CLOCKIT: MONITORING AND VISUALIZING STUDENT SOFTWARE DEVELOPMENT PROFILES

A Thesis
by
JOSHUA JOEL ROUNTREE

Submitted to the Graduate School
Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

December 2010
Department of Computer Science

CLOCKIT: MONITORING AND VISUALIZING STUDENT SOFTWARE DEVELOPMENT PROFILES

A Thesis
by
JOSHUA JOEL ROUNTREE
December 2010

APPROVED BY:

____
E. Frank Barry
Chairperson, Thesis Committee

____
Cindy Norris
Member, Thesis Committee

____
James B. Fenwick Jr.
Member, Thesis Committee

____
James T. Wilkes
Chairperson, Computer Science

____
Edelma D. Huntley
Dean, Research and Graduate Studies

Copyright by © Joshua Joel Rountree 2010
All Rights Reserved

ABSTRACT

CLOCKIT: MONITORING AND VISUALIZING STUDENT SOFTWARE DEVELOPMENT PROFILES. (December 2010)

Joshua Joel Rountree, B.S., Appalachian State University

Chairperson: E. Frank Barry

Monitoring software development practices can result in improved estimation abilities and increased software quality. A common drawback associated with many monitoring schemes is the manual overhead needed to make the monitoring effective. This overhead results in users abandoning the monitoring scheme shortly after it is adopted or poor quality in the data produced. Alternatives have been introduced that automate part, or all of the monitoring. ClockIt is a fully automated extension for the pedagogical integrated development environment (IDE) BlueJ, and focuses on aspects of the development practices seen in introductory level students. By automatically monitoring introductory student development behavior, instructors and students gain insight about development practices. In addition to the ClockIt extension, Visualization tools are provided to assist students or instructors in exploring the data. Data collected via ClockIt for four semesters confirm previous independent findings. And, new insights about how compilation error frequency changes in introductory students and the relationships between pairs of compilations have been discovered.

TABLE OF CONTENTS

ABSTRACT iv

TABLE OF CONTENTS v

List of Tables viii

List of Figures ix

Chapter 1 – Introduction 1

1.1 Problem 1

1.2 Solutions 1

1.3 ClockIt Solution 2

1.4 Thesis Overview 2

Chapter 2 – Background 4

2.1 First Generation Approaches 5

2.1.1 PSP 5

2.2 Second Generation Approaches 6

2.2.1 Leap Toolkit 6

2.2.2 PSP Tool 7

2.2.3 PSP Studio 7

2.2.4 PSP Dashboard 8

2.3 Third Generation Approaches 8

2.3.1 Hackystat.. 8

2.3.2 Marmoset 9

2.3.3 Eclipse Watcher Plugin.. 10

2.3.4 Novice Compilation Behavior 10

2.3.5 ClockIt.. 11

Chapter 3 – Data Logger Extension.. 13

3.1 BlueJ Extension 13

3.2 Log File... 14

3.3 Events Stored 15

3.3.1 Project Events 16

3.3.2 Package Events 16

3.3.3 Compilation Events.. 17

3.3.4 Invocation Events... 19

3.3.5 File Change Events 20

3.3.6 User Name Events.. 21

3.3.7 Version Events 22

3.4 Uploading User Data... 23

Chapter 4 – Visualization Extension.. 24

4.1 Invoking The Visualizer Inside BlueJ.. 24

4.2 External Visualizer.. 25

4.3 Visualizer Views 26
4.3.1 Summary Tab 27
4.3.2 Overview Tab 29
4.3.3 Compilation / Invocation Tab 31
4.3.4 Statistics Tab 32
4.3.5 Log File Tab 33
Chapter 5 – Results 34
5.1 Most Common Errors 34
5.2 Successive Compilations 35
5.2.1 Time Differential 35
5.2.2 Compilation Event Pairs 36
5.2.3 Combined Perspective 37
5.3 A New Insight About Compilation Errors 38
5.4 Student Cheating 39
Chapter 6 – Summary and Future Work 40
6.1 Future Work 41
Bibliography 43
Biographical Information 47

List of Tables

Table 5.1 – Most common errors encountered by students 35

Table 5.2 – Time between compilation event pairings. 38

Table 5.3 – Compile errors, early term vs. late term 39

List of Figures

Figure 3.1 – BlueJ tools menu item customized with ClockIt tool ... 14

Figure 3.2 – Event structure ... 15

Figure 3.3 – Project open/close event structures ... 16

Figure 3.4 – Package open/close event structures ... 17

Figure 3.5 – Successful compile event structure ... 17

Figure 3.6 – Compile error event structure ... 18

Figure 3.7 – Compile warning event structure ... 18

Figure 3.8 – BlueJ object bench ... 19

Figure 3.9 – Invocation event structure ... 20

Figure 3.10 – File change event structure ... 21

Figure 3.11 – File delete event structure ... 21

Figure 3.12 – User name prompt dialog window ... 22

Figure 3.13 – User name event structure ... 22

Figure 3.14 – Version event structure ... 22

Figure 4.1 – Starting the ClockIt visualizer BlueJ extension ... 25

Figure 4.2 – External visualization tool directory search results ... 26

Figure 4.3 – ClockIt visualizer summary tab view ... 27

Figure 4.4 – ClockIt visualizer overview tab ... 30

Figure 4.5 – ClockIt visualizer compilation/invocation tab ... 31

Figure 4.6 – ClockIt visualizer statistics tab ... 32

Figure 4.7 - ClockIt visualizer log file tab 33

Figure 5.1 – Time between successive compilation events. 36

Figure 5.2 – Successive compilation event pairings. 37

Chapter 1 – Introduction

1.1 Problem

According to the Bureau of Labor Statistics, employment for computer software engineers is projected to increase by 32 percent over the 2008 to 2018 period (Bureau of Labor Statistics, 2010). Despite this positive outlook, Computer Science has only recently begun to recover from a significant decline in Computer Science as a major. The most recent Taulbee Survey, which reports on enrollment trends and degree production in computer science departments, shows the first increase, 1.7 percent, in computer science enrollment in six years (Taulbee, 2009). While the enrollment trend has changed, it does not match the increase in jobs, nor does it negate the previous six years of declining enrollment. Moreover, it is unclear how significant or long lasting this change may be. In addition to the problem of attracting new students, maintaining already enrolled students is becoming increasingly more difficult. Attrition rates as high as 30 to 40 percent are being seen with the majority occurring during the freshman and sophomore years (Beaubouef, & Mason, 2005).

1.2 Solutions

Solutions to lessen the attrition rate and better prepare students entering the job market have changed how introductory courses are taught. How to teach basic concepts and skills to introductory students is a much debated topic with strong opinions for both object oriented and procedural approaches (Bruce, 2005). Regardless of opinions, the object oriented paradigm has been embraced, with Java as the language of choice. In an effort to

allow introductory students to more easily grasp the complex abstractions of object oriented principles, pedagogical IDEs are being deployed in place of simple text editors. Pedagogical IDEs present language concepts in a graphical and automated way allowing students to visualize the abstractions and interactions that are key to understanding object oriented languages (Allen, Cartwright, & Stoler, 2002; Hsia, Simpson, Smith, & Cartwright, 2005; Reis & Cartwright, 2004).

1.3 ClockIt Solution

ClockIt is a solution to explore the attrition problem by monitoring and studying the programming habits of introductory students. Using BlueJ, a widely used object oriented pedagogical IDE[1], ClockIt can monitor and log the software development process of individual students. While there is a cognitive component to learning programming, an important process component has been identified as well (Shaffer, 2005). Logging software development events provides information about how a student develops software that will aid in determining what characteristics are representative of both good and bad programming practices. Knowing when students begin a project or how long a programming session lasts in conjunction with programming events such as compilation or testing allows investigation of the process component students go through.

1.4 Thesis Overview

Several processes or methodologies already exist to monitor development activities, events, and practices. These processes are covered in Chapter 2 and compared to ClockIt. Chapter 3 describes the implementation details of the BlueJ extension to log events that occur during development. Chapter 4 discusses the BlueJ extension that provides feedback and

[1] The BlueJ website lists over 400 institutions at which BlueJ is currently being used.

visualization based on the events in a log file. Chapter 5 reports on the impact and usefulness of 4 semesters of classes using ClockIt and how ClockIt can be applied to both the student and the instructor. Chapter 6 summarizes ClockIt and discusses future work.

Chapter 2 – Background

Using a process based approach to programming has shown to improve not only the quality of the programming but also reducing the time needed to produce a finished product (Humphrey, 1996). Several methodologies currently exist to collect information about how a developer creates software and aids in improving time estimation and reducing defects. A common drawback associated with these methodologies is the amount of developer overhead needed to ensure their success, and many are geared for professional developers, not students. The goal in developing the ClockIt project is to provide a monitoring system which requires little, if any, learning overhead, and use it to collect introductory student development data.

The different approaches to metric collection are classified into three generations (Johnson, Kou, Agustin, Chan, Moore, Miglani et al. 2003). The first generation approaches require all metric collection and data analysis to be performed manually. Manual Personal Software Practice (PSP) is in this category. Second generation approaches consist of manual metric collection, but eliminate the need for paper by providing automated storage and analysis via the use of a tool. These partially automated approaches include the Leap toolkit, PSP Tool, PSP Studio, and PSP Dashboard. Third generation tools automate both the metric collection and analysis. By automating both the collection and the analysis, the burden of switching between metric collection and development is relieved. This also ensures metric collection is accurate by placing the responsibility on the tool, not the developer. Third generation approaches include Hackystat, Marmoset, Eclipse Watcher, research by Matthew

C. Jadud, and ClockIt. The following sections discuss prior work regarding estimation and monitoring of software development. This discussion is focused on what type of information is gathered, benefits from the gathered information, and effort needed for the monitoring to be beneficial.

2.1 First Generation Approaches

2.1.1 PSP

PSP is a well-defined process-based approach to software development that is learned through four steps (Humphrey, 1995; Humphrey, 1997). Personal Measurement (PSP0) serves to establish a baseline by measuring development time and defects. Personal Planning (PSP1) shows how to estimate program size and development time based on data from the previous step, and how to plan schedules and tasks. Personal Quality (PSP2) deals with defect management by using checklists in design and code review, and applying design specifications. Scaling Up (PSP3) can be used to apply multiple PSP2 steps to develop a program consisting of multiple modules.

