

ClockIt: Collecting Quantitative Data on How Beginning Software Developers Really Work

Cindy Norris, Frank Barry, James B. Fenwick Jr, Kathryn Reid and Josh Rountree
Department of Computer Science
Appalachian State University
Boone, NC, USA
can@cs.appstate.edu

ABSTRACT

The Information Technology sector is suffering from a dramatic reduction in the number of students studying the field and subsequently entering the IT market. The number of freshmen expressing “interest in CS” has dramatically decreased since 2000 [16] and CS attrition rates are very high [3]. As part of an effort funded by the National Science Foundation (DUE 0633640), this paper introduces the ClockIt toolset that we believe can be used to help educators understand and reduce the high attrition rates of CS 1 and CS 2 students. Using ClockIt, we can unobtrusively monitor and log student software development activities allowing us to determine what practices make a student a successful software developer and what practices do not.

Categories and Subject Descriptors

K.3.1 [Computers and Education]: Computer Uses in Education—*Computer Assisted Instruction (CAI)*

General Terms

Measurement, Human Factors

Keywords

CS1, CS2, assessment, programming practices

1. INTRODUCTION

The Information Technology sector, and Computer Science in particular, is suffering from a dramatic reduction in the number of students studying the field and subsequently entering the IT market. A recent Taulbee Survey reports that the number of bachelor’s degrees granted in CS continues to drop [15]. In addition, the number of freshmen expressing “interest in CS” has dramatically decreased since 2000 [16]. Another CS problem is retaining the students who do enter the field. Cohoon and Chen report that CS attrition was on the rise even before the enrollment swells

of the late 1990’s [3]. Recent attrition rates as high as 66% are reported [12].

The students who are likely to stay in the major are those students who succeed in the introductory programming courses, typically known as CS 1 and CS 2. However, success is more than just a passing grade; it also includes a student’s confidence in their ability to continue to succeed. Most educators would probably agree that the top students in their classes already believe they will succeed and generally do. These students are the ones we are currently retaining. Our concern is with retaining students whose work habits may be causing them to become attrition statistics. Shaffer summarizes several theories about how programming is learned [13]. While there is clearly a cognitive component to learning programming, there is also an important process component. That is, some aspects of programming can be taught in a behavioral fashion.

We believe that part of the multi-faceted issues surrounding capable but unsuccessful students is related to effective programming work habits. We recently conducted surveys of CS students and CS faculty at our institution on software development strategies. The goals of the survey were twofold. First, we sought to gain insight into student and faculty perceptions of student software development strategies such as whether students started early on assignments and worked on them incrementally. Second, we were interested in determining whether information collected by our data collection tools would be considered valuable. The students and faculty surveyed could respond with either strongly agree, agree, neutral, disagree or strongly disagree.

Table 1 summarizes the results of the survey omitting the neutral answer. The majority of students suggested that they *test thoroughly*, *compile frequently*, *develop code incrementally*, and *rewrite messy sections of code*. However, most faculty believed their students do not consistently employ these strategies. For example, 92% of CS 1 and CS 2 students believe they compile code frequently and fix compilation errors before continuing development while only 25% of the faculty surveyed believe this.

This disparity between student and faculty perceptions is an indication of the profound lack of understanding of student software development practices. Faculty complaints are universal: “students start too late,” “they think if the code compiles, it works properly,” “students type in huge amounts of code before compiling or testing anything,” “students don’t bother to design.” However, these complaints are largely anecdotal; little data has been collected about the actual development practices of students, particularly

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ITiCSE’08, June 30–July 2, 2008, Madrid, Spain.

Copyright 2008 ACM 978-1-60558-115-6/08/06 ...\$5.00.

	Student Responses		Faculty Responses	
	Agree or Strongly Agree	Disagree or Strongly Disagree	Agree or Strongly Agree	Disagree or Strongly Disagree
Compile code frequently and fix compilation errors before continuing development	92%	2%	25%	68%
Develop a design for a project before coding	45%	12%	17%	83%
Implement, test, and debug one piece of code (class or method) at a time	68%	6%	25%	50%
Start on projects early and work on them incrementally up until due date	42%	30%	0%	75%
Test their code thoroughly before turning in	80%	4%	8%	42%
Write modular and well organized code	80%	6%	17%	42%
Recode messy or poorly designed parts of project	68%	6%	8%	75%

Table 1: Student and faculty responses to questions about development practices

<i>Would the following info be helpful for introductory CS students?</i>	Student Responses	Faculty Responses
	Agree or Strongly Agree	Agree or Strongly Agree
The amount of time spent on project	60%	42%
Number of compilations	38%	33%
Types of compilation errors encountered	77%	75%
Time of day when most work was accomplished on a project	32%	17%
How a student's habits compare to other students at the same level	74%	83%
How a student's habits compare to other students at a higher level	58%	83%

Table 2: Perceived value of programming habits

introductory students, and we are only guessing about what practices make a student a successful software developer and what practices do not.

