Continuously Collecting Software Development Event Data As Students Program

Joseph Abraham Luke

Thesis submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
In
Computer Science and Applications

Stephen H. Edwards, Chair
Clifford A. Shaffer
Manuel A. Pérez-Quiñones

April 27th, 2015
Blacksburg, VA

Keywords: Data Collection, Software Engineering, Computer Science Education

Continuously Collecting Software Development Event Data as Students Program

Joseph Luke

Abstract

Teaching good software development practices is difficult, both in theory and in practice. Time management and project organization are skills often left by the wayside by students too focused on the coding itself. Educational research has been invested in developing strategies to combat these bad habits. In order to provide better support for interventions discouraging bad development habits, more data about student development is needed.

The purpose of this research is to design and implement software to collect data continuously as students work on programming projects and provide it in useful forms to instructors and researchers so that they may make headway in designing new curricula, assignments, and interventions that better help students to succeed.

The DevEventTracker is a software system that interfaces with existing Web-CAT services to track student development data continuously, without any student effort. Development and compilation events are tracked within the Eclipse IDE through a plugin and sent to a Web-CAT server. Code snapshots corresponding to each event are also committed to a server-side repository.

The system provides a dashboard as a set of instructor-visible web pages that display useful data in generated charts and tables. Data are presented in both class overview and individual student summaries.

The system presented will enable future research in education and specifically in intervention development. Particularly, the system can be used to allow instructors to identify students who have a tendency to procrastinate and design more effective interventions.

Acknowledgements

I would like to thank all of those who supported me in completing this work:

My advisor, Dr. Stephen Edwards, for providing help, motivation, and deadlines that enabled me to complete my work and degree.

My committee members, Dr. Cliff Shaffer and Dr. Manuel Pérez-Quiñones, for providing guidance and advice in directing my work and writing my thesis.

My family, for their endless love and support.

My very dear friend, Mariëtte, for her continual support and companionship on the walks that help keep me sane.

This work is supported in part by the National Science Foundation under grant DUE-1245334. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

Table of Figures

Chapter 1: Introduction

1.1 Motivation

Teaching good software development practices is difficult, both in theory and in practice. Students often focus primarily on the coding aspects of software development, potentially ignoring other important skills such as time management and project organization that should also be developed when learning software engineering practices [4]. Procrastination is also a serious issue in computer science teaching, as it is in education in general. Procrastination prevents students from reaching their potential in both academic success and in learning important material and skills [3].

Addressing these problems depends first on identifying suboptimal practices, and then developing interventions to combat them. Edwards et al. [3] report experiences on interventions intended to address procrastination in computer science courses, and this research aims to extend the possibilities of these and other interventions. Analysis for interventions is currently based on student-reported data (e.g. schedules and time sheets) as well as their official submissions for grading on particular assignments. However, this approach does not generate enough information on when and how students actually perform their work to accurately measure behaviors like procrastination and practices such as test-driven development. The intention of this research is to provide more exhaustive and continuous data collection in students' programming development environments in order to determine the presence of bad work habits so that interventions may be applied sooner and developed more effectively.

To these ends, this research focuses on developing a new piece of software, named DevEventTracker, that collects compilation and development events and snapshots of code from students as they work and then gives instructors and researchers the tools to analyze this data for future use.

1.2 Problem Statement

To combat procrastination and other bad work habits, especially in software development, it is important to understand exactly what students are doing while working on assignments and when they are (or aren't) working.

When a student sits down to start a programming assignment in a computer science course, he or she may do some planning of program structure or time management, or none at all. Information about this planning stage is some of the hardest to gather, but interventions such as required schedule sheets at the beginning of an assignment can, when faithfully completed, help instructors determine whether students are planning out the priority of and time commitments to different pieces of an assignment. Whether or not the student conducts any planning, they will next start coding. This is the step where data collection is usually absent, despite its obvious value. Regardless of whether the student is following good development practices or not, the existing methods for collecting data are focused on specific points when the student submits work to the instructor or automated grading system. With automated systems, students may submit many times, but we still do not have a good idea of what pieces of the assignment they are working on and when. It would be useful to collect data, then, that fills in the gaps of our knowledge and allows us to infer whether students are following good practices or not. Specific examples including measuring how the student distributes his or her time to detect procrastination and examining test and solution code amounts to determine whether test-driven development is being employed.

Information about students' work habits can also be used for many other potential research applications, including designing and implementing new courses and assignments as well as determining predictors of future student performance. To address these issues, it is imperative to collect and interpret much more continuous data than is currently commonplace.

The objective of this research is to approach the problems listed above by developing software to collect continuous data relating to student work as they are programming. Without introducing extra overhead or time spent on the students' part, these data can be used to provide instructors and researchers with more insight into how exactly individual students and entire classes work. This data can then be used to develop more effective interventions and enable future educational research.

1.3 Solution Approach

Software was designed and implemented to support tracking of student development in the Eclipse Integrated Development Environment (IDE).

The software developed consists of three parts:

1. Event tracking
2. Web-CAT subsystem
3. Web-based dashboard

The software interfaces with existing pieces of the Virginia Tech software system Web-CAT (Web-based Center for Automated Testing).

The event tracking aspect of the software responds to events in the IDE such as compilation and file saves by uploading a record of these events to a server-side database. This is built into the existing Eclipse Web-CAT submission plugin. Source code is also stored into a local repository. This additional functionality operates in the background of students' IDEs, adding no additional setup or steps to follow. No separate installation is necessary, since these features are bundled into an existing Eclipse plugin that is already required by most courses using Web-CAT and Eclipse.

The second piece of the system is a new Web-CAT subsystem that accepts events sent from the plugin and stores them into a server-side database. It also accepts push requests from the local repository to store snapshots of student code on a server-side repository for future reference. This subsystem also computes statistics and charts for the final piece of the system, the dashboard.

The third piece of software, the dashboard, is accessed by instructors from a webpage hosted on the Web-CAT server. It enables them to see charts and tables showing useful views of the data being collected. The dashboard uses data harvested from both events and code snapshots, and will have one view for an entire class and a separate view for each individual student, with each student's corresponding page accessible from the class view.

The main motivation of this development is to help identify and rectify problematic student development behaviors such as procrastination. Instructors shall be able to use the developed dashboard to see statistics of how their classes and individual students are performing on particular assignments over time. The charts and tables presented will allow identification of underperforming students and general trends through semesters.

1.4 Organization of Chapters

There have been several efforts to provide similar development-tracking tools, both through events and code snapshots. These will be discussed in Chapter 2, along with the lessons learned from prior research efforts in this direction. Chapter 3 discusses design and implementation of the software, broken down into the Eclipse plugin and sensors first, repository collection second, and the dashboard software third. Chapter 4 presents a user guide for the overall system, including visualization of user-facing features. Chapter 5 presents conclusions and opportunities for future work.

Chapter 2: Related Work

2.1 Web-CAT

The Web-based Center for Automated Testing (Web-CAT), initially developed at Virginia Tech in the early 2000s, provides a submission and testing system for student code. An open-source project that continues to be updated, it allows institutions to set up a class and assignment system on a server and allow users to submit code. Intended to promote test-driven development, Web-CAT runs this code against instructor-provided reference tests as well as the unit tests students include. These results, along with coverage of how much student solution code is executed by all the given student test cases, are provided to the student, as are style analyses (including commenting and formatting). Multiple submissions are allowed and tracked, and instructors are able to review submitted code and edit and finalize scores [15].

With its large presence as a submission system for many institutions worldwide, Web-CAT provides a strong platform for continuous data collection. With a server-client model and an internal structure based around subsystems for generating and delivering content as well as processing data, Web-CAT is an ideal candidate for an extension to continuously collect data and process it, providing information to instructors in much the same way their classes' scores already are. An existing submission plugin for the Eclipse integrated development environment (IDE) provides the basis for the client piece of our model, providing connection and authentication to the server as well as a piece of technology that need only be updated, rather than freshly installed, in order for data collection to start.

2.2 Data Tracking Systems

Several projects in the last decade or so have focused on capturing and analyzing event streams and code snapshots in order to promote both student education and industry best practices. While planning for our project, we looked into various options discussed below for starting points, ideas, and code reuse where possible.

Hackystat is an open-source project started at the University of Hawaii in 2001 designed to provide product and process measurements in a variety of software engineering situations, from industry to education. Hackystat consists of a variety of sensors integrated

into the user's environment of choice that trigger upon specified actions. Sensors, as used in Hackystat and throughout this paper, refer to pieces of software hooked into a development environment that listen for specific changes and record information about the event when triggered. For instance, there are sensors for detecting changes in the current file, detecting the results of unit tests, and recording results of compilation and debugging. This data is automatically sent (when a connection to the central server is present) to a server and stored for analysis. It is saved in log form for later transmission when no connection to the server can be established. Hackystat has support for multi-user projects, and provides users with a constantly-updated analysis dashboard on the web. Users can log on and see all of their collected data from each project they are involved in, as well as useful charts, tables, and other analysis [7, 8, 9].