Each PSP step is accompanied by scripts used as a guide for filling out the forms necessary for learning about how an individual programs. The steps are used to demonstrate the process and also provide initial data for calculations that are used to estimate future work. A significant investment in time and effort, about 150 to 200 hours, is needed to learn how to correctly apply PSP (Humphrey, 1995).

Multiple success stories have shown the usefulness of PSP in keeping projects on schedule and improving quality (O’Connor & Coleman, 2002). While these benefits have been realized, several drawbacks have been identified as well. The manual recording of data is not strictly followed, which results in data quality problems (O’Connor & Coleman, 2002).

Manual recording of data has also been attributed to a false success in PSP (Disney & Johnson, 1998). Developers tire of manually recording defect information so it appears that defects have lessened, when in actuality they are not recorded.

2.2 Second Generation Approaches

Second generation approaches provides tools to aid in collection of data and automate analyses in an attempt to improve data quality concerns, and increase usage by relieving users of manually recording data on paper.

2.2.1 Leap Toolkit

Similar to manual PSP, reflective software engineering (Johnson, 1999; Moore, 2000) enables a developer to improve estimation abilities and quality, but achieves this through automated support. This support is provided via the Leap (*lightweight, empirical, anti-measurement dysfunction, portable*) toolkit. The Leap toolkit gives developers the ability to record a large variety of metrics including the size of a project, time taken for development, defects found, patterns learned, checklists used during design, estimations for time and size, goals, questions, and measures that motivate the data recording (Moore, 2000). Although the data collected must be manually entered, the collection is paperless freeing the developer from the burden of creating paper forms like those associated with manual PSP. The tool also provides automatic analyses and reports generated from the data.

Although the Leap toolkit automates the process of recording data, a developer is still required to manually enter data such as the size of a project and elapsed time. Research indicates that developers feel that the time spent manually recording data could be better spent on developing software (Johnson, Moore, Dane, & Brewer, 2000).

2.2.2 PSP Tool

The PSP Tool is a web based tool that students use to record their weekly development habits (Ferguson, Grissom, & Berberoglu, 2000). These development habits are characterized by class and lab attendance, hours spent studying, opinions on weekly workload, course satisfaction, time estimates, actual time, and optional comments. Students use a worksheet to record their daily development habits, and enter this data using the tool at the end of each week. Students and faculty have access to summary information based on data from all students. The tool is aimed at beginning Computer Science students and produces charts based on satisfaction and workload, attendance and study time, time estimates versus actual time, and correlations between grades and the ratio of time estimates and actual time.

Drawbacks to the PSP Tool are the same as other partially automated methods. Problems exist with the students' inability to provide accurate data. One study reports that only forty-nine percent of students' data was 'accurate' or 'very accurate' (Ferguson et al., 2000).

2.2.3 PSP Studio

The Personal Software Process Studio (PSPS) developed by the Design Studio team at East Tennessee State University (ETSU) automates data collection for the PSP (ETSU, n.d.). PSPS provides tables for "scripts" that define the steps needed in each development phase, logs to record the amount of time spent and software defects, and summarization information about the project. PSPS also stores all past information in a local database allowing users to recall information from previous projects. PSPS also suffers from the drawback that developers must switch from software development to recording information

using PSPS in order for the tool to be effective. In addition, the tool relies on the developer to provide accurate measurements.

2.2.4 PSP Dashboard

Like the other partially automated tools, PSP Dashboard requires users to enter data manually that is then used to provide automatic analyses. The software was originally developed in 1998 by the United States Air Force, but now exists under the open-source model (The Software Process Dashboard, n.d.). The Dashboard is aptly named in that it is a small, lightweight utility that takes up very little screen real estate. It provides scripts that will walk a developer through each phase in PSP, and also makes use of audible cues for completed steps. Buttons are provided for recording defects and time spent correcting errors. After all development phases have been completed, data analysis can be performed. The data analysis provides metrics for defects, planning, and processes, and can be based across all projects. PSP Dashboard is like other PSP tools that require manual data entry to make the tool effective.

2.3 Third Generation Approaches

Third generation approaches alleviate developer burden further by automating the data collection instead of providing a tool to facilitate data collection as seen in second generation approaches.

2.3.1 Hackystat

Hackystat was developed in response to the deficiencies associated with manual, or partially automated, software process schemes (Johnson et al., 2003). Hackystat resolves these deficiencies by using “sensors” to fully automate the recording of software metrics. Client-side sensors are attached to development tools such as IDEs, build tools, and testing

tools. These sensors collect data about active file modification, size data (such as the Chidamber-Kemerer OO metrics (Chidamber & Kemerer, 1994) and non-comment lines of code), defect data, and performance of unit tests (Hackystat, n.d.).

Hackystat users install client-side sensors and register with a Hackystat server. Once registered, development data is collected by the sensors and sent to the server automatically and unobtrusively. The server regularly performs data analysis, and groups the raw data stream into 5 minute intervals. This abstraction of the raw data stream serves as the "Daily Diary," and is the basis for other analyses such as the amount of developer effort on a given module, change in the size of a module, distribution of unit tests, and invocation results. Users of Hackystat are also able to define analysis threshold parameters and receive informative emails when those thresholds have been reached.

Although Hackystat is an automatic solution, developers must still switch from developing in an IDE to a web browser in order to see analyses. Hackystat also requires multiple tools and sensors to be installed such as an IDE, a build tool, and a unit testing tool in order for all metric data to be collected. Because of this overhead, Hackystat is more suitable for research, industry applications, and upper level software engineering courses where advanced IDEs are used.

2.3.2 Marmoset

Marmoset is an automatic data collection, submission, and testing plugin for the Eclipse IDE that adds a student's project to a repository each time a file is added or removed, or the project is saved (Marmoset, n.d.). The students are also able to explicitly submit their projects and test them against different test classes. There are three classes of tests available. Public tests are available to students directly and may be run at any time with the results

available as well. Secret tests are only used for grading purposes and are optionally available after grading deadlines. Release tests are not distributed to students but the results are available in a time limited manner, and the results display all passing tests but only the first two failed tests. The testing system inside Marmoset provides better feedback to students while the snapshots are used to find common errors among students.

2.3.3 Eclipse Watcher Plugin

The Eclipse Watcher plugin monitors browsing, scrollbar, mouse, and coding events and generates an XML file for each session (McKeogh & Exton, 2004). Like other third generation tools, the user is not required to interact with the plug-in in any way, thus ensuring data integrity. Once the xml data files are produced, post mortem tools abstract the raw data into a general graph and an event graph. The general graph shows the time spent on each line of code per file, and the event graph shows the distribution of browsing, scrollbar, mouse, and coding events per file. These graphs can be used to identify which areas of code received the most attention, and how the IDE itself was used. Unfortunately the Watcher plug-in shows no testing data, nor any compilation data, both of which can be useful. In addition, since the plug-in is for the powerful Eclipse IDE, its usefulness for lower-level CS students is limited.

2.3.4 Novice Compilation Behavior

Jadud presents research focused on the compile/edit cycle of novice programmers (Jadud, 2006a). A BlueJ extension collects data on students as they compile programs using BlueJ and stores the compilation information in a database. Jadud describes various dimensions about novice programmers. For example, the time between compilations, and the syntactical differences between them are used to compute how many syntax errors a student

encountered during a programming session and whether or not the student fixed them. This is known as the *error quotient* and is used to draw conclusions on effectiveness of the programming.

The research contained in (Jadud, 2006a) is significant but limited by its focus on compilation behavior. Quantitative properties such as the time spent on a project, when the project was started, and testing data could be used to profile successful students and poor students.

2.3.5 ClockIt

The subject of this research effort, ClockIt, is an extension for the BlueJ IDE that monitors and stores events generated during development. The logging of events is invisible to the user with no perceived difference than if the tool was not present. Using an automatic logging utility circumvents the reliance on the user to record accurate and precise information and frees them from having to switch to a tool to record data. The user can fully focus on programming, not recording development data. This is useful to introductory students as it allows for accurate data to be collected, but doesn't require students to learn how to collect data.

All BlueJ compilation and invocation behavior is recorded in the log file. This data is used to show compile and invocation events and their results. Information about start and end time is used to show the amount of time spent on a project as well as when the time was spent. The information is beneficial to instructors because instructors rarely have the time to sit down with each individual student to monitor how the student develops projects. By collecting data on students, instructors can look at past projects of students that are doing poorly and intervene before problems are beyond repair. Instructors can also use the

information to better gauge the effort and time taken to complete projects. This information can assist them in tailoring assignments to better suit the needs of the students.

Chapter 3 – Data Logger Extension

ClockIt is composed of two major components that are responsible for logging student programming events and providing a visual representation of the data. This chapter discusses how student event data is logged while using the BlueJ IDE. Chapter 4 describes the data visualization component. The following sections discuss the data logger extension implementation with regards to the extension API, log file design, a detailed description of events captured, and how logged data is uploaded to a central database.

3.1 BlueJ Extension

BlueJ supports extending functionality of the IDE via an extension API; other software systems call these extensions “plugins”. An extension is created by subclassing the *Extension* base class and overriding a *startup* method that is called when the extension starts its work. The BlueJ system creates your custom extension object and calls the startup method passing a proxy object to itself. Using this proxy object the extension can register event listeners for each category of reported event. ClockIt registers event listeners for compilation, invocation, and package events.

Extension writers are provided mechanisms that allow some customization of the BlueJ user interface. Specifically, an extension may add menu items to BlueJ’s tools menu, class menu, or object menu. A tool menu item provides a way for a user to invoke optional tools, such as the ClockIt BlueJ visualization tool as shown in figure 3.1. The class menu item and object menu item allow users to provide tools specific to a class or object respectively. The ability to add items to BlueJ’s preference panel is also available and allows

the creation and saving of user preferences pertinent to a particular extension. The preferences are saved in a preferences file that persists between BlueJ program executions. ClockIt exposes a user definable parameter via the preferences panel that is used in the visualization extension.