Table 2 summarizes the results of questions about what types of information regarding programming habits could benefit introductory students. Of particular interest are the last two questions about the perceived value in a quantitative comparison with other student practices. Both faculty and students agree this information would be valuable.

This paper introduces ClockIt, our data collection and analysis toolset allowing us to measure the software development practices of introductory CS students and visually compare the practices of these students to one another as well as more experienced developers such as graduate students, faculty, and members of industry. Ultimately, we believe that an empirically based understanding of good software development habits will improve teaching at the CS 1 and CS 2 level, increasing the likelihood of student success, and thereby improving retention and we plan to discuss these results in a future paper. However, we anticipate some immediate ancillary benefits of our ClockIt toolset. For example, our data collection toolset can be used to determine when a student begins an assignment, the time period over which the work is distributed, and the amount of time it takes to complete an assignment. This information could be used to identify students that are not being sufficiently challenged as well as those students that are spending inordinately large amounts of time on an assignment.

Our ClockIt toolset consists of two components: a data logger/visualizer and a web interface. Currently, we have a data logger/visualizer associated with the BlueJ IDE and an Eclipse data logger/visualizer is expected to be completed by conference presentation. The remainder of this paper

discusses our BlueJ Data Logger/Visualizer, ClockIt Web Interface and preliminary results.

2. BLUEJ DATA LOGGER AND VISUALIZER

BlueJ is a Java development environment designed for teaching object-oriented programming concepts to introductory Computer Science students [2, 10]. BlueJ is a front end to various versions of Sun's Java Development Kit and provides tools for editing, debugging, constructing class diagrams, instantiating objects, testing methods, and creating HTML (javadoc) documentation.

BlueJ also provides an extension mechanism allowing third-party developers to add customizations to the environment. An extension is created by first subclassing the Extension base class provided with BlueJ and overriding the startup method. The startup method accepts a BlueJ proxy object as a parameter which provides methods for registering event listeners. In addition, BlueJ allows menu items to be added to the tools menu, class menu object menu, and allows the preference panel to be modified.

The ClockIt BlueJ Data Logger/Visualizer is implemented as two separate extensions: the data logger extension captures events that occur during software development and adds them to a log file, and the data visualizer extension displays graphs of the data captured by the logger in an on-demand fashion.

2.1 BlueJ Data Logger

The ClockIt BlueJ Data Logger extension registers event listeners for compile events, package events, and invocation events. This extension along with custom Data Logger file monitoring are used to detect software development events

to add to the log file. The log file can contain 10 types of events: project open, project close, package open, package close, compilation success, compilation error, compilation warning, invocation event, file change, and file delete.

When a programmer uses BlueJ to work on a project, she first uses the BlueJ menu to create a new project. BlueJ creates a directory to maintain the project and generates a package open event. Our data logger extension creates a log file to be stored in the project directory. In addition, the data logger adds project open and package open events to the log file. Each of these events contains an event identifier, a time stamp and the name of the package or project opened. A package open event also identifies the number and size of each file in the package. Additional package opens within the same project will cause package open (but not project open) events to be added to the log file. When the programmer manually closes a package or exits BlueJ, package close events are added to the log file. The last event in the log file for each BlueJ session is a project close.

Compilation events (compilation success, compilation error, or compilation warning) are fired in response to a programmer requested compile. Compilation event information stored in the log file consists of an event identifier and information about each file involved in the compilation. In the case of a compilation error, the error message and error location are also stored. Note that BlueJ suspends compilation when the first compile error is found thus only a single error is indicated with the compilation error event. Similarly, compilation warning event information contains the compile warning message and location of the warning.