Marmoset is an automated grading system developed at the University of Maryland similar to Web-CAT. Marmoset collects student code using an Eclipse plugin, and stores it in a Concurrent Versioning System (CVS) repository each time a file is saved. Although this feature is only supported for single-user projects, it provides an automatic backup to students and allows researchers to conduct experiments on changes between versions of code, including how many errors and warnings are introduced, alleviated, or remain from one snapshot to another. Snapshots are also grouped into "work sessions" in order to determine approximately when students performed their work. However, analysis is conducted separately from data collection, and there is no automated analysis done on individual snapshots or their differences [16, 17, 18, 19, 20].

In Jadud's work with novice programmers [5], the BlueJ environment was used to capture data about their compilation habits. Snapshots were automatically taken of students code at each compilation, successful or not, and then parsed into tables containing information about all errors that occurred. Snapshots were organized into sessions based on a heuristic of seven or more compilation events followed by quitting the program. He also developed some tools that allow researchers better insight into student behavior: a code browser to view successive snapshots of a particular program, a visualization for types and amounts of various syntax errors, and an algorithm to score sessions based on student error rate and improvement. The algorithm is interesting, and measures whether each successive compilation ends in the same error, a different error, or no errors. Although

intended to determine if novice programmers are grasping syntactic concepts, it could likely be adapted to more complex situations such as we describe here. In a later paper [6], Jadud adds in support for interaction events, relating to the ability in BlueJ to create and manipulate objects on a virtual workbench corresponding to the objects described by the classes being written in Java. While this approach does not apply directly to the Eclipse IDE, these kinds of interaction events are potentially useful to determine if students are having trouble using or understanding the software, especially with novice programmers.

Clockit [12] collects data from the BlueJ IDE that both stores and displays data with two separate tools. With the data logger, the plugin records events corresponding to file manipulation, editing, invocation, and compilation events and stores them to a local log file. Data is uploaded to the project's server upon program close. The plugin also maintains a data visualization interface within the plugin as feedback to the student, summarizing activity sessions through types of events and their frequencies. Instructors can also access data from individuals or classes through a web interface that provides summary information on compilation, build errors, project growth, and time spent.

Other research on BlueJ programming has been conducted on a much larger scale through the Blackbox project based at the University of Kent. Run by the developers of BlueJ, this project provides opt-in, anonymous collection of data from hundreds of thousands of users, the vast majority of them novice programmers. All users of BlueJ are presented with a popup on first use and an option to change later, on whether they would like their data to be made available to researchers. For classroom research purposes, instructors may also provide their class with an identifier to enter which they may then use to access their class's data. The large-scale data collected has been used to verify and expand work done on smaller data sets in education research, such as Jadud's results on types of compilation errors committed by novice programmers [1, 2, 10].

2.3 Lessons Learned

From previous data tracking approaches discussed above, there are many useful concepts useful for our application. Specifically, Hackystat provides a clear architecture that can be leveraged into Web-CAT's existing system to collect most of the needed data. By reusing Hackystat's preexisting sensors and client-side connection protocols, much of

the development focus becomes on integrating these pieces into the Web-CAT Eclipse submission plugin and developing appropriate receiving code on the Web-CAT server to handle and store events. However, since it is apparent that event data are not enough to capture the entirety of how students work, the idea of maintaining repositories of student work from Marmoset's plugin is a welcome addition. From several projects above, including Hackystat and Clockit, data analysis and visualization is an important piece of utilizing these collection mechanisms, and our dashboard aims to capture these in a single webpage.

Chapter 3: Design and Implementation

3.1 DevEventTracker Architecture Overview

DevEventTracker is designed and implemented as three parts that each interface with existing Web-CAT systems, as seen in Figure 3.1. The first piece is an Eclipse IDE plugin that captures events and snapshots of code. The plugin, a modified version of the existing Web-CAT submission plugin, is already required for students using the system. The second architectural component is a new Web-CAT subsystem containing the server-side code to handle storing and processing events and snapshots, using an extension of the existing Web-CAT user database and a new set of Git repositories. The last piece of the system is the dashboard, the instructor-oriented user interface provided by the DevEventTracker subsystem on Web-CAT, used to allow instructors to view charts and tables generated from data tracked per class and student over the lifetime of an assignment.

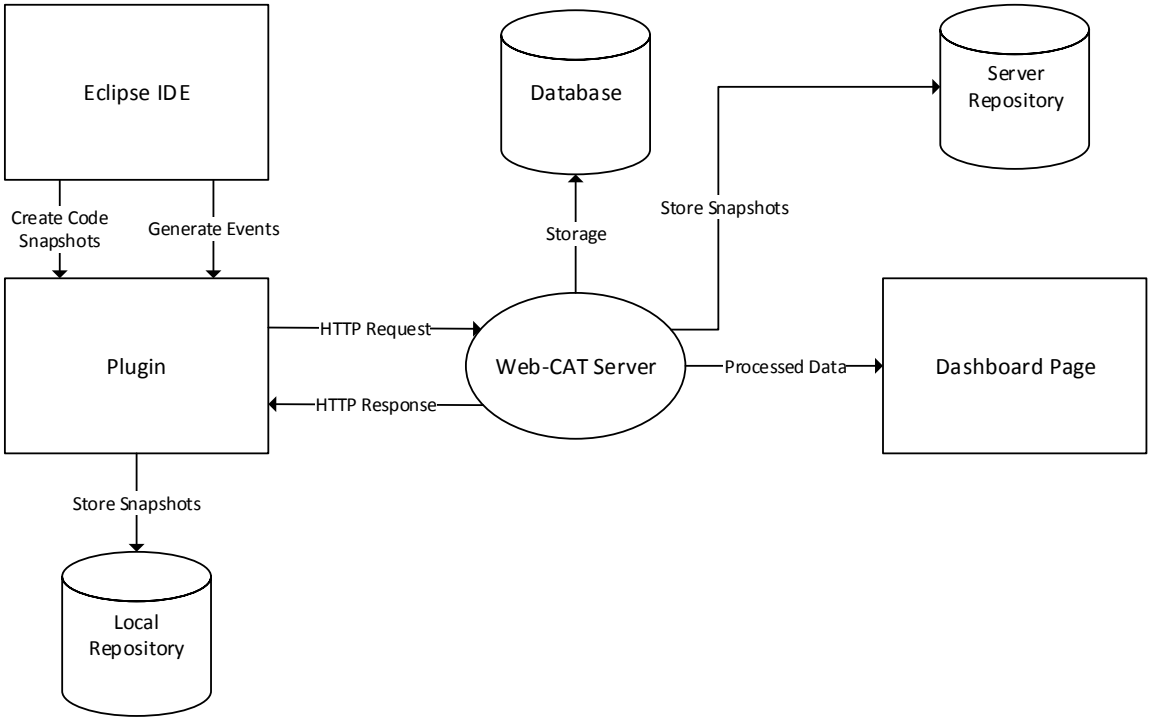


Figure 3.1 Architectural overview of the DevEventTracker

3.2 Eclipse Plugin

3.2.1 Goals and Objectives of the Plugin

The goals envisioned for the Eclipse plugin are:

1. Work as part of the existing Web-CAT submission plugin, requiring no extra effort to install or set up.
2. Provide sensors to detect important compilation and development events.
3. Send sensor data to Web-CAT server upon generation if a server connection is present.
4. Maintain files containing records of events when no server connection is present, and send them to the server when a connection becomes available.
5. Maintain a local repository of the student's code base.
6. Push the local repository to the server-side repository when certain events are generated if a server connection is present, and synchronize them when one becomes available.

3.2.2 Design Decisions

Design of the plugin portion of the project began with deciding which type of data collection paradigm to use. After looking at the various solutions in the projects detailed in Chapter 2, we decided to use an approach based on Hackystat. This was based on the similarity of Hackystat's Eclipse plugin design to the existing Web-CAT submission plugin and the potential for reusing the *sensors* implemented as part of the Hackystat project. Hackystat sensors are small software modules which listen for particular events (such as compilation success, file save, etc.) and trigger recording of the circumstances of the event, including the time, the type of event, and the associated file. With this in mind, it was decided to reuse as much code as possible from the Hackystat project, including the sensors themselves and the client-side HTTP connection. However, several implementation-specific features, such as local database storage and Java Architecture for XML Binding (JAXB) generation of Java class structures, were removed to better match the existing architecture.