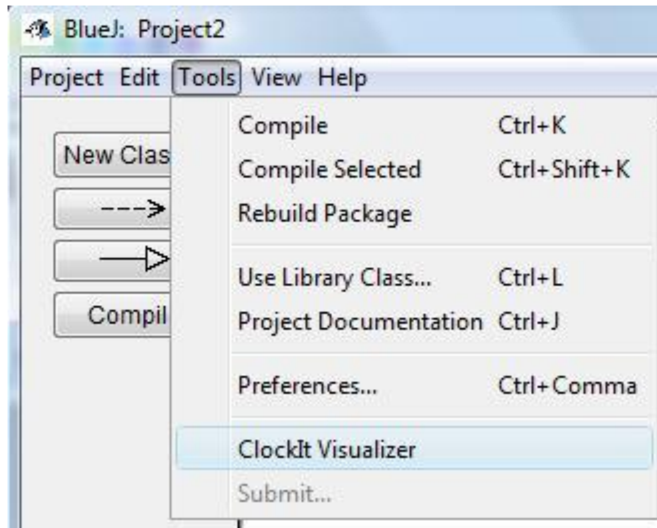

Figure 3.1 – BlueJ tools menu item customized with ClockIt tool

Lastly, users are allowed to either extend or completely replace the editor portion of the IDE. The source code for the editor is provided along with documentation needed to interface a custom editor with the IDE. By providing the editor source code and documentation, users are allowed to add enhancements to the editor such as auto indention and syntax highlighting. ClockIt does not replace any portion of the editor.

3.2 Log File

ClockIt creates and accesses the log file using methods and classes provided by the Java API, namely the *DataInputStream* and *DataOutputStream* classes, and *writeUTF* and *readUTF* methods. The *writeUTF* method is used to write data to the log file, and *readUTF* is used to retrieve data. Conveniently, *writeUTF* allows the log file to be appended each time an event occurs preventing the need to read the log file each time before data can be added.

All event data that is written to the log file shares a generic structure. The first line (line 0) contains information about when the event occurred and what type of event is present. The event timestamp is shown in two ways: milliseconds since midnight 1970, which is commonly used in timing, and in a human readable form of the time which aids in reading the raw log file. Finally, at the end of line 0 is the event identifier. Subsequent lines contain event specific information. In Figure 3.2 the event identifier shows that a project was opened and that the event occurred on March 31st. The name of the opened project is contained in the one subsequent line of the log file; in this example, the project was named "NewProject."

0	1206942505468 3/31/08 1:48:25 AM #openProject
1	NewProject

Figure 3.2 – Event structure

3.3 Events Stored

In order to capture development behavior, ClockIt registers event listeners for compile events, package events, and invocation events. Since BlueJ is a front-end to Sun's Java SDK, the layout of a BlueJ project mirrors that of Java programs with class organization based on packages. Package based organization results in package events being reported when a project is created or opened. Package events are used to create the log file for a project, or open a previously created log file. Each BlueJ project has its own directory and the log file location is based on this project location. Thus, the ClockIt extension can always locate the log file because it is kept with the BlueJ project files. In addition, students are able to move entire projects without worrying about the log file, since the creation of the log file and its use are transparent to the user.

3.3.1 Project Events

The project events, project open and project close, are inferred by the number of packages currently open for a given project. If there are zero packages open when BlueJ reports a package open event, ClockIT will append a project open event to the log file. Similarly, if there are zero packages open after a package close, a project close event is logged. Both project open and close events are denoted by the line 0 timestamp and event identifier. There is 1 additional line that is used to record the project name. Figure 3.3 shows a project being opened followed by the same project being closed. Project close and opens are used to determine total time spent on a project.

0	1206942505468 3/31/08 1:48:25 AM #openProject
1	NewProject
2	1206942509765 3/31/08 1:48:29 AM #closeProject
3	NewProject

Figure 3.3 – Project open/close event structures

3.3.2 Package Events

As their name implies, package open and closes occur when a package is either opened or closed. Package events are used to initialize resources for a project and to keep track of files contained in the package. When the first package of a project is opened a ClockIt descriptor is created and then added to a list of descriptors kept for all currently open projects. A descriptor keeps track of the open packages for a project and is responsible for creating a file monitoring thread when a package is opened, and stopping the thread when a package closes. The descriptor is also responsible for log file access and provides a “synchronized” method through which all event data is written to the log file. The Java language specification ensures a method labeled “synchronized” is accessed by only one thread at a time. Figure 3.4 shows the information about a package open and a package close.

Like other events, line 0 contains the time the event occurred and the event identifier. On line 1 is the package name, which is commonly blank signifying the default nameless package. Line 2 indicates the number of files in the package with 3 additional lines per file that record the file's path, number of lines of code, and number of lines of comment.

0	1206942505515 3/31/08 1:48:25 AM #openPackage
1	
2	1
3	C:\bluej\NewProject\MyClass.java
4	12
5	18
6	1206942509765 3/31/08 1:48:29 AM #closePackage
7	
8	1
9	C:\bluej\NewProject\MyClass.java
10	12
11	18

Figure 3.4 – Package open/close event structures

3.3.3 Compilation Events

Compilation events occur when a user presses the compile button in BlueJ. Since BlueJ allows multiple projects to be open at once, ClockIt must find the project to which the compile event belongs by looking through the list of descriptors for each open project. Once the project descriptor is found the appropriate event information is written to the log file. A compilation can result in a success, error, or warning.

0	1191098067109 9/29/07 4:34:27 PM #compileSucceeded
1	1
2	C:\bluej\NewProject\MyClass.java
3	12
4	18

Figure 3.5 – Successful compile event structure

Figure 3.5 shows the information stored about a successful compile. Following the time and event identifier is a number showing how many files are involved in the compile. Each file's location and the number of lines of code, and the number of lines of comments

are shown next. Figure 3.6 shows the information stored about a compile error. Following the event identifier is the line number containing the error. The error message is shown next with the number of files compiled, each with their number of lines of code and number of lines of comment.

0	1191090094312 9/29/07 2:21:34 PM #compileError
1	12
2	cannot find symbol - class ArrayList
3	1
4	C:\bluej\NewProject\MyClass.java
5	12
6	18

Figure 3.6 – Compile error event structure

The final compilation event recorded is a compilation warning. Although not likely in a strongly typed language such as Java, they are possible when using the generics available in Java 1.5 and 1.6. Figure 3.7 shows the information stored for a compile warning. Following line 0 is the warning message and number of files compiled in the compile. Each file's attributes are stored next.

0	1191090111468 9/29/07 2:21:51 PM #compileWarning
1	Note: C:\bluej\NewProject\MyClass.java uses unchecked or unsafe operations. Note: Recompile with -Xlint:unchecked for details.
2	1
3	C:\bluej\NewProject\MyClass.java
4	12
5	18

Figure 3.7 – Compile warning event structure

Compilation error events are used to track the errors encountered during a project and their frequency. Successful compiles can be used to indicate how much information a student changes between successful compiles. This information can help to reinforce good programming practices such as iterative development. Warnings can be used to track students' grasp of concepts such as the generics seen in Java 1.5 and 1.6.

3.3.4 Invocation Events

When a user wishes to execute some part of their program within BlueJ they often do not invoke the main method traditionally used to execute a program. Instead BlueJ provides a graphical menu to create objects. Once an object has been instantiated it is placed on the "object bench." Figure 3.8 shows an instantiated object on the object bench with the right-click context menu shown and the "sampleMethod" method highlighted.

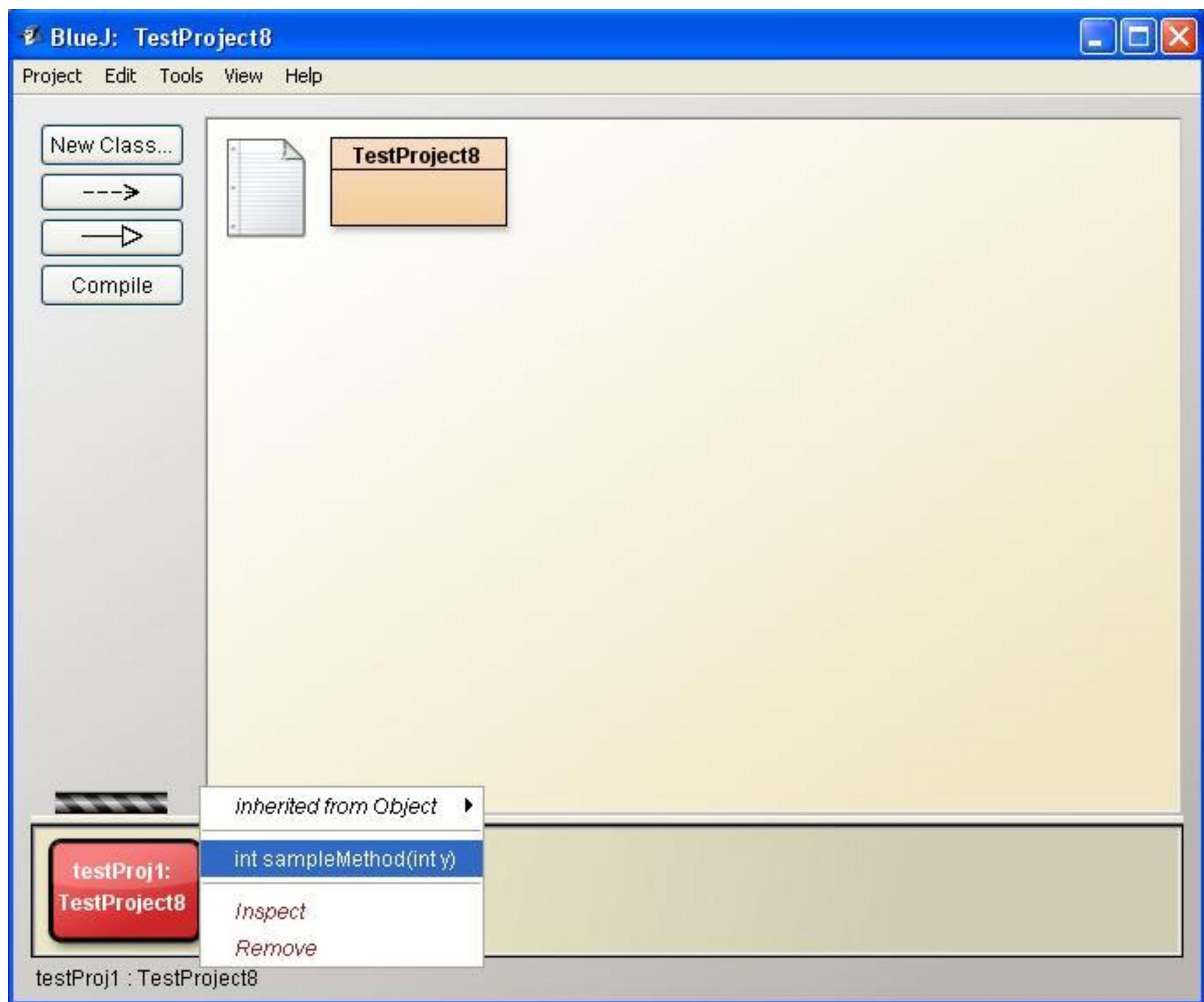

Figure 3.8 – BlueJ object bench

Invocation events are generated when users instantiate an object in the BlueJ project, or invoke a method on an object in BlueJ's "object bench". Figure 3.9 shows the information stored for an invocation. The first piece of information written to the log file is the line 0 time

stamp and event identifier. Following the event identifier is the package that contains the instantiated object followed on the next line by the class name itself. Line 3 records the result of the invocation (exception, forced exit, normal exit, terminated exit, unknown exit) coded as an integer. The next line identifies the method invoked with null indicating a constructor method. The last two lines indicated the number of lines of comment and the number of lines of code.

