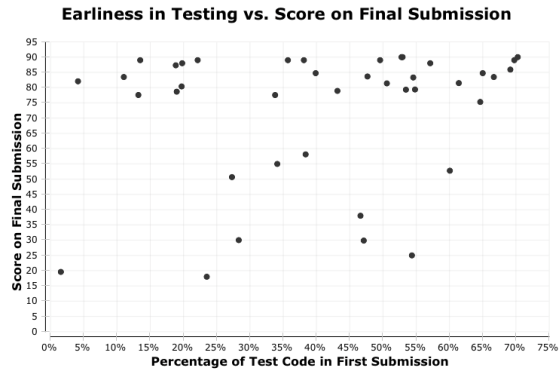


Figure 6.

suggest that students score higher if they test early, and experimental evidence indicates that if they are graded on how well they test, they produce significantly fewer bugs and score higher [3].

To answer this question, we first need to be able to quantify how early a student began writing tests. For simplicity, we will define this as the percentage of test code in their first submission. We rationalize this choice by stating that students should be doing their own testing as they write their solution piece by piece, even before having a sufficiently complete attempt to submit for grading. Students who do not start writing tests early will have a low percentage, while students who do should have a higher one.

A number of other more detailed metrics could be explored instead. Rather than examining the percentage of test code, one could combine this in some fashion with the percentage of test code that was actually executed according to code coverage tools, the degree of code coverage achieved by executing the tests, or a time-based approach determining when, if at all, a student passes a pre-defined threshold of testing.

Figure 6 shows a scatter plot where the horizontal axis is the percentage of test code in each student’s first submission and the vertical axis is the score received on his or her final submission. Because the earliness of testing is based on the percentage of test code found, *higher* numbers toward the right indicate *earlier* testing.

A chi-square test between students who achieved passing scores and those that did not fails to show any significant difference in the percentage of test code in the first submission. However, the plot shows an interesting gap in the percentage of test code around 25-30%. Indeed, using the likelihood-ratio chi-square to produce an optimum partitioning by percentage of test code leads to a split point of 25%. We can consider students below this threshold as those who “test less” early, and those above the threshold as those who “test more” early. One possible explanation is that students with greater programming experience—who also are more resistant to test-first development practices, and end up in the “test less” group—manage to do well on assignments at this level. Another is that students who did little testing early quickly learned that it was to their advantage to do more and improved on this before their final submission. We could delve further into either theory by generating another report with the progression of

percentage of test code from the first submission to final submission, breaking it down by students who started with little testing vs. students who started with more.

5 Future Work

We realize that many instructors would prefer to make use of an existing set of report templates rather than undertake the task of creating their own. While we cannot predict the exact needs of everyone who uses Web-CAT, it behooves us as the system designers to develop as comprehensive a library of templates as possible so that other instructors can immediately benefit from the system. As is the case with Web-CAT's grading plugins, we can make use of Web-CAT's auto-updating feature to distribute new or revised report templates to users as we would distribute system component updates.

At the same time, many of the reports discussed in Section 4 required significant JavaScript code and OGNL expressions to be written into the data set descriptions. BIRT provides these tools for creating reports that are more intelligent, and they were used here to transform the data in a way that was appropriate for the view that we desired. However, using these features necessarily increases the learning curve for new users. We can simplify this either by extending the Web-CAT data model as deemed necessary, or by providing utility code to assist with common operations and transformations, either in the form of JavaScript "snippets" that can be dropped into a report or server-side Java code that can be called from within a report.

By doing this, common data mining techniques such as regression, classification with decision-trees, and others could be embedded so that a user could integrate their results into a report more easily. Even without these more advanced features, however, it is easy under the current system to export various cuts and views of the Web-CAT data store into CSV files that could be analyzed by more specific data mining tools off-line.

Another topic that the reporting tool opens up is the possibility of doing *predictive analysis* based on the data mined from Web-CAT. Instructors could use data about past assignments to predict future performance, going as far as breaking down an assignment into specific problem areas and identifying students who had particular trouble with certain concepts. These ideas would benefit further from the ability to set up a report to be automatically generated on a recurring schedule. Trends in data could be predicted in one run of the report and then compared to the actual outcome of the next run. E-mail notifications to instructors or even to students could be used to send alerts of at-risk situations, while students are still developing their solutions before an assignment is due.

6 Conclusions

This paper has presented a solution to the issue of accessing and analyzing the large amount of data that is generated from an automated grading system. We have discussed the design and integration of a report generator with an existing automated testing system, Web-CAT, and shown how complex reports can be used to mine data from student coursework to answer deep questions about their performance and habits.

As of this writing, the implementation of the reporting engine was only recently completed. As such, no formal evaluation of the system has yet been undertaken, but the potential for this type of system should be apparent. Instructors and computer science researchers now have the ability to both generate simple grade reports and distributions, as well as perform empirical analysis on their coursework to answer questions as complicated as those described above. The answers to these types of questions may aid an instructor in improving their curriculum.

Due to the flexibility of both Web-CAT and BIRT, there are also potential opportunities outside the realm of computer science education. Web-CAT is in essence a large plug-in manager and integrating BIRT extends that idea to include report templates. Plug-ins can be devised that would allow for any type of analysis on code, and the results of that process could then be rendered for study. An example might be an in-depth study of how a department in a corporation complies with the company's coding standards. An evaluation tool of this sort would be beneficial in any case where statistical data is desired out of a large collection of source code. Further, applying these ideas to general-purpose course management systems could lead to similar capabilities for other disciplines.

References

- [1] Blanshard, L. and Davidson, D. *OGNL: Object-Graph Navigation Language*, 2008. Available at: <<http://www.ognl.org>>
- [2] The Eclipse Foundation. *BIRT Project: Business Intelligence and Reporting Tools*, 2008. Available at: <<http://www.eclipse.org/birt/>>
- [3] Edwards, S. H. Improving student performance by evaluating how well students test their own programs, *Journal of Educational Resources in Computing*, 3(3): 1-24 (2003).
- [4] Edwards, S.H. *Web-CAT Wiki*, 2008. Available at: <<http://web-cat.org>>
- [5] Edwards, S. H., M. Pérez-Quiñones, M. Phillips and J. RajKumar. Graphing Performance on Programming Assignments to Improve Student Understanding, *Proceedings of the International Conference on Engineering Education*, 2006.
- [6] JasperSoft Corporation. *JasperReports*, 2008. Available at: <<http://jasperreports.sourceforge.net/>>
- [7] McCabe, T. A Complexity Measure, *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, 1976, pp. 308-320.
- [8] Mierle, K., K. Laven, S. Roweis and G. Wilson. Mining Student CVS Repositories for Performance Indicators, *2005 International Workshop on Mining Software Repositories*, ACM, 2005, pp. 1-5.
- [9] Mozilla.org. *Rhino: JavaScript for Java*, 2008. Available at: <<http://www.mozilla.org/rhino/>>
- [10] Object Refinery Limited. *JFreeReport*, 2008. Available at: <<http://www.jfree.org/jfreereport/index.php>>
- [11] Spacco, J., J. Strecker, D. Hovemeyer and W. Pugh. Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System, *2005 International Workshop on Mining Software Repositories*, ACM, 2005, pp. 1-5.