After work had begun on the plugin, it became apparent that more fine-grained data would be necessary to achieve some of the aims for the overall project. Although the

initial plans did not include any repository storage, this was added to the system to capture source code changes made by students in addition to other event data. In particular, the events being collected do not provide enough information on their own to obtain the majority of software metrics, including things like cyclomatic complexity, velocity, and even simply lines of code. As a result, it was decided to implement a system similar to Marmoset's CVS plugin, allowing collection of code snapshots through Eclipse.

As with any data collection, privacy is a major concern. Since our work will be used for class improvement, which is not subject to required consent, data will be collected automatically and recorded on the Web-CAT server for all users of the system. A notification is shown on the first startup of Eclipse with the plugin installed in order to notify users that their data is being collected. However, in order for the collected data to be used for research purposes, researchers would be obligated to follow accepted practices to protect research subjects, and would need to use a separate mechanism to obtain informed consent from students where necessary.

3.2.3 Implementation Details

The plugin itself records both compilation and development events. Specific events tracked are compilation success and failure, file open, close, edit, and save, and editing/addition of program units (e.g. import statements, methods, classes, and fields). The time and type of event (compilation/development) and the associated file's Uniform Resource Identifier (URI) are recorded and sent to the server immediately if possible, or written to a file for uploading once a connection is established.

In addition to event data, capturing the contents of student files is also important. Marmoset [16] is a prior system that uses an Eclipse plugin to capture student file changes using a centralized version control system, checking in each student change as a separate snapshot. Inspired by Marmoset, the DevEventTracker system also captures student source code changes as they happen by checking in snapshots using a version control system. One major difference between Marmoset's code snapshot system and the one provided here is the underlying repository system. While Marmoset utilized a CVS system, the DevEventTracker uses Git repositories through JGit, a Java Git implementation with programmatic support for all necessary repository actions. Git

was chosen as it is already used with existing subsystems within Web-CAT. Each time an event is generated as a result of a file being saved, a commit is made to the local repository, and then is pushed to the server-side repository if possible.

3.3 Web-CAT Subsystem

3.3.1 Goals and Objectives of the Web-CAT Subsystem

The goals envisioned for the server-side piece of the system are:

1. Respond to HTTP requests from the plugin for retrieval of UUID data.
2. Receive HTTP requests from the plugin passing in data.
3. Maintain a database of all events recorded by the plugin and sent to the server.
4. Maintain a link on the server between each sensor event recorded and the associated user and assignment.
5. Maintain a server-side repository of the student's code base.
6. Maintain a link on the server between the repository and the associated user and assignment.
7. Maintain a link on the server between each commit and the associated event.

3.3.2 Design Decisions

Code for the server side of the system was written in a new WebObjects subsystem dedicated to handling HTTP requests, storing data, and generating and serving the dashboard page (discussed in Section 3.4).

An issue that was encountered during design concerns the automatic linking of multiple users on a group project. When multiple users are allowed to work together, as is common on large software projects, it seems reasonable to maintain links between their work to enable more detailed analysis and comparisons with their interim and final submissions. The easiest way to do this involves automatic parsing of the Eclipse projects to determine if multiple users are editing the same project (i.e. if two events are passed in with the same project UUID but different user UUIDs) and grouping them together if they are. Since the project UUID is stored in the local file system of the project after being passed back from the server, this indicates that the directory structure has been copied or shared in some way. However, we realized that grouping users together automatically as partners if this pattern existed was problematic. It is not

known without an official submission to Web-CAT (when students are required to list their partners, if any), whether a student is working with any other. As a result, we have decided to only include a connection between users working together in our database when we are sure one exists; i.e. after we receive an official submission with all group members' names included.

3.3.3 Implementation Details

The existing database used for users and assignments on the Web-CAT server was extended to support include the entities and relationships shown in Figure 3.2. The underlying repository system used on the server mirrors that in the plugin, a JGit implementation similar to the one already used in other Web-CAT subsystems. The Git commit hash is stored with the save event generating the snapshot.

As shown in Figure 3.2, the main objects stored in the server-side database are Users, ProjectsForAssignment, SensorDatas, and StudentProjects. Users exist already within the Web-CAT framework, and are identified by their email address, but our project extends their definition to allow for connections the other entities as shown. When an event is recorded and submitted by the plugin, it is recorded as a SensorData instance. Each of these instances within a particular Eclipse project are grouped together as a StudentProject. Upon receipt of an event from a new Eclipse project, the server attempts to determine which assignment the Eclipse project is for, and groups it with other StudentProjects for the same User under one ProjectForAssignment. If a submission occurs and the system determines that two students are working together, their ProjectsForAssignment for that assignment are merged. Users and StudentProjects are tracked by universally unique identifiers (UUIDs) generated server-side and passed back to the plugin to maintain consistency across different Eclipse installations or different versions of the same project owned by the same User.

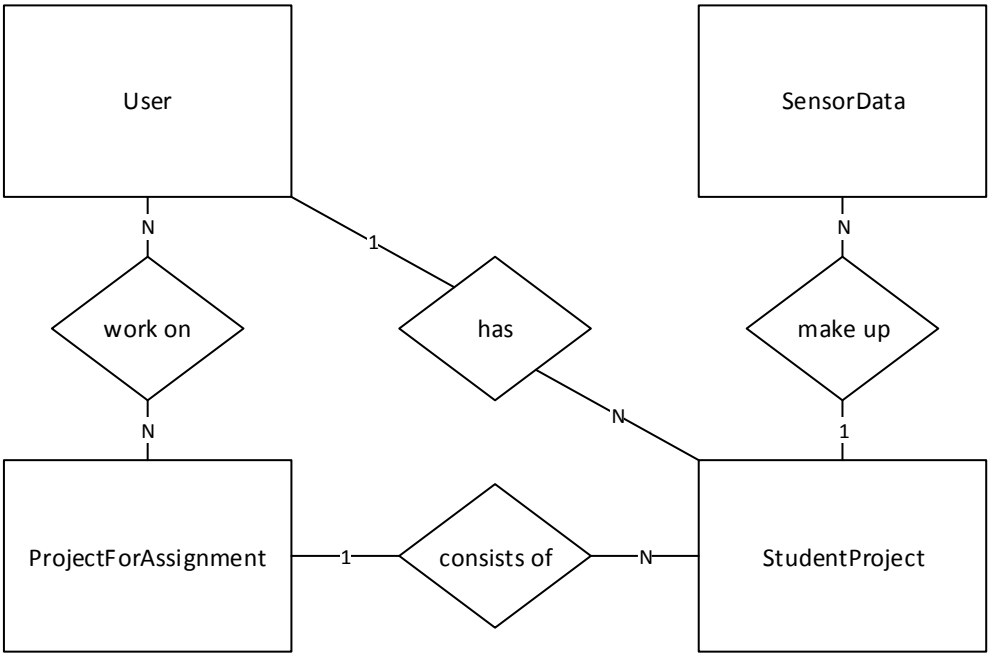


Figure 3.2 Simple entity-relationship chart for the DevEventTracker event database

3.3.4 Plugin-Server Interaction

In order to pass data related to event tracking to and from the server, the plugin uses an HTTP connection through a Restlet [14] client and the GET command to perform one of three actions:

1. *retrieveUser(email)* – given a user's email, the server will look up the UUID for the associated user, generate one if none exists, and then pass it back to the plugin.
2. *retrieveStudentProject(projectURI, userUUID)* – given an Eclipse project's Uniform Resource Identifier (URI) and the UUID of the user as mentioned above, the server will look up the UUID for the associated StudentProject on the server, generate one if the StudentProject has not been created, and then pass it back to the plugin.
3. *postSensorData(studentProjectUUID, userUUID, time, runtime, tool, SensorDataType, URI)* – given the appropriate UUIDs, the time of event submission, the runtime of the event, the name of the tool (always Eclipse in our case), the type of SensorData, and the URI of the file or folder affected by the event, this will cause the server to create the SensorData instance in its

database, populate it with the given data, and associate it with the correct entities.

For the local Git repositories, the local repository has a WebObjects adapter URL as its remote repository link. When a push command is given to the local repository, the plugin will attempt to push the repository to the server through the remote URL. Existing Web-CAT code handles the Git push request and other management of the Git repository on the server.