0	1191090573296 9/29/07 2:29:33 PM #invocation
1	NewPackage
2	NewClass.java
3	1
4	Null
5	13
6	18

Figure 3.9 – Invocation event structure

Recording invocation events gives an instructor an idea of the amount of testing performed by the student during development. For example, instructors commonly believe that students do not test until all code is developed (Reid, 2007). Collecting invocation information allows us to verify or refute this belief.

3.3.5 File Change Events

A package open event causes ClockIt to begin the execution of a thread responsible for monitoring all files in the package. The monitoring thread operates with a frequency of one second and generates a file change event when a change occurs to a file in the package. A file change occurs when the editing buffer inside BlueJ is saved to disk. This occurs automatically when the compilation button is pressed or when a user closes an editing window and changes were not previously saved to disk. Figure 3.10 shows the information stored for a file change event. The event specific data lines record the name of the file along with the number of lines of code and the number of lines of comments.

0	1191090054015 9/29/07 2:20:54 PM #fileChange
1	C:\bluej\NewProject\MyClass.java
2	13
3	18

Figure 3.10 – File change event structure

One could argue that the file monitor thread is unnecessary since file changes can be tracked from compilation or invocation events, but its purpose is to capture changes in files that occur when a user exits an editor window without compiling or invoking methods. For example, if a student is working in a lab setting and the lab ends, but the student has not yet completed enough editing to warrant a compile, the closing of the editor window results in a file change event. Even if the overall lines of code or lines of comments do not change, a student may modify a file. This type of activity will result in a sequence of file change events indicating the effort that is occurring. File deletes are also monitored and may provide information about whether or not a student decided to start over on a particular file. As shown in Figure 3.11 file delete events only name the file.

0	1210928024000 5/16/08 4:53:44 AM #fileDelete
1	C:\bluej\NewProject\MyClass.java

Figure 3.11 – File delete event structure

3.3.6 User Name Events

When a user opens a project they are presented with a modal dialog requesting their email address that is used for their user name. Figure 3.12 shows this dialog. By clicking the “OK” button, the ClockIt extension communicates with a servlet (see section 3.4) to check the validity of the user name. If the user name is not valid, the user is prompted again. After two invalid entries, the dialog is removed.

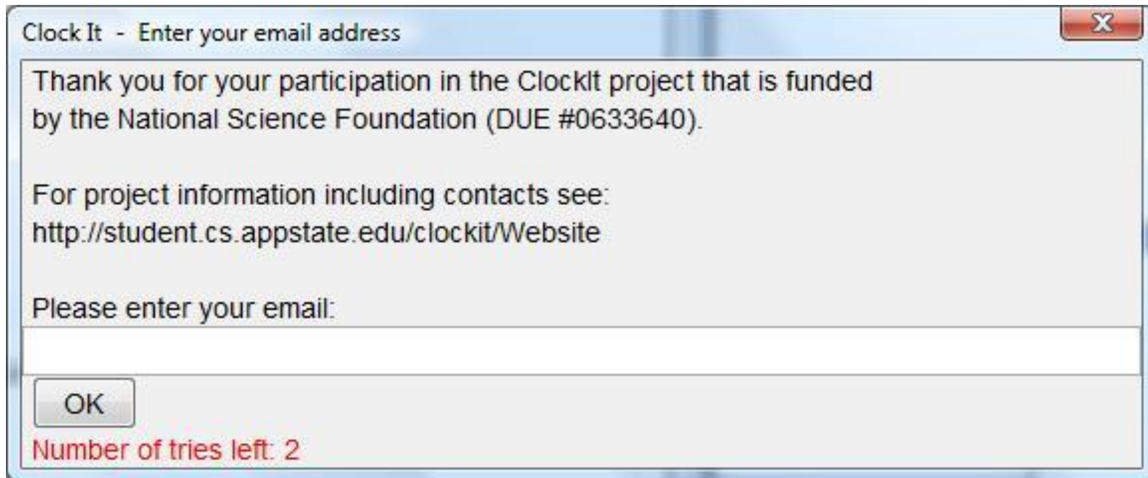

Figure 3.12 – User name prompt dialog window

After a valid entry has been received the user name is stored in the log file. Figure 3.13 shows the event structure consisting of the time stamp and the user's email address. Prior to showing the dialog box the log file is scanned for previous stored user name events. If a user name is found the dialog box is pre-filled with this information so the user does not have to enter it again; they can simply click "OK". User name events provide a way to track projects once they are submitted to a database. They also provide clues to the possibility of cheating when two different user name events exist in a single log file.

0	1287610409917 10/20/10 5:33:29 PM #userName
1	jr37596@appstate.edu

Figure 3.13 – User name event structure

3.3.7 Version Events

A version event is provided to track changes made to the ClockIt extension itself. The events captured or the data produced by each event may change. Knowing the version number provides a way to know what data is expected with each event and its format and will allow for backward compatibility should the need arise.

0	1281801697794 8/14/10 12:01:37 PM #version
1	1.2

Figure 3.14 – Version event structure

3.4 Uploading User Data

Providing information about single projects is useful, but being able to analyze data against multiple students, different semesters, or other schools provides valuable insight into how students program as a whole. Aggregating and storing development data was designed to be as seamless and transparent as possible. Students should not be burdened with having to manually submit their projects. If a student has a valid email associated with their project, the project files are uploaded to a repository via an HTTP-POST utilizing the features provided by the Apache Software Foundation's HTTP Components project (Apache Software Foundation, n.d. a). If an email is not present in the log file, then a submission will not occur because email addresses are used to track projects. If an error occurs a dialog box will be shown indicating the error.

On the receiving end of the HTTP-POST is a Java servlet that runs in Apache Tomcat (Apache Software Foundation, n.d. b). In addition to Tomcat, the Commons FileUpload package is used to process the files submitted by the HTTP-POST (Apache Software Foundation, n.d. c). After a successful submission, the files are saved in a directory based upon email address and the project name. After the files are saved, the servlet uploads the project events into a database which serves to provide the data for other analyses (Reid, 2007).

Chapter 4 – Visualization Extension

Visualizing the data recorded in a log file attempts to provide information about programming behavior that is easier to grasp than sequentially reading through the file. The visualization extension has both summary data and detailed, event-by-event information about how a student programs. The visualization extension provides an instructor the ability to see how a student progresses through program implementation and can aid in identifying problem areas. While the visualizations provided are not exhaustive, they do allow navigation between points of interest with varying degrees of granularity.

To offer both instructors and students flexibility, the visualization extension is separate from the data logging extension and can be excluded if an instructor wishes to install only the data logging extension. Separating the data logger extension from the visualization extension allows each to be run independently of the other. In addition, the visualization extension is able to run inside BlueJ as an extension or as an external program.

4.1 Invoking The Visualizer Inside BlueJ

Invoking the visualizer from within BlueJ displays information based on the project that is currently open. This frees the user from having to select a project and allows them quick access to the visualizer anytime during a programming session. Figure 4.1 illustrates how the visualization tool is started from within BlueJ. Using BlueJ's extension mechanism, a menu item for the visualizer is registered on the "Tools" menu. When the visualizer is clicked, a modal window is brought into view. By using a modal view, the user is forced to close the visualizer before being able to do any further work on a project. If a modal view

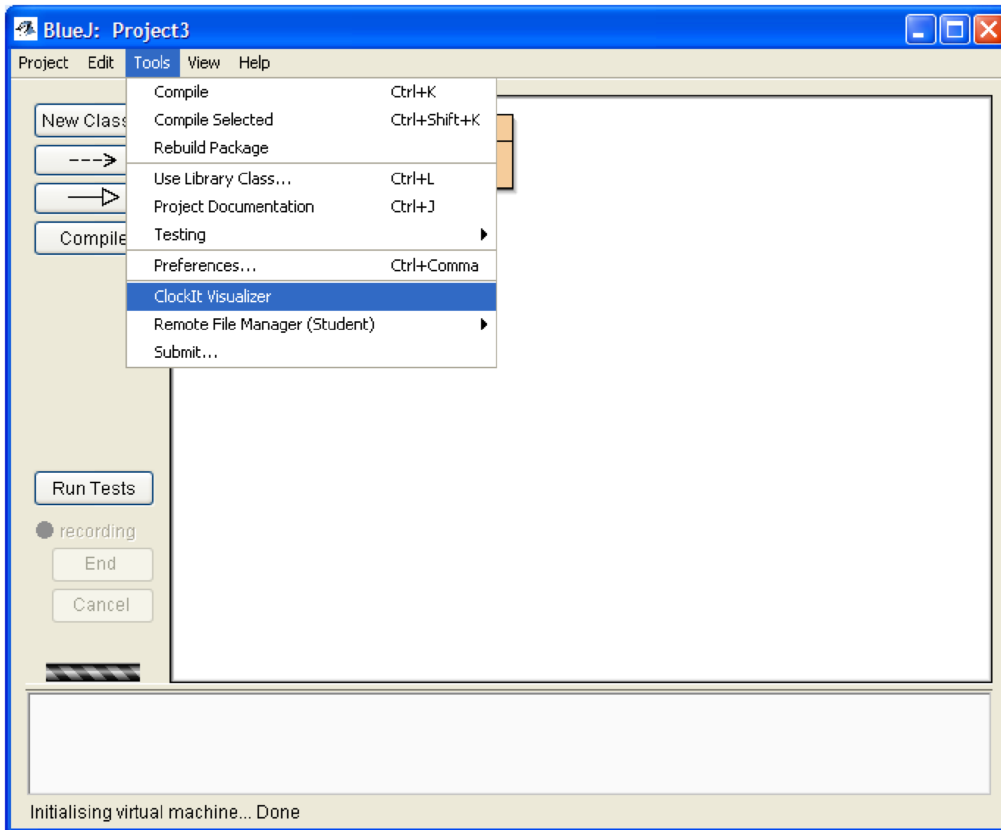

Figure 4.1 – Starting the ClockIt visualizer BlueJ extension

were not used, the user could edit their project while the visualizer is still running which could lead to an inconsistent view in the visualizer. Thus, a modal view relieves the visualizer extension from having to constantly monitor changes to the log file. The visualizer simply reads whatever is in the log file each time it is started.