Invocation events are generated when the programmer uses the BlueJ IDE to instantiate an object or invokes a method on an instantiated object in BlueJ's "object bench." Information stored with the invocation event consists of the object class, method invoked and result of the invocation (exception, forced exit, normal exit, terminated exit, or unknown exit). Recording invocation events allows us to determine whether students are performing testing at this level.

When a package is opened within BlueJ, our data logger extension creates a thread to monitor all files in the package. When a monitored file changes in size (by editing or deleting the file), the data logger adds a file change (or file delete) event to the log file. This event indicates the event type, the name of the file and the number of lines of comment and code. Maintaining this information can give us an idea about how much time and effort a student has spent editing a file before performing a compile.

Upon exit of BlueJ, our data logger extension automatically submits the log file to a log file parser that parses the file and stores the information in a MySQL database. A programmer may exit BlueJ many times over the course of project development; the parser handles this by updating the database with only the new information in the log file. If the student uses BlueJ while not connected to the internet, the collected data is submitted the next time the student works on the project while connected to the internet. The ClockIt web interface discussed in section 3 allows faculty members to access the information collected by the data logger on a per student or per class basis. Students may also access data collected on their own projects.

2.2 BlueJ Data Visualizer

The ClockIt BlueJ Data Visualizer can be invoked within



Figure 1: ClockIt BlueJ Data Visualizer summary

BlueJ to view the data collected about the current project. The visualizer uses JFreeChart to provide the user with a visual representation of the events that have occurred thus far during project development. The user can select from various views to get a high level or a detailed view of their activities. For example, Figure 1 shows the summary page for a sample project. The summary contains pie charts representing the number of compilation and invocation occurrences and results. In addition, an activity session graph displays information about how development has progressed throughout the project. The user can see how many events have occurred within various time spans: a specific hour, six hours, 12 hours, 16 hours, 1 day, 2 days or since project creation. This information represents how a student develops a project over time: short bursts of many events, long periods with few events, one long session before due date, etc.

Other tabs provide more specific information. For example, the overview tab displays a "play by play" view of events, identifying each event that occurs within a specific time frame. The user can see how much code is written between compiles, whether/when testing occurs, how much code is being developed at particular points in time during the course of the project, etc.

The BlueJ Data Visualizer provides an abstract visualization of a student's development for only the project currently open. As such, it is not particularly convenient for the instructor to view log file data for several students or projects this way. Moreover, using the BlueJ Data Visualizer to see multiple visualizations simultaneously would require multiple instances of BlueJ running. Instead, we provide a more convenient way for instructors to peruse the visualizations of log file data, as well as class averages of log file data, via the ClockIt Web Interface discussed in the next section.

3. CLOCKIT WEB INTERFACE

The ClockIt web interface supplies access to the ClockIt MySQL database through a web browser and allows the user to view computed measurements of software development habits obtained from submitted log files in a graphical form. A user can be a student with access to his own data, an instructor with access to the data of students in her courses, or an administrator with access to all data. A student can

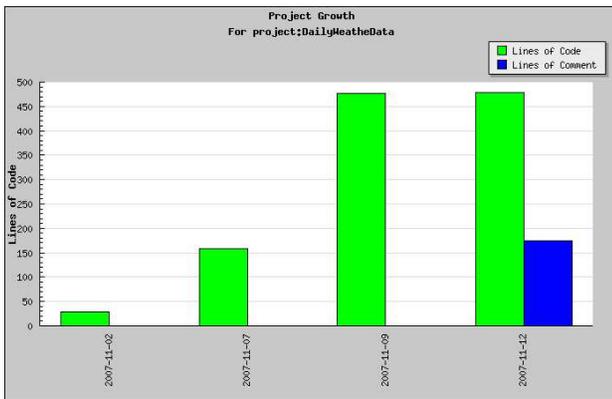


Figure 2: ClockIt Web Interface Project Size Graph

use the web interface to select a single project and their view will be very similar to what is provided by the BlueJ Data Visualizer. Instructors can view data for a project completed by any of their students or view class average data.

The web interface displays graphs about the following types of data: compilation, time, project growth, testing, and project overview. The compilation page displays information about the number and types of compilation errors on individual students and class averages. For example, our preliminary results indicate that the most common compilation error made by students is “symbol expected” likely due to the omission of a semicolon or brace. Another common student error is “missing return statement.” The percentage of unsuccessful compiles is also displayed.