3.4 Dashboard

3.4.1 Goals and Objectives of the Dashboard

The goals envisioned for the dashboard system are:

1. Provide a webpage for instructors and researchers to see charts of various statistics generated class wide and viewed over time.
2. Provide a webpage for instructors and researchers to see charts of various statistics generated for a particular student and viewed over time.
3. Provide a summary table of all statistics for both the class-wide (most recent values for each student) and individual student (values over time for each statistic) pages.
4. Provide a link to an individual student's page when clicking on that student's row in the summary table on the class page.
5. Provide the two most useful charts at the top of the page with the option to display more at the user's discretion.
6. Merge into the existing Web-CAT menu system as another link seen by instructors but not students.

3.4.2 Design Decisions

The design of the dashboard was also influenced by the Hackystat project. Their Telemetry and DailyProjectData pages provide access to a variety of generated statistics from recorded events and allow the user to view both charts and the underlying raw data in tabular form.

3.4.3 Implementation Details

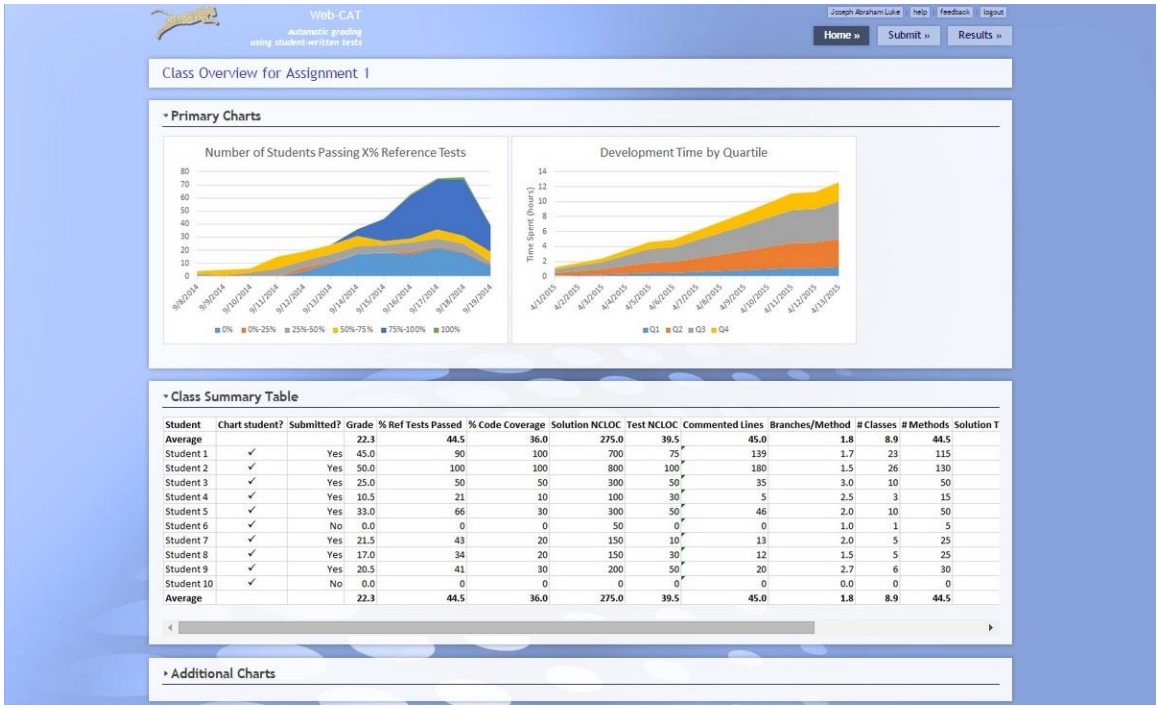
The dashboard is implemented as a Web-CAT-hosted webpage. Web-CAT webpages are generated through the WebObjects framework, with Java classes providing the data to fill in the HTML structure. The dashboard has two member pages, one for a class overview for a single assignment, and a second student-specific assignment overview. On each page, four charts are presented to the user along with a summary table and the option to select other charts to view. On the class overview page, the table is organized with one student per row and each student's most recent available data for each column. Columns represent different generated statistics about the assignment including grade, non-commented lines of code (NCLOC), submission status, and percentage of reference tests passed. This data is generated by looking at both Web-CAT submissions and events with their associated code snapshots. Charts on this page show information about the class's performance as a whole over time, mostly displaying averages, with a few exceptions as documented in Chapter 4. The student page has a summary table with the same columns, but with each row as a particular time point. Because some statistics are generated from data only available at submission and some through analysis of code snapshots and events and submissions are much rarer than event collection, some statistics may have empty cells.

Chapter 4: User Guide

As the student using the Eclipse plugin should not notice any difference in their Eclipse environment beyond the initial data collection notification, the vast majority of the user experience lies with the instructor using the dashboard portion of the DevEventTracker on the Web-CAT website. This presents a description and explanation for this dashboard webpage.

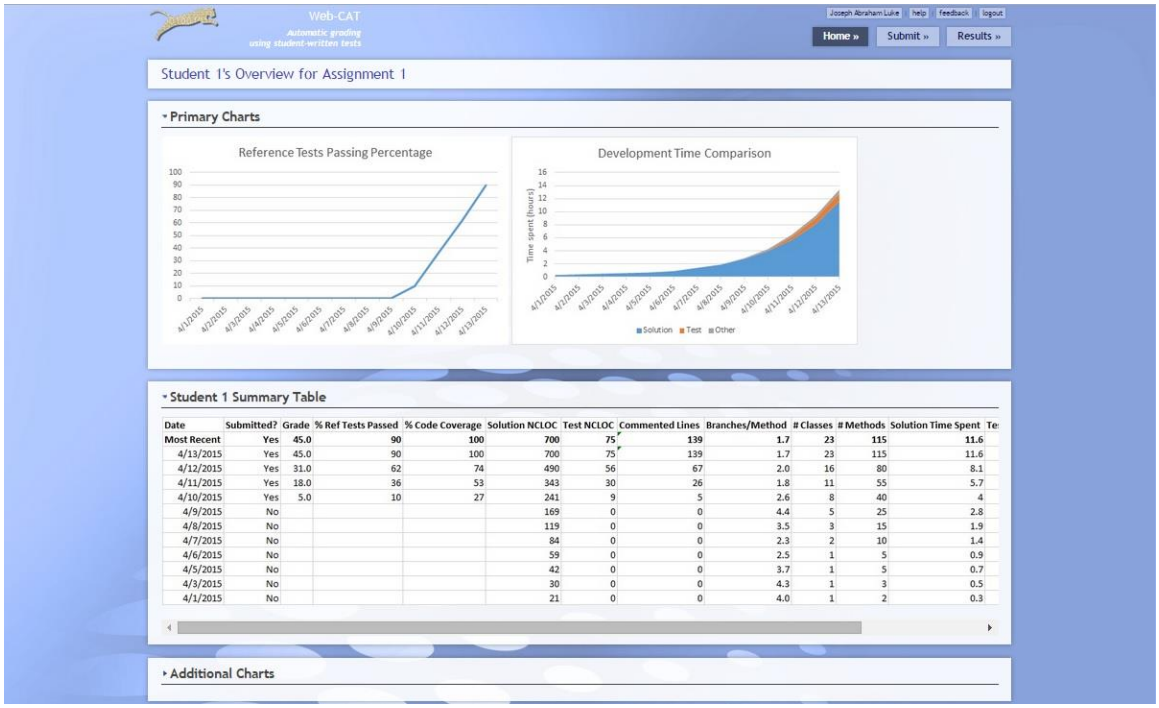
4.1 Dashboard Overview

Mockups of the dashboard are shown in Figure 4.1 (entire class view), and Figure 4.2 (individual student view). In each view, the dashboard displays two preselected charts at the top of the page, but below identifying information for the class, assignment, and student. These two charts aim to summarize the progress of the class or student, and are the first two charts covered in Section 4.2. They cover reference test passing percentage and development time. Directly below these initial charts is a summary table showing the most recent value for each statistic for each student in the class. In the class view, this table has selection checkboxes for each student so an instructor can view a specific subset of the class (e.g. those who have submitted at least once). Below this table are several other additional charts along with a selector for allowing the user to move more charts to the top of the page. The remainder of Section 4.2 after the first two subsections covers these additional charts. Values throughout the dashboard are calculated from the most recent data available. Submission data is used alongside data gleaned from the event stream and code snapshots if a student has made a submission to Web-CAT for the assignment.