4.2 External Visualizer

An external visualizer program is provided because viewing a project from within BlueJ means a project must already be opened. Each open generates a project open event. These events may not be desired in the case of an instructor viewing a student's project, or a student opening an old project. By using an external visualize, the possibility of generating extraneous events is eliminated. Included in the external program is a directory searching utility which has the ability to recursively search a top level directory finding all log files

contained under the top level directory. Figure 4.2 shows the external program with the log files contained from a search. Being able to search a directory hierarchy is useful for instances where an instructor or student wants to see all log files for a current class.

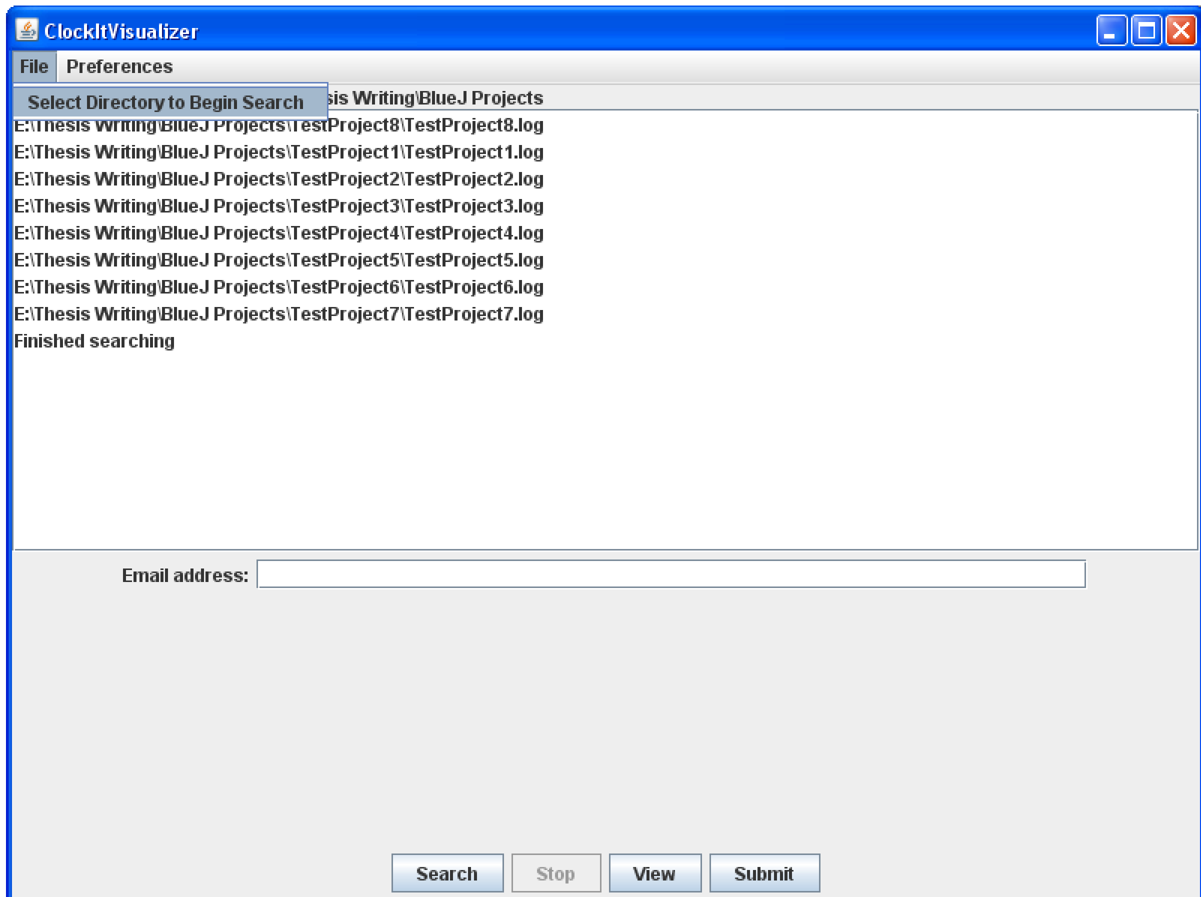

Figure 4.2 – External visualization tool directory search results

4.3 Visualizer Views

The visualizer uses a “tabbed pane” layout to allow users to select various views of a project’s development events and statistics. Once the log file is opened, an in-memory object representation is built, and used by all the panes. The panes consist of a Summary tab, Overview tab, Compilation/Invocation tab, Statistics tab, and a Log File tab. All graphs are created using JFreeChart, a free graphing package for Java (Gilbert & Morgner, n.d.).

4.3.1 Summary Tab

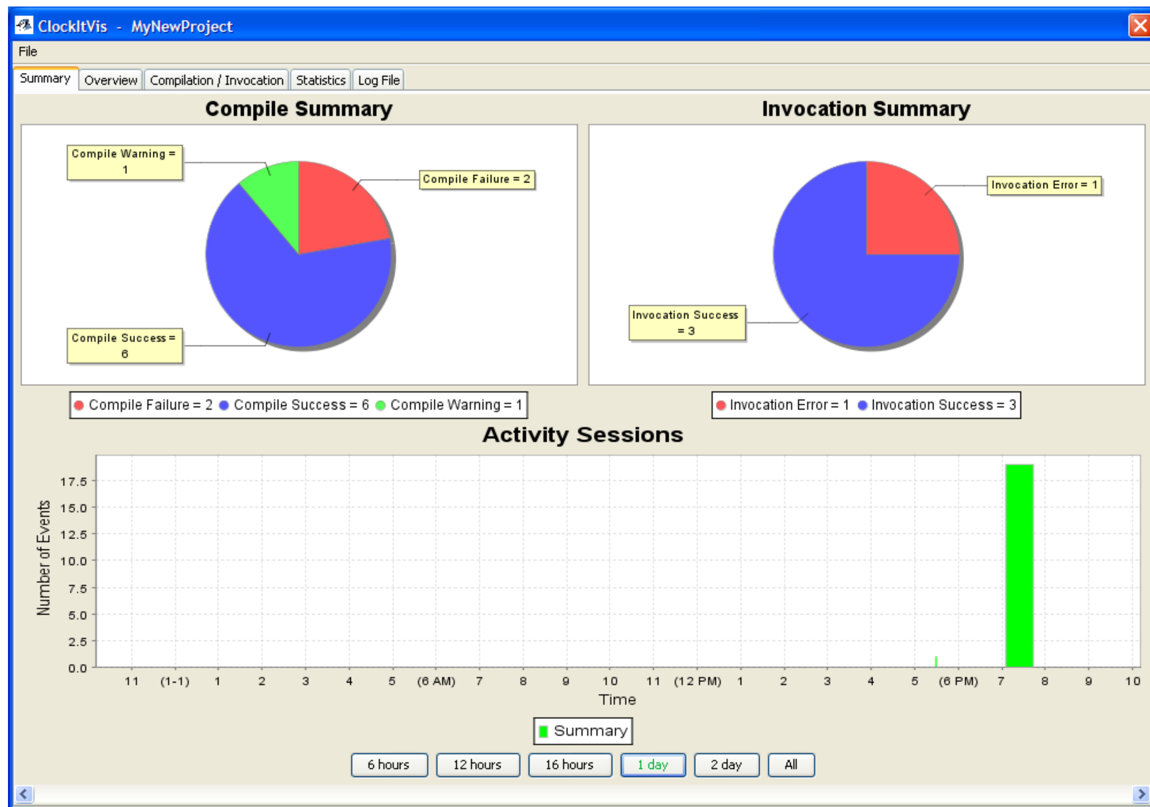

Figure 4.3 – ClockIt visualizer summary tab view

When a user starts the visualization tool, they are presented with a visual summary of their project. Figure 4.3 shows the summary tab for a sample project, and summarizes the compilation events, invocation events, and the distribution of time spent working on the project.

The top-left pie chart summarizes the compilation events that have occurred throughout the project. The chart shows the number of compile warnings (yellow), compile failures (red), and successful compilations (green). This information can be used as an indication of compilation behavior such as the ratio of successful compilations to compile failures. The chart can also be used to determine whether or not the student has compiled often enough as well as give insight into difficulties the student had with the project.

The top-right pie chart summarizes the invocation events that have occurred throughout the project, and can be used as an indication of testing behavior. In order to test or use objects created in BlueJ, they are first created by right-clicking on the object and invoking one of the available constructors. Once the object has been instantiated, it is placed on the BlueJ object bench where methods can be invoked on the object. Invoking either the constructor or methods on an object results in invocation events being generated by BlueJ and logged by ClockIt. The invocation pie chart shows the number of invocation successes and the number of invocation failures. This gives information about run time behavior and gives insight about whether or not a student properly tests their program.

The activity sessions bar graph that occupies the lower half of the screen provides information about how development has progressed throughout the project. The width of a bar in the graph indicates an "activity session." An activity session is a group of events that occur within a specified time interval. The time interval is a user definable value with a default of 60 minutes. For example, consider the following scenario. A student opens his project thus generating open project and open package events. The student then reads an assignment for 50 minutes in which no events occur. Then, the student begins the assignment consisting of edits and compiles for 5 minutes that generate file change and compilation events. Next, the student takes a break for a phone call 50 minutes, then resumes working on the assignment. Now, the student invokes some of the classes in the project for 5 minutes generating invocation events. Assuming a default time interval of 60 minutes, the ClockIt Visualizer will put all the previous events into one activity session.

The height on the y-axis marks the number of events in the activity session. The activity session length coupled with the number of events in the activity session show "event

density," the number of events per unit time. The activity sessions help to understand how a student develops: in short bursts with many events, in long time periods with few events, one long session right before a project is due, etc.