The time distribution graphs indicate the amount of time a student spent per day on a given project (omitting days where there was no activity). An instructor can also view the average amount of time spent per day on a project over all students in a class. This information can help determine which students start late, if work was completed incrementally, whether the assignment is too difficult or too easy.

It is interesting to compare the time distribution graphs to the project growth graphs. Project growth is represented as the number of lines of code and lines of comments at the end of each day during project development (omitting days on the graph where there was no activity). From this graph we can see when the student began work on the project and the number of lines of code and comment produced per day. Figure 2 shows a project growth chart displayed by our ClockIt web interface. The x-axis identifies days the student worked on the project. For each day, there are two bars. The left bar is green and represents lines of code. The right bar is blue and represents lines of comments. The y-axis identifies the number of lines in the project files. In the figure, we can see this student worked on four different days on the project and that they did all the commenting on the last day. The project growth graph can be viewed along with the time distribution graph to determine the number of lines of code produced per unit time.

The testing visualization displays the number of invocations of methods in the project. This information is displayed at the method level (list of methods invoked and the number of invocations), the class level (list of classes with methods invoked and the number of invocations), the package level and the project level. The project level view

provides an overall indication of the amount of testing. The class level view provides an indication of the amount of time spent testing individual classes and likely the difficulty of implementing a class.

Like the overview visualization provided by the BlueJ Data Visualizer, the overview visualization provided by the web interface gives a “play by play” view of events.

4. PRELIMINARY RESULTS

We began monitoring students in three sections of CS 1 during the Fall 2007 term. 75 students and 3 instructors consented to participate in the ClockIt study. Other than signing consent forms, the only intrusion experienced by students using BlueJ with our BlueJ Data Logger extension is the requirement of entering an email address when a project is created. The email address is used to link the data submitted to a particular student in the database. Students choosing not to participate were not added to the database causing their data submission attempt to be ignored.

Most of the data collected was during closed laboratory sessions so the assignments were smaller; nonetheless, some useful observations can be made. We chose three students from a CS 1 class to indicate the type of information we can collect. These are summarized in Table 3. The student with id 1 received the highest grade on the project, spent the most time on the assignment, but didn’t write the largest amount of code. The student with id 2 spent nearly as much time as the student with id 2, and produced more (although non-working) code, earning a D on the assignment. The student with id 3 spent the least amount of time on the assignment, produced little code, and received the lowest grade. One interesting point to note is that the percentage of compilation errors and the type of compilation errors appears to relate to the student’s performance on the project. For example, 66% of the time the best student invoked a compile, the compile resulted in an error; the worst student encountered a compile error 87% of the time. In addition, the poorest performing student encountered compilation errors that the better students were not making, leading us to question whether the student had properly mastered early course material such as how to properly form a method. Whether these students are representative of other students in the class and the application of this information to teaching remains to be done and will be the subject of a future paper.

5. RELATED WORK

Certainly, other researchers have explored collecting metrics during software development. Our approach distinguishes itself by being fully automatic as well as focusing on tools useful for improving our understanding of beginning software developers. Johnson [8] divides the approaches for metrics for developers into three generations. The first generation approach is manual PSP (Personal Software Process) [6]. Users create and print forms to log data on time, size and defects. Next, other forms are used to calculate project estimation and quality assurance. The second generation approach uses a tool for collecting data and calculating PSP-style metrics. PSP Dashboard [1], Leap [11], PSP tool [4] and PSP studio [5] are examples of second generation tools. These tools provide dialog boxes that allow the user to record time, size and defect information and perform various analyses as requested. These tools reduce, but

Student Id	Assignment Grade	Time (min)	Size (lines)	Failed Compilations Percent	Number	<i>is already defined</i>	<i>incomparable types</i>	<i>invalid method declaration</i>
1	100	240	130	66%	116	0%	0%	0%
2	60	210	155	78%	50	2%	0%	0%
3	12	85	44	87%	20	10%	5%	15%

Table 3: Preliminary analysis of collected data

do not eliminate, the overhead required to collect the metrics. Third generation approaches eliminate the overhead of collecting data for analysis because this data is collected automatically. Thus, the user need not context switch between development and the PSP forms or tool. Hackystat [8], Jadud’s work [7], the Eclipse Watcher software [9] and our ClockIt tool are third generation approaches.