Student	Chart student?	Submitted?	Grade	% Ref Tests Passed	% Code Coverage	Solution NCLOC	Test NCLOC	Commented Lines	Branches/Method	# Classes	# Methods	Solution T
Average			**22.3**	**44.5**	**36.0**	**275.0**	**39.5**	**45.0**	**1.8**	**8.9**	**44.5**	
Student 1	✓	Yes	45.0	90	100	700	75	139	1.7	23	115	
Student 2	✓	Yes	50.0	100	100	800	100	180	1.5	26	130	
Student 3	✓	Yes	25.0	50	50	300	50	35	3.0	10	50	
Student 4	✓	Yes	10.5	21	10	100	30	5	2.5	3	15	
Student 5	✓	Yes	33.0	66	30	300	50	46	2.0	10	50	
Student 6	✓	No	0.0	0	0	50	0	0	1.0	1	5	
Student 7	✓	Yes	21.5	43	20	150	10	13	2.0	5	25	
Student 8	✓	Yes	17.0	34	20	150	30	12	1.5	5	25	
Student 9	✓	Yes	20.5	41	30	200	50	20	2.7	6	30	
Student 10	✓	No	0.0	0	0	0	0	0	0.0	0	0	
Average			**22.3**	**44.5**	**36.0**	**275.0**	**39.5**	**45.0**	**1.8**	**8.9**	**44.5**	

Figure 4.1 Mockup of the class-view dashboard



Date	Submitted?	Grade	% Ref Tests Passed	% Code Coverage	Solution NCLOC	Test NCLOC	Commented Lines	Branches/Method	# Classes	# Methods	Solution Time Spent	Te
Most Recent	**Yes**	**45.0**	**90**	**100**	**700**	**75**	**139**	**1.7**	**23**	**115**	**11.6**	
4/13/2015	Yes	45.0	90	100	700	75	139	1.7	23	115	11.6	
4/12/2015	Yes	31.0	62	74	490	56	67	2.0	16	80	8.1	
4/11/2015	Yes	18.0	36	53	343	30	26	1.8	11	55	5.7	
4/10/2015	Yes	5.0	10	27	241	9	5	2.6	8	40	4	
4/9/2015	No				169	0	0	4.4	5	25	2.8	
4/8/2015	No				119	0	0	3.5	3	15	1.9	
4/7/2015	No				84	0	0	2.3	2	10	1.4	
4/6/2015	No				59	0	0	2.5	1	5	0.9	
4/5/2015	No				42	0	0	3.7	1	5	0.7	
4/3/2015	No				30	0	0	4.3	1	3	0.5	
4/1/2015	No				21	0	0	4.0	1	2	0.3	

Figure 4.2 Mockup of the student-view dashboard

4.2 Chart Descriptions

Charts in this section are displayed as the version to be shown in the class view except where otherwise noted. This means that the charts depict quartile views of class averages where the corresponding individual student chart would show only that student's values for each statistic. These charts were generated (with the exception of Figure 4.4) from data from a data structures and algorithms programming assignment in fall 2014. The vertical red line on each chart indicates the due date for the assignment.

4.2.1 Reference Test Pass Percentage Comparison

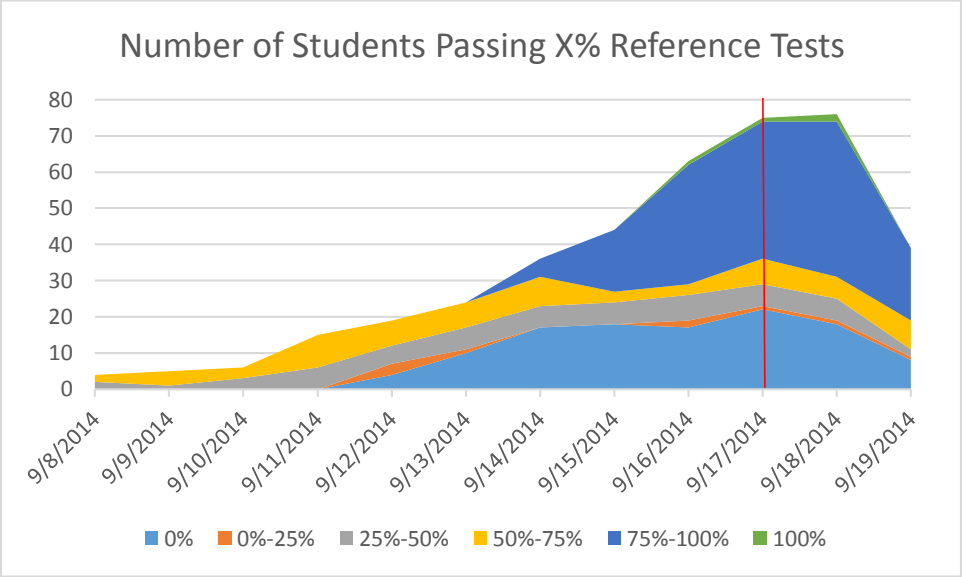


Figure 4.3 Reference test pass percentage comparison

Figure 4.3 shows the number of students who have submitted and passed some percentage of reference tests. Students who have submitted at least once are broken up into six categories: the four quartiles along with students passing exactly 0% and exactly 100% of the reference tests. This chart provides a quick view of how many students are submitting over time (the total height of the shaded area), as well as an indication on how the class is doing by quartile at each time step. This chart is shown only on the class-view page. It is replaced by a line chart showing reference-test passing percentage over time in the individual student view.

4.2.2 Development Time Comparison

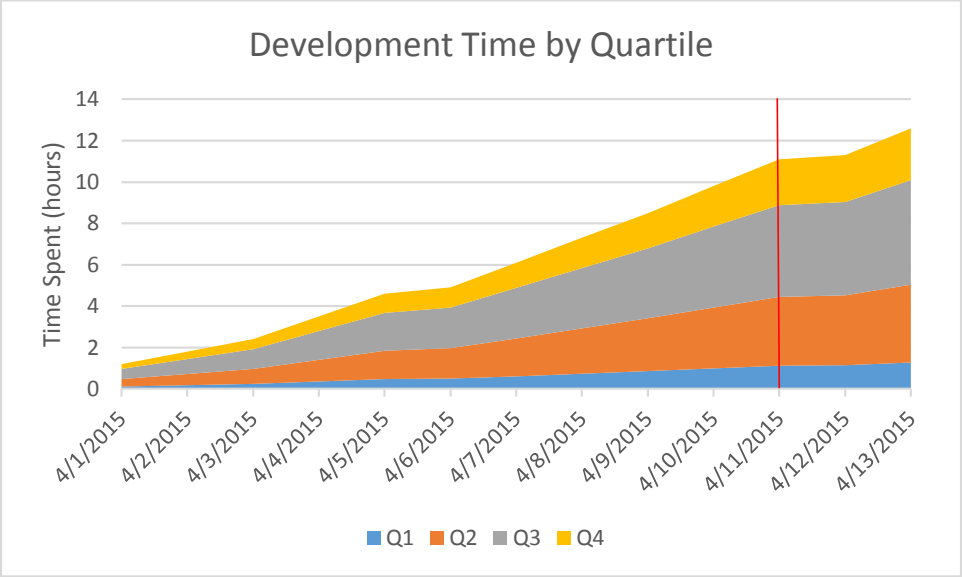


Figure 4.4 Development time comparison

Figure 4.4 shows a measurement of development time broken down by quartile. This chart is the only one in this section not created from real data, as a semester of data from the DevEventTracker is not yet available. This data allows for determining how student effort is distributed among the class. This also provides us information on how long different types of students take to finish their projects, which could prove useful when designing future assignments. In the individual student view, this chart instead shows a simple running total of student development time, but with the added breakdown of time spent on test code, solution code, and unidentified other time. This information does not fit well on the quartile chart that seems most useful for the class view.