An initial problem associated with JFreeChart was constraining the visible domain in order to see events on the graphs. The time spent on a project can vary greatly based on both the student and the project. When the amount of time from start to finish increases above two days, the visible domain is such that the activity sessions were not able to be seen. The solution was to constrain the domain to the point where events were easily viewable and provide a scroll bar to move through the events should the entire project duration span more than one screen. Buttons are provided to conveniently change the visible domain as needed in order to properly view the data.

4.3.2 Overview Tab

The overview tab shows the project in more detail by showing individual events contained in the activity sessions. Figure 4.4 shows the overview tab for a sample project. On the left side of the screen the development events are separated into three different groups. The groups allow a user to view events on a project, package, or file basis with a tilde mark denoting the default nameless package. Under each group are the activity sessions for the specific group labeled with elapsed time of the session and the number of events in the session. Figure 4.4 shows the selection of second activity session. This activity session has an elapsed time of 67 minutes and consists of 21 events. Since it is in the "All" group it shows events from all files in all packages.

Corresponding buttons are provided to change the size of the visible domain to a suitable size similar to the buttons provided on the Summary tab. In this case, the visible

domain is much smaller as a result of many individual events possibly occurring during a minute. The overview tab shows each event separately and allows a “play-by-play” viewing of events. Using this view an instructor can identify specific behaviors shown by a student. Also, given the arrangement by project, package, and file, an instructor has the ability to identify certain aspects of a project as they change from file to file. A red color is assigned for compilation errors and a lighter color red for invocation errors, other colors are consistent but chosen by the graphing package used.

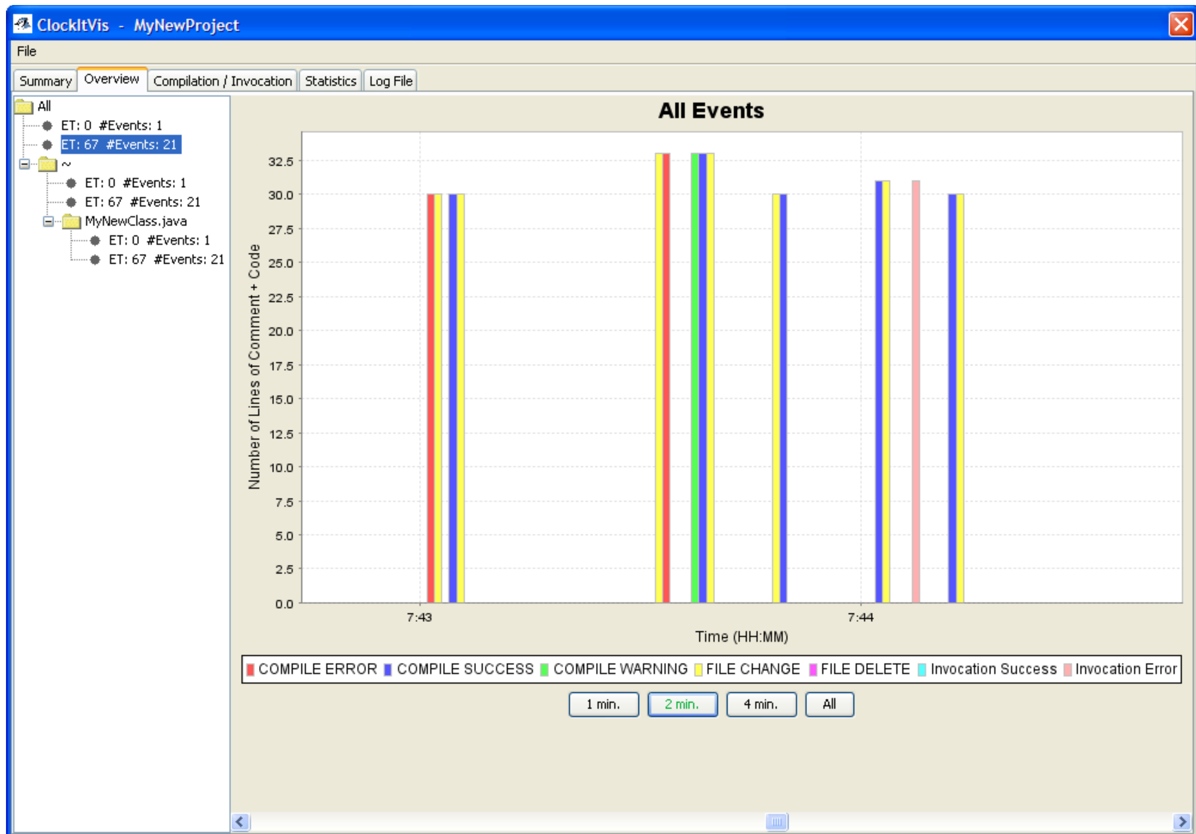

Figure 4.4 – ClockIt visualizer overview tab

4.3.3 Compilation / Invocation Tab

The compilation/invocation tab, shown in figure 4.5, displays information about the project's compilation and invocation events. The pie chart on the left shows the number of compile errors for each file, and can be used to determine which file, if any, caused a developer the most problems. The pie chart on the right shows the number of invocations per object class, and can be used to determine which object classes, if any, have been tested. At the bottom of the screen are compilation error messages along with their frequency. Should there be more error messages than can fit on the screen, a scrollbar will appear. Compilation error messages can help a user identify common problems that are seen throughout the project's development. By considering all the components in the compilation/invocation tab, a user can quickly and easily identify which file caused the most errors, the most prevalent compilation error, and which object received the most testing.

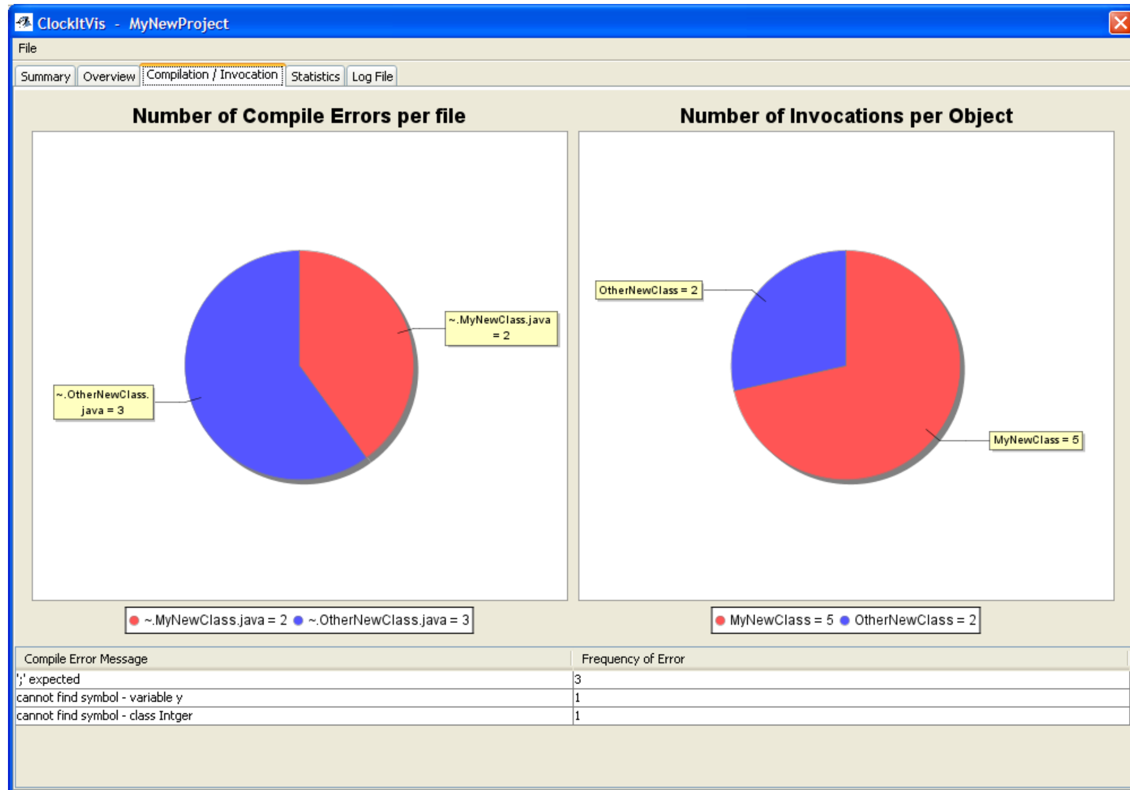

Figure 4.5 – ClockIt visualizer compilation/invocation tab

4.3.4 Statistics Tab

Figure 4.6 shows the statistics tab that reports various numerical pieces of information about the project. The first statistic shown is total project duration, which can be used to verify time estimates related to the project. Project duration can also be used by instructors as an aid in adjusting projects to fit inside a specific time interval. The second statistic, average length of activity session, can be used to profile a user's average time spent developing the project at each sitting. The third statistic provided is the average number of lines changed between successful compiles. This statistic can be used to promote incremental development by encouraging users to keep this number low.

Figure 4.6 – ClockIt visualizer statistics tab

The final statistic, number of times testing an object did not occur after a successful compile, is a good indication of whether or not proper testing occurred. A successful compile

indicates syntactically correct code, but says nothing about the intended behavior. Proper testing after a successful compile ensures correct program behavior, and is essential to good developer habits. The remainder of this window is a listing of the files contained in the project and their corresponding lines of code and lines of comment. It can be used to determine total project size.

4.3.5 Log File Tab

The log file tab, (shown in figure 4.7), is provided as a plain text view of the information that is actually stored in the log file. This tab is useful to anyone wishing to create their own visualization tool.