Hackystat [8] is a software metric collection tool in which sensors are attached to development tools such as emacs, Junit, Ant and JBuilder. These sensors gather information at regular intervals and send the collected data to a server. On the server, analysis programs are run regularly using the metrics information collected about each developer. Since Hackystat requires the developer to install sensors, we believe Hackystat is not suitable for introductory students.

Jadud [7] has undertaken a quantitative, empirical analysis of introductory programmer compilation behaviors using an extension to BlueJ. Jadud’s work is closely aligned with our own. The most significant difference is that his work focused on compilation events whereas we track additional event data. Our work also monitors student behavior across project development whereas Jadud’s work focuses on single BlueJ sessions. Like Jadud’s work, the Eclipse Watcher [9] tool captures data collected by an plug-in to the Eclipse IDE during a single Eclipse session. The Eclipse Watcher captures every Eclipse event and stores events in an xml file associated with the session.

Marmoset [14] automatically commits student code to a software repository as a “snapshot” at each file save event. Normally, project correctness is verified via a test suite when a student explicitly runs the test suite. However, the snapshots allows for finer-grained statistical analysis of student project development history.

6. CONCLUSIONS

We know that faculty and student perceptions of software development practices do not agree. It is our goal to gather quantitative data of actual student practices. Thus, we have developed the ClockIt toolset that unobtrusively monitors student software development practices. Currently our toolset monitors programs written using the BlueJ IDE, but a plug-in for Eclipse is expected by the time of conference presentation. Using the Data Visualizer or web interface, students and instructors can view collected data. Preliminary analysis of CS 1 data seems to provide some insight regarding student software development practices.

Work is continuing on the Eclipse plug-in and further analysis of collected data. Our goal is to discover “patterns” of student practices that are shared by successful students but not by struggling or unsuccessful students. Understanding these patterns will then allow a variety of interventions to increase a student’s likelihood of success in introductory programming courses. Success in early courses may keep students interested in CS and reduce attrition.

7. REFERENCES

- [1] Software process dashboard. <http://processdash.sourceforge.net>.
- [2] D. J. Barnes and M. Kölling. *Objects First with Java: A Practical Approach using BlueJ*. PrenHall, 2006.
- [3] J. M. Cohoon and L. Y. Chen. Migrating out of computer science. *Computing Research News*, 15(2), March 2003.
- [4] R. Ferguson, S. Grissom, and K. Berberoglu. A time management & feedback tool for students in programming courses. In *Proceedings of the seventh annual CCSC Midwestern conference on Small colleges*, 2000.
- [5] J. Henry. Personal software process studio, 1997. <http://csciwww.etsu.edu/psp>.
- [6] W. S. Humphrey. *Introduction to the Personal Software Process*. Addison-Wesley Longman Inc., 1997.
- [7] M. Jadud. A first look at novice compilation behavior using BlueJ. 16th Workshop of the Psychology of Programming Interest Group, 2005.
- [8] P. M. Johnson, H. Kou, J. Agustin, C. Chan, C. Moore, J. Miglani, S. Zhen, and W. E. J. Doane. Beyond the personal software process: metrics collection and analysis for the differently disciplined. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 641–646, Piscataway, NJ, May 3–10 2003. IEEE-CS.
- [9] J. McKeogh and D. C. Exton. Eclipse plug-in to monitor the programmer behaviour. In *OOPSLA’04 Eclipse Technology eXchange (ETX) Workshop*, 2004.
- [10] A. P. Michael Kolling, Bruce Quig and J. Rosenberg. The Bluej system and its pedagogy. *Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology*, 13(4), 2003.
- [11] C. A. Moore. Lessons learned from teaching reflective software engineering using the leap toolkit. In *ICSE*, pages 672–675, 2000.
- [12] D. Penniman. CRA outline of CS overview. July 2003. <http://www.cra.org/Activities/itdeans/penniman.pdf>.
- [13] S. C. Shaffer. A brief overview of theories of learning to program. *Psychology of Programming Interest Group Newsletter*, November 2005.
- [14] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software repository mining with marmoset: An automated programming project snapshot and testing system. In *MSR ’05: Proceedings of the 2005 international workshop on Mining software repositories*, 2005.
- [15] J. Vegso. Drop in CS bachelor’s degree production. *Computing Research News*, 18(2), March 2006.
- [16] J. Vegso. Freshmen interest in CS and degree production trends. October 2007. <http://www.cra.org/wp/index.php?p=126>.