4.2.3 Solution and Test NCLOC

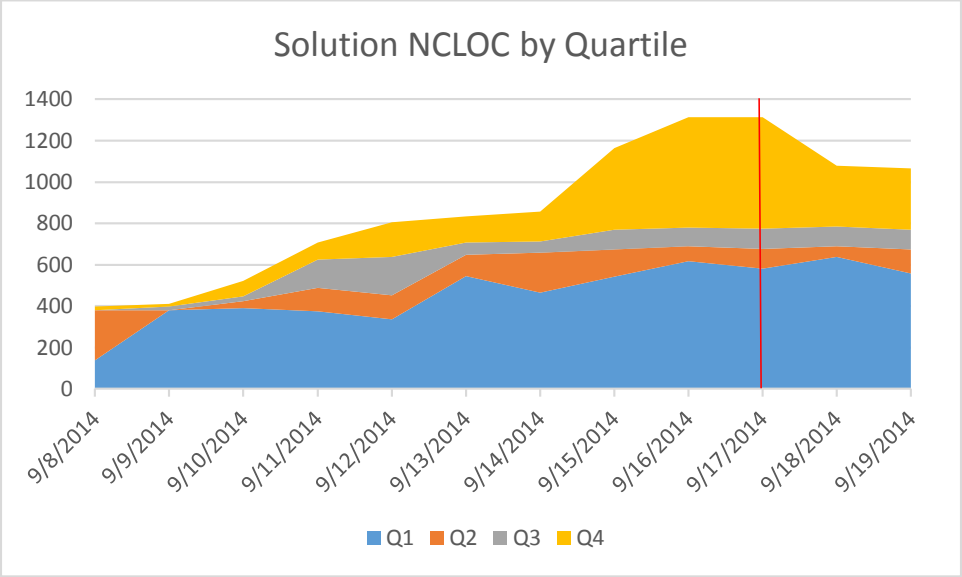


Figure 4.5 Solution NCLOC by quartile

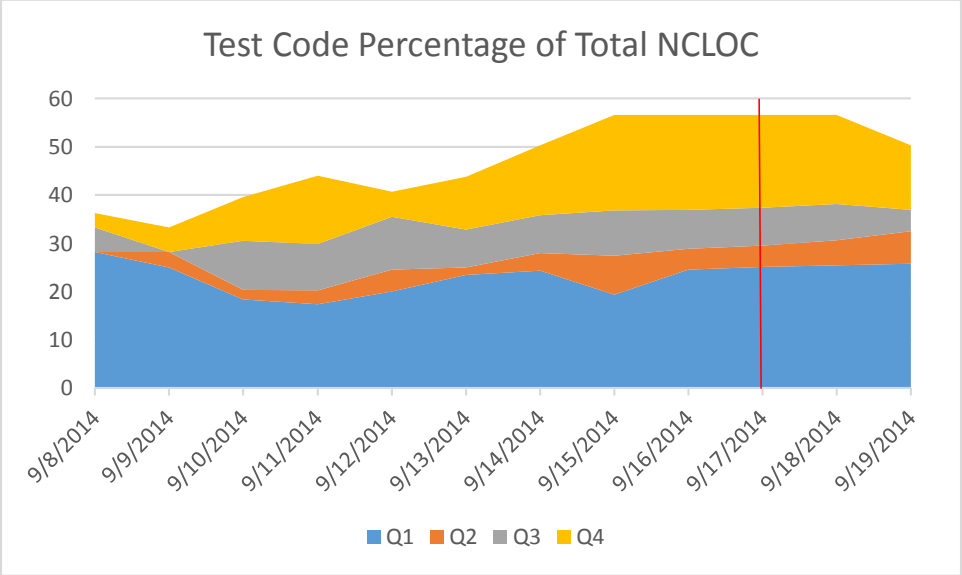


Figure 4.6 Test code percentage of total NCLOC

Figures 4.5 and 4.6 show the number of NCLOC in students' solutions and the percentage of total lines that are part of tests, by quartile, respectively. The line between Q2 and Q3 is the class average and the top of Q4 is the class maximum. Figure 4.5 allows instructors to see how code is being added, whether by large

amounts or gradually, and about how much code different portions of the class are writing. The percentage in Figure 4.6 allows instructors to see how students are distributing their effort, and whether they are utilizing test-driven development (TDD), a core principle of Web-CAT. This chart is used in both the class and student view, with class averages displayed in the former.

4.2.4 Cyclomatic Complexity

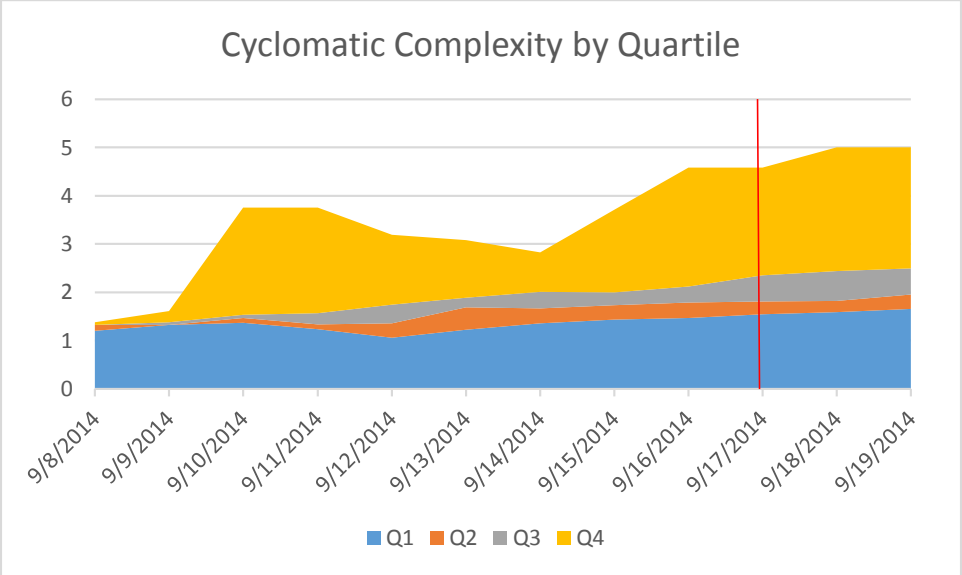


Figure 4.7 Cyclomatic complexity by quartile

Figure 4.7 shows the cyclomatic complexity of the class's code, on average, and also broken down by quartile. Cyclomatic complexity is here defined as the number of logical branches in solution code divided by the total number of methods. The top line in the chart represents the class's maximum complexity, with the four bands representing the number of students residing in each quartile of the maximum value. Hence the average complexity is shown by the line dividing the second and third quartiles (orange and gray in the example chart). This chart is used only in the class view, and is replaced by a line chart showing cyclomatic complexity over time in the individual student view.

4.2.5 Code Coverage Percentage

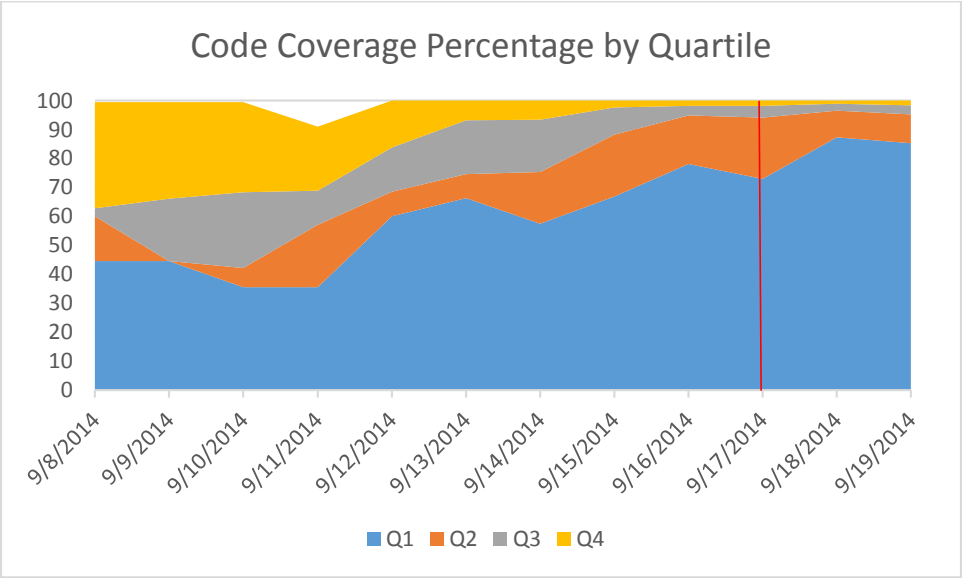


Figure 4.8 Code coverage percentage by quartile

Figure 4.8 shows the class's average coverage of solution code by test code plotted over time and split into quartiles. Because code coverage measures percentage of solution code run when all of a student's unit tests, this metric may be used to judge how the class is utilizing test-driven development. This chart is used in both the class and student view, with class averages displayed in the former.