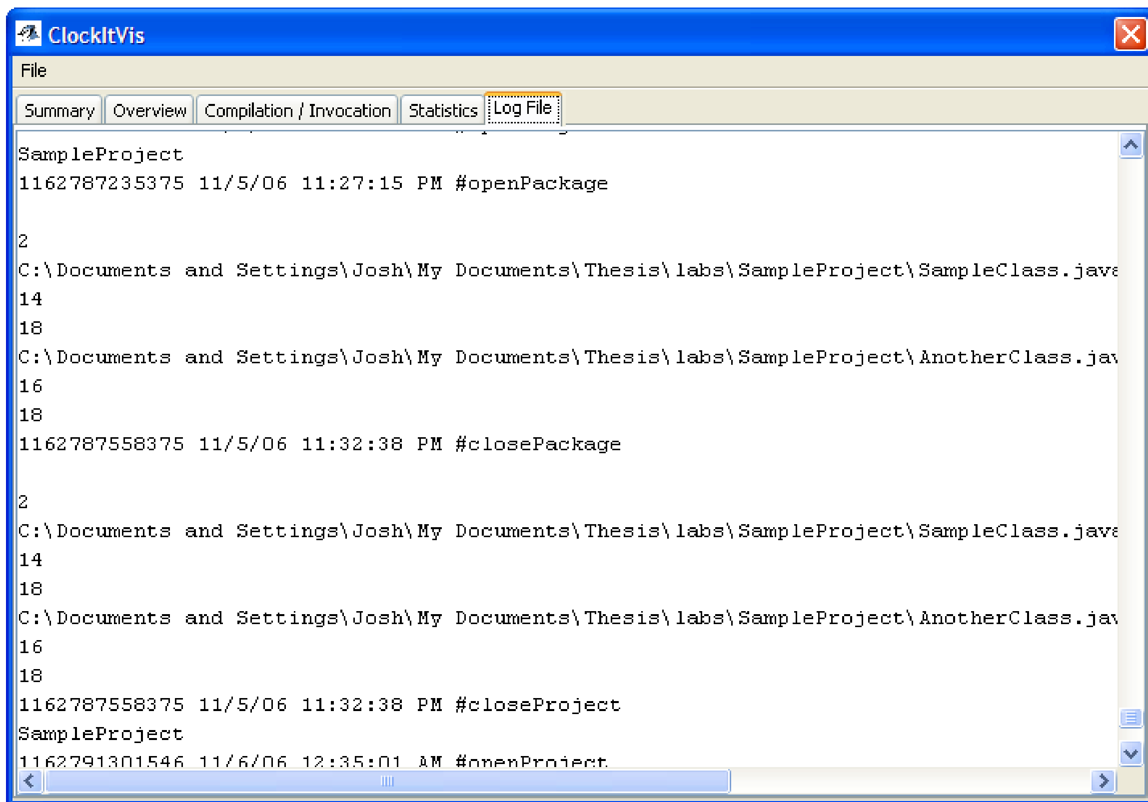

Figure 4.7 - ClockIt visualizer log file tab

Chapter 5 – Results

The ClockIt data logger has been successfully used in introductory courses in the CS department at Appalachian State University each semester since Fall 2007. In the period from Fall semester 2007 through Spring semester 2009 over 147,000 compilation events were logged and uploaded to the central database. Matthew Jadud also built a data logger extension to BlueJ as part of his PhD thesis (Jadud, 2006b). The Jadud BlueJ extension was used to collect 42,000 compilation events during three terms from 2003-2005. Jadud analyzed these events to characterize behaviors of introductory students.

The first two subsections compare our data to Jadud's data using his behavior metrics. The following two subsections discuss new findings discovered in analysis of the ClockIt data.

5.1 Most Common Errors

One of the key results Jadud reports is the straightforward measurement of the frequency of specific compiler errors. Knowing what error is most common can help instructors better understand what misconceptions students may have about programming. Table 5.1 lists the top six compiler errors determined by Jadud. As shown, five of the top six from ClockIt are also present in Jadud's results. Since these errors comprise more than 64 percent of all errors, one can generally conclude that this provides quantitative evidence of the common problems experienced by beginning Java programmers. The differences between the two are likely attributable to variations in assignments. Despite these variations, the errors experienced by beginning Java students are somewhat predictable.

Table 5.1 – Most common errors encountered by students

Compile Error	Jadud Pct.	Rank	ClockIt Pct.	Rank
Missing Semicolon	18.2%	1	9.5%	3
Unknown Variable	12.0%	2	16.2%	1
Bracket Expected	11.9%	3	7.6%	4
Illegal Start of Expr.	8.9%	4	5.6%	5
Unknown Class	7.0%	5	4.0%	7
Unknown Method	6.4%	6	12.6%	2
Total	64.4%		66.2%	

5.2 Successive Compilations

The previous subsection considered compilation events individually. Jadud then investigated the relationship between pairs of successive compilation events. This subsection describes three analyses of successive compilation. First is the time between the events. Secondly, the success or failure of the compilations is considered. Last is a combination of these two perspectives. This discussion presents Jadud's data and ClockIt data, which confirms Jadud's analysis.

5.2.1 Time Differential

Jadud analyzed the amount of time between two successive compilation events. These compilation events are analyzed by grouping them into 10 second bins. For instance, if a compilation event occurs less than 10 seconds after the preceding one, it will be placed in the 10 second bin; if the event occurs 11 to 20 seconds after the preceding event, it will be placed in the 20 second bin, etc. As figure 5.1 shows, the Jadud and Clockit data agree that 50 percent of compilations occur within 30 seconds of the previous compilation. The other significant bin contains about 20 percent of the compilation events and consists of compilation events separated by two minutes or more.

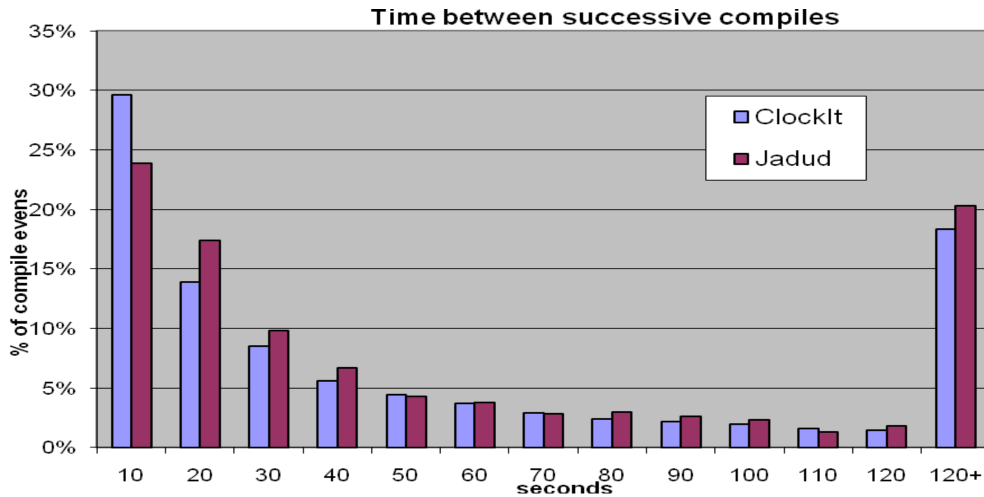

Figure 5.1 – Time between successive compilation events.

5.2.2 Compilation Event Pairs

Why do follow-up compilations occur largely within 30 seconds or after more than 2 minutes? Jadud looked into the nature of the compilation event pair and categorized them into four distinct cases. The nature of a single compilation event is either failure (F) or success (T) thus resulting in four combinations for a pair of compilation events: a failure followed by a failure F>F, a failure followed by a success F>T, a success followed by a failure T>F, and a success followed by a success T>T. Figure 5.2 shows that most pairs are failures following failures. Once again the ClockIt data closely mirrors that of Jadud's.

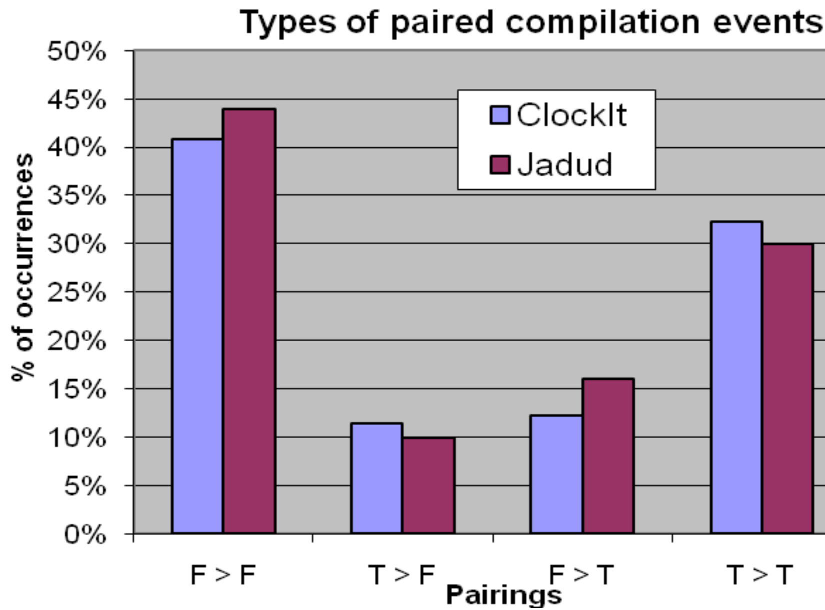

Figure 5.2 – Successive compilation event pairings.

5.2.3 Combined Perspective

Finally, Jadud examined the compilation event pairings when grouped in the 10 second bins. As shown in Table 5.2, this combined view provides quantitative evidence that beginning students attempt to correct errors quickly resulting in high percentages of F>F and F>T in the shorter time bins. The high percentage contained in the last bin for T>F can be explained by the situation where a student reaches a successful compilation and then does not need to compile again because they are testing or working on a different section of code. Evidence to support this claim comes from the events that occur between the successful compilation and the failed compilation. In 48.5 percent of T>F pairings in the 121+ bin, one or more invocation events are also present. The presence of invocation events supports the idea that after a student reaches a successful compilation he will want to test their changes to see if the code produces the desired result. In 13.5 percent of T>F pairings in the 121+ bin, no invocation events occur, but the failed compilation is on a different file than the successful

compilation. This scenario suggests that a student has finished working on one file and is focused on another area of code.

Table 5.2 – Time between compilation event pairings.

	F > F		F > T		T > F	
Time	Jad.	ClockIt	Jad.	ClockIt	Jad.	ClockIt
≤ 10	**26.0%**	**32.6%**	**32.5%**	**41.5%**	5.0%	15.8%
≤ 20	**25.0%**	**19.3%**	**30.0%**	**20.8%**	3.0%	4.1%
≤ 30	**12.6%**	**10.3%**	**12.5%**	**9.1%**	3.0%	3.9%
≤ 40	9.5%	6.2%	8.0%	5.6%	2.0%	3.5%
≤ 50	5.0%	4.6%	4.2%	3.6%	4.5%	3.7%
≤ 60	4.0%	3.6%	3.0%	2.7%	3.9%	3.9%
≤ 70	2.5%	2.7%	2.5%	2.1%	1.0%	3.3%
≤ 80	3.0%	2.1%	2.0%	1.5%	4.0%	3.0%
≤ 90	1.0%	1.9%	2.5%	1.3%	3.0%	3.0%
≤ 100	1.0%	1.6%	1.0%	1.1%	3.5%	2.9%
≤ 110	0.5%	1.3%	0.5%	0.9%	1.5%	2.7%
≤ 120	1.0%	1.2%	1.0%	0.8%	1.5%	2.6%
121+	9.0%	12.6%	4.5%	8.8%	**62.5%**	**47.6%**

5.3 A New Insight About Compilation Errors

Collecting data over multiple assignments throughout a semester provides insight into how a student's programming proficiency changes. As a student progresses one would expect routine skills such as terminating statements with a semicolon to become familiar and thus decreasing the number of compilation errors due to a missing semicolon. Table 5.3 compares compiler error frequency data from assignments in the early third of the semester compared to the latter third. The table provides evidence that these simple syntax errors do in fact decrease. Also seen in Table 5.3 is an increase in unknown classes and methods, which can be attributed to assignments requiring students to write their own methods or classes, or assignments using objects more instead of simply primitive data types.