4.2.6 Comment Percentage of Total Code

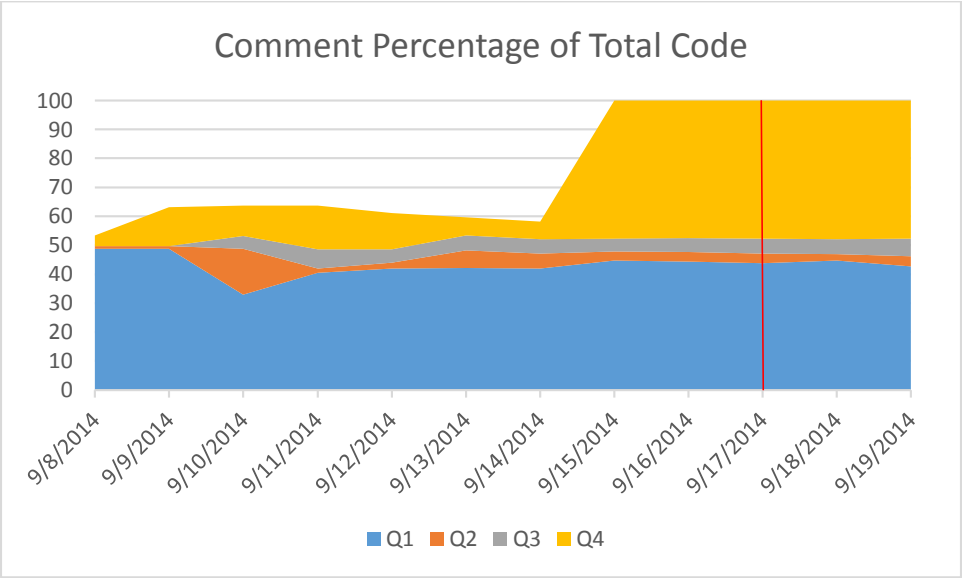


Figure 4.9 Comment percentage of total code

Figure 4.9 displays a percentage of the total lines of code that are comments, by quartile. For the Java code Web-CAT and Eclipse are primarily used for with this system, commented code consists of all lines with the single-line comment symbol (‘//’) or lines between the multi-line comment symbols (‘/**’ and ‘*/’). All other non-blank lines are counted as lines of code. This can help instructors determine whether students are following the good practice of commenting as they write code, or saving all of it until the end, as is quite common, but less useful. This chart is used in both the class and student view, with class averages displayed in the former.

4.2.7 Code Churn

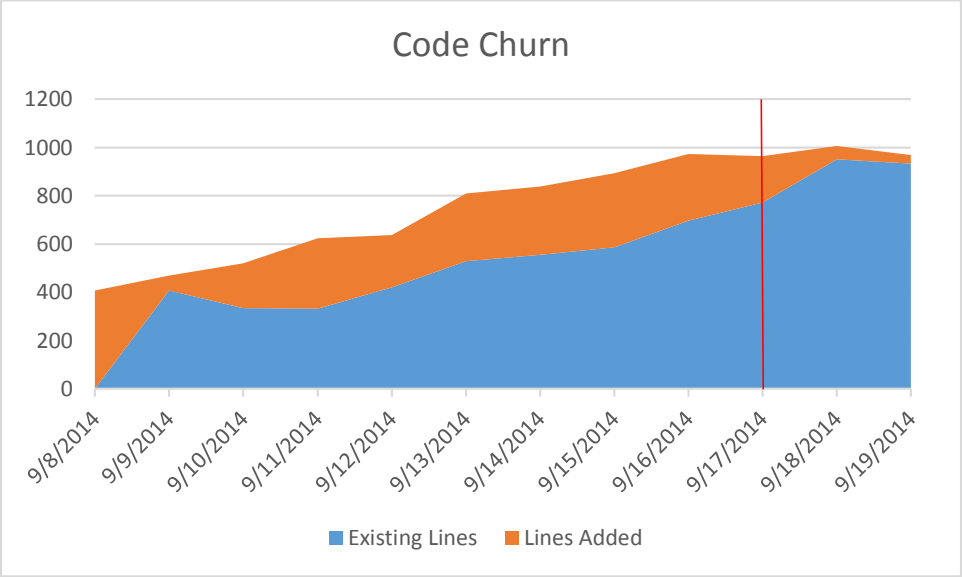


Figure 4.10 Code churn compared to total code

Figure 4.10 shows code churn as compared to total lines of code. Code churn is a software metric measuring the change between two versions of a code base, specifically the number of lines added for our use, which is often used to determine problem areas of code [11]. This chart displays as the top of the area the total number of lines of non-commented code, with the bottom band being previously written code and the top band showing new code. This is useful to determine actual progress over small timeframes and can be used with development time to determine in what time frames (relative to the assignment) students are getting stuck the most. This chart is used in both the class and student view, with class averages displayed in the former.

4.2.8 Code Velocity

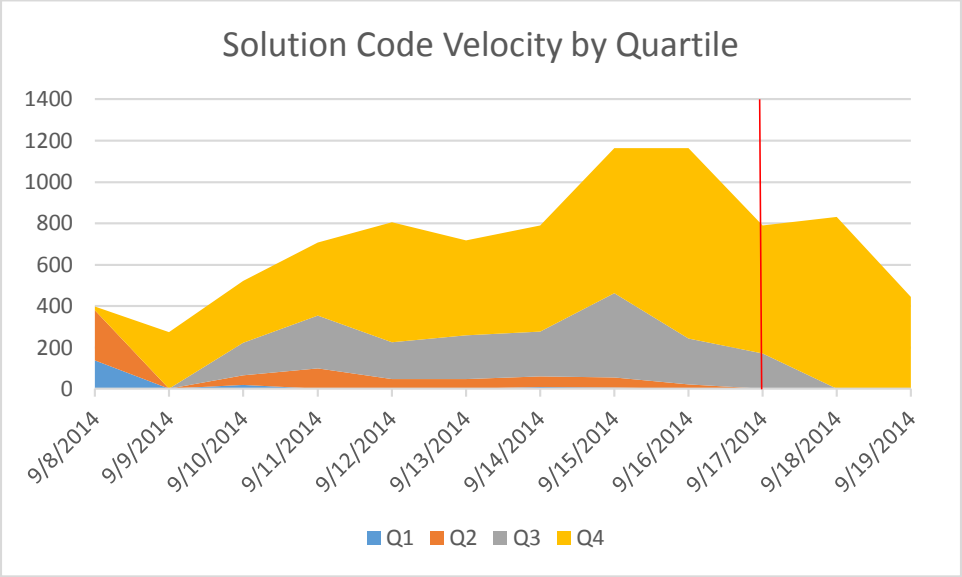


Figure 4.11 Velocity of solution code addition and edits by quartile

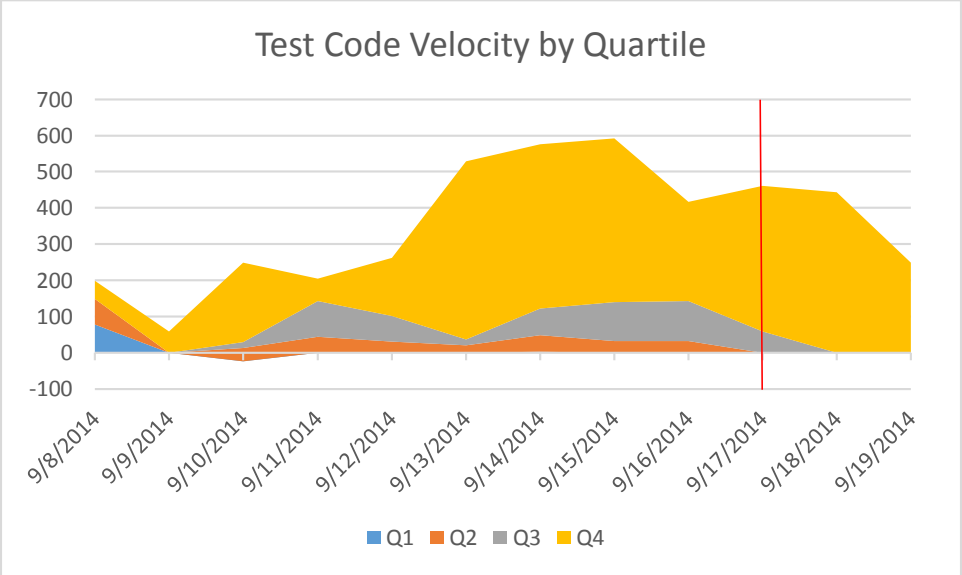


Figure 4.12 Velocity of test code addition and edits by quartile

Figures 4.11 and 4.12 display the velocity of code changes, both additions and edits, for solution code and test code, respectively. With these charts, instructors have extra information on edited lines of code in comparison to the charts that display changes in the number of lines of code above. A high number of edits

without a large change in amount of code might indicate failure to understand a particular piece of an assignment, specifically in the individual case, where an instructor can view reference test results and code snapshots to determine the problem area. This chart is used in both the class and student view, with class averages displayed in the former.