Table 5.3 – Compile errors, early term vs. late term

Compile error	Early term	Late term
Missing semicolon	10.0%	7.7%
Unknown variable	16.4%	14.8%
Bracket expected	9.3%	8.2%
Illegal Start of Expr.	5.8%	4.3%
Unknown class	3.0%	5.5%
Unknown method	12.6%	16.7%

5.4 Student Cheating

The ClockIt Data Logger can provide clues about student cheating. While the clues are not conclusive as to whether or not a student has cheated, it does provide indirect evidence that may be used to confirm an instructor's suspicions.

Student email addresses are used to identify projects belonging to a specific student. When a project is opened the ClockIt Data Logger prompts for an email address via a standard GUI dialog box (see Figure 3.11). If an email address is already present in the project log file, the email address field in the dialog box is pre-filled. The Data Logger will only log a user name event under two conditions: no email is present in the log file, or the email address entered is different from the one already in the log file. In the case of a different email being entered, the log file will then contain multiple email addresses. These email addresses are associating multiple students with the same project.

Chapter 6 – Summary and Future Work

Currently, projected job growth for computer software engineers and enrollment in Computer Science appear to be inversely related. Job growth is expected to increase and enrollment has been declining or improving only nominally. Factoring in attrition rates with the decline in enrollment creates a void in the capable work force of software engineers. Part of a solution to this void is to monitor how students develop software. This will allow instructors to see what happens between the start of an assignment and when it is turned in. Instructors can intervene before a student gives up on the course and possibly, the major. Monitoring software development can also provide essential feedback to students that can be used in time estimation and defect prevention.

Several methodologies have been created to aid in producing higher quality software by using process monitoring techniques. While these processes have proven to provide higher quality software, there are several key areas that limit their effectiveness, primarily due to the requirement that data be recorded manually. This manual intervention results in poor data quality and significant effort that many feel could be better spent programming.

ClockIt is part of a solution to curb the attrition and better prepare introductory students by creating an extension for the popular pedagogical IDE BlueJ. Using BlueJ's integrated extension mechanisms, ClockIt is a minimally intrusive way to record how a student develops software. The ClockIt extension also uploads data to a central server database. Placing the burden of recording data on the extension allows the student the freedom to focus on programming and ensures accurate data collection.

Analyzing the data recorded for four semesters of introductory course use has shown that ClockIt is a viable means to automatically record how a student develops software. Analysis of the ClockIt data confirms the results of a previous study by Matthew Jadud. Specifically, there is general agreement in Jadud's study and the ClockIt study on the following: compilation error frequency, timings between compilation events, and relationships between pairs of compilation events. In addition, ClockIt provides quantitative evidence of certain compilation errors decreasing during the students' semester-long learning while other errors increase. All of this information can be used by instructors and students to improve their understanding of how students learn to program.

6.1 Future Work

Several areas of improvement and enhancement are available for both the data logger and the visualizer. The data logger could be enhanced to provide more information about what changes occur between subsequent file changes. Providing metrics other than the lines of code and comments could be useful in a deeper understanding of how a student develops software, but care must be taken to not have a negative effect on the performance of the IDE. The BlueJ development team has received requests to provide more event information, and any change of this nature would allow enhanced ClockIt data logging as well.

Changes to the visualizer can include information found by further analyzing the data produced by the data logger. As data analysis continues and improves, it becomes possible to automate interventions within BlueJ. For example, this research has shown that some compiler errors decrease as the student gains proficiency. It seems possible to intervene with alerts for students where certain errors do not decrease as the semester progresses. These

alerts could be for the student with help specific to the error, or an alert can notify an instructor to provide extra help.

Data analysis can be further expanded upon by comparing the programming habits of upper level students to the data from introductory level students to see if certain patterns exist in one group versus the other. These patterns could be used as a benchmark to determine whether or not introductory level students are progressing through a course like they should. Further insight into what makes a student successful can be studied by augmenting project data with student grades. Combining information from the events in a log file with time information such as length of project, or activity sessions may yield a statistically significant predictor for student performance that could be beneficial in identifying weak students earlier rather than later.

Bibliography

Allen, E., Cartwright, R., & Stoler, B. (2002). DrJava: A lightweight pedagogic environment for Java. *ACM SIGCSE Bulletin, 37* (1), 137-141. DOI: 10.1145/563517.563395

Apache Software Foundation. (n.d. a). *Jakarta Commons HTTP Client.* Retrieved from http://hc.apache.org/httpclient-3.x/index.html

Apache Software Foundation. (n.d. b). *Apache Tomcat.* Retrieved from http://tomcat.apache.org/

Apache Software Foundation . (n.d. c). *Commons File Upload.* Retrieved from http://commons.apache.org/fileupload/

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin, 37* (2), 103-106. DOI: 10.1145/1083431.1083474

Bruce, K. B. (2005). Controversy on how to teach CS 1: A discussion on the SGICSE members mailing list. *ACM SIGCSE Bulletin, 37* (2), 111-117. DOI: 10.1145/1083431.1083477

Bureau of Labor Statistics, U.S. Department of Labor. (2010). *Occupational Outlook Handbook, 2010-11 Edition, Computer Software Engineers and Computer Programmers.* Retrieved from http://www.bls.gov/oco/ocos303.htm

Chidamber, S. R., & Kemerer, C. F. (1998). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering, 20*, 476-493. DOI: 10.1109/32.295895

Disney, A. M., & Johnson, P. M. (1998). *Investigating data quality problems in the PSP.* Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 143-152. DOI: 10.1145/288195.288292

ETSU. (n.d.). *East Tennessee State University Design Studio.* Retrieved from http://csciwww.etsu.edu/psp/

Ferguson, R., Grissom, S., & Berberoglu, K. (2000). A time management & feedback tool for students in programming courses. *Journal of Computing Sciences in Colleges, 16* (1), 101-110.

Gilbert, D., & Morgner, T. (n.d.). *JFreeChart*. Retrieved from http://www.jfree.org/jfreechart/

Hackystat. (n.d.). *Hackystat.* Retrieved from http://code.google.com/p/hackystat/

Hsia, J. I., Simpson, E., Smith, D., & Cartwright, R. (2005). Taming Java for the classroom. *ACM SIGCSE Bulletin, 37* (1), 327-331. DOI: 10.1145/1047124.1047459

Humphrey, W. S. (1995). *A discipline for software engineering*. Reading, MA: Addison Wesley.

Humphrey, W. S. (1996). Using a defined and measured personal software process. *IEEE Software, 13* (3), 77-88.

Humphrey, W. S. (1997). *Introduction to the Personal Software Process.* Reading, MA: Addison Wesley.

Jadud, M. C. (2006, September). *Methods and tools for exploring novice compilation behavior.* Paper presented at the second international computing education research workshop, University of Kent, Cantebury, UK.

Jadud, M. C. (2006b). *An exploration of novice compilation behavior in BlueJ* (Unpublished doctoral thesis). University of Kent, Cantebury, UK.

Johnson, P. M. (1999). *Leap: A "Personal Information Environment" for software engineers.* Proceedings of the 21st International Conference on Software Engineering, IEEE Computer Society.

Johnson, P. M., Moore, C. A., Dane, J. A., & Brewer, R. S. (2000). Empirically guided software effort guesstimation. *IEEE Software* , 51-56.

Johnson, P. M., Kou, H., Agustin, J., Chan, C., Moore, C., Miglani, J., et al. (2003). *Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined*. Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society.

Marmoset. (n.d.). *The Marmoset Project.* Retrieved from http://marmoset.cs.umd.edu/

McKeogh, J., & Exton, D. C. (2004). *Eclipse plug-in to monitor the programmer behavior.* Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology. DOI: 10.1145/1066129.1066148

Moore, C. A. (2000). *Lessons learned from teaching Refelctive Software Engineering using the Leap Toolkit.* Proceedings of the 22nd International Conference on Software Engineering. DOI: 10.1145/337180.337508

O'Connor, R., & Coleman, G. (2002). *Strategies for Personal Software Process improvement a comparison*. Proceedings of the 2002 ACM Symposium on Applied computing, 1036-1040. DOI: 10.1145/508791.508992

Reid, K. (2007). *ClockIt: Implementation and evaluation* (Unpublished master's thesis). Appalachian State University, Boone, NC.

Reis, C., & Cartwright, R. (2004). Taming a professional IDE for the classroom. *ACM SIGCSE Bulletin, 36* (1), 156-160. DOI: 10.1145/1028174.971357

Shaffer, S. C. (2005). A brief overview theories of learning to program. *Psychology of Programming Interest Group Newsletter,* November 2005.

Taulbee Survey, Computing Research News. (2009). *2007-2008 Taulbee Survey*. Retrieved May 5, 2009, from http://archive.cra.org/statistics/survey/0708.pdf

The Software Process Dashboard. (n.d.). *The Software Process Dashboard Initiative.* Retrieved from http://www.processdash.com/

Biographical Information

Joshua Joel Rountree was born on April 20, 1981 to Joel and Patsy Rountree. He attended Grover Elementary School, Kings Mountain Middle School, and Kings Mountain High School graduating in the summer of 1999. He attended Appalachian State University graduating in the Fall of 2003 with a B.S. in Computer Science. He began Graduate School in the Spring of 2004. He took some time away from graduate school to work full time at LifeStore Financial Group, formerly known as AF Financial Group, located in West Jefferson, NC. He currently works full time for LifeStore Financial Group. He will graduate with a M.S. in Computer Science in December of 2010. His permanent mailing address is PO Box 337, Grover, NC 28073.