4.2.9 Reference Test Velocity

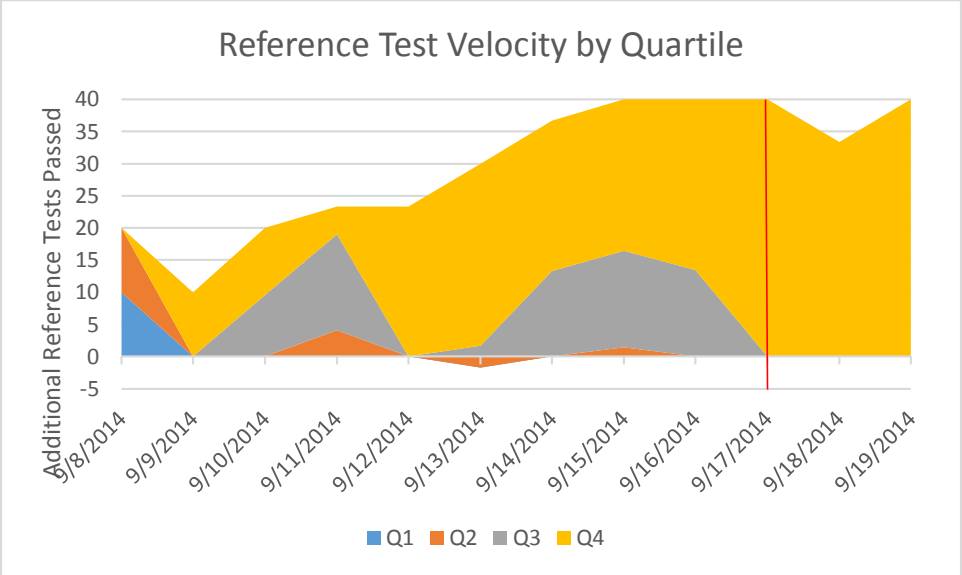


Figure 4.13 Reference test velocity by quartile

Figure 4.13 displays the velocity of reference tests pass percentage, or the number of additional reference tests passed since the last time label (e.g. in one day on this chart), by quartile. This is a quick way to determine when the most end-result progress was made on the assignment, either by the class or an individual. This chart is used in both the class and student view, with class averages displayed in the former.

Chapter 5: Conclusion and Future Work

The purpose of this thesis was to design and implement a system for continuously collecting fine-grained data about student software development. This data collection would enable future research in education and to inform development of classroom interventions against negative student habits, especially procrastination. To this end, the system was designed and implemented to collect event data and code snapshots from student IDEs without additional overhead in time or effort on the part of the student. The data that is collected is stored on a server where it can be viewed using an instructor dashboard with charts and tables designed to summarize how individual students and entire classes are performing on assignments. The software was designed to interface with existing Web-CAT software on both the client and server side in order to provide simple transition and use.

The system has been tested as it was developed to handle data collection in the single-user case, tracking live editing in Eclipse and sending manually generated log files. Further testing for larger-scale use will take place in Summer 2015, and the system will deploy in Fall 2015 if no major issues are found. The data collected and presented by this system were chosen as a starting point based on what was easily implemented and deemed likely to be useful by instructors and researchers. As this system is used, we will collect feedback about how useful instructors find the dashboard presentation and available charts, and iterate based on their feedback. If we determine that some data collection is unnecessary or other sensors need to be added, the Eclipse plugin can be passively updated to include new or remove extra sensors.

5.1 Contribution

The primary contribution of this work is the DevEventTracker plugin and Web-CAT subsystem that were developed. This software may be used with any Web-CAT installation with minimal refactoring, and as such allows institutions around the world the potential to track student data. Similar systems to the Eclipse plugin implemented for this system may be created in order to expand data collection to other IDEs, environments, and languages. Systems need only conform to the structure of the HTTP interactions described in Section

3.3.4 and run a Web-CAT server in order to collect and visualize data with our server-side code.

5.2 Future Work

As mentioned in the problem statement, the primary purpose of this research and the implementation of this software is to enable future research and development in educational research and intervention design. With the data available to instructors and researchers from the dashboard, it is envisioned that interventions can be automatically customized and generated to some degree, and potentially presented to the user, given some pre-planned ideas from an instructor. Interventions can also be refined through continual data, with results of previous versions and the current data stream input to a learning algorithm to tweak various settings and increase effectiveness. With these data, future research may also be conducted on ordering and structure of assignments within a class, as well as which pieces of assignments are more useful to be pre-written. One could also see research on how individual classes prepare students for the workplace and for other classes.

In addition to the above continuations, other branches of work include:

More sensors can be added to the plugin, including monitoring the results of unit testing and determining information about code structure.

Some low-level analysis can be performed by the plugin itself and displayed to the user in near real-time.

A student-viewable dashboard can be developed, with more novice-friendly statistics and information, as an automated indicator of how the student is performing on a particular assignment, and in the class. Related to this the Signals project [13] at Purdue University analyzes student class data and provides the student with an easily visible traffic light graphic on the front page of their course management system, with a green light indicating good progress, a yellow light indicating some problems, and a red light indicating major issues relative to the expected progress. A similar analogy would seem to be appropriate as an ongoing intervention on a student's dashboard page. This could indicate their progress on a particular assignment relative to the rest of the class or an instructor's standard, performance in the class overall, or as a summary of good and bad programming habits (i.e. procrastination, test-driven development).

References

1. Altadmri, A., & Brown, N. C. (2015, February). 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In Proceedings of the 46th ACM Technical Symposium on Computer Science Education (pp. 522-527). ACM.

2. Brown, N. C. C., Kölling, M., McCall, D., & Utting, I. (2014, March). Blackbox: a large scale repository of novice programmers' activity. In Proceedings of the 45th ACM technical symposium on Computer science education (pp. 223-228). ACM.

3. Edwards, S.H., Martin, J., & Shaffer, C.A. (2015). Examining classroom interventions to reduce procrastination. In Proceedings of the 2015 Conference on Innovation & Technology in Computer Science Education. ACM, to appear.

4. Glassy, L. (2006). Using version control to observe student software development processes. Journal of Computing Sciences in Colleges, 21(3), 99-106.

5. Jadud, M. C. (2006, September). Methods and tools for exploring novice compilation behaviour. In Proceedings of the second international workshop on Computing education research (pp. 73-84). ACM.

6. Jadud, M. C., & Henriksen, P. (2009, August). Flexible, reusable tools for studying novice programmers. In Proceedings of the fifth international workshop on Computing education research workshop (pp. 37-42). ACM.

7. Johnson, P. M., Kou, H., Agustin, J. M., Zhang, Q., Kagawa, A., & Yamashita, T. (2004, August). Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. In Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on (pp. 136-144). IEEE.

8. Johnson, P. M. (2007, September). Requirement and Design Trade-offs in Hackystat: An In-Process Software Engineering Measurement and Analysis System. In ESEM (Vol. 7, pp. 81-90).

9. Johnson, P. (2010). Hackystat - A framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data - Google Project Hosting. Retrieved April 10, 2015 from https://code.google.com/p/hackystat/.

10. Kölling, M., & Utting, I. (2012, February). Building an open, large-scale research data repository of initial programming student behaviour. In Proceedings of the 43rd ACM technical symposium on Computer Science Education (pp. 323-324). ACM.

11. Nagappan, N., & Ball, T. (2005, May). Use of relative code churn measures to predict system defect density. In Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on (pp. 284-292). IEEE.

12. Norris, C., Barry, F., Fenwick Jr, J. B., Reid, K., & Rountree, J. (2008, June). ClockIt: collecting quantitative data on how beginning software developers really work. In ACM SIGCSE Bulletin (Vol. 40, No. 3, pp. 37-41). ACM.

13. Pistilli, M. D., & Arnold, K. E. (2010). In practice: Purdue Signals: Mining real-time academic data to enhance student success. About Campus, 15(3), 22-24.

14. Restlet API Platform. (2015). Retrieved May 1, 2015, from http://restlet.com/.

15. Shah, A. (2003). Web-cat: A web-based center for automated testing. Unpublished master's thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia.

16. Spacco, J., Hovemeyer, D., & Pugh, W. (2004, October). An eclipse-based course project snapshot and submission system. In Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange (pp. 52-56). ACM.

17. Spacco, J., Strecker, J., Hovemeyer, D., & Pugh, W. (2005, May). Software repository mining with Marmoset: an automated programming project snapshot and testing system. In ACM SIGSOFT Software Engineering Notes (Vol. 30, No. 4, pp. 1-5). ACM.

18. Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J. K., & Padua-Perez, N. (2006). Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. ACM SIGCSE Bulletin, 38(3), 13-17.

19. Spacco, J., Pugh, W., Ayewah, N., & Hovemeyer, D. (2006, October). The Marmoset project: an automated snapshot, submission, and testing system. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (pp. 669-670). ACM.

20. Spacco, J., Fossati, D., Stamper, J., & Rivers, K. (2013, July). Towards improving programming habits to create better computer science course outcomes. In Proceedings of the 18th ACM conference on Innovation and technology in computer science education (pp. 243-248). ACM.