Recording and Analyzing In-Browser Programming Sessions

Juha Helminen[*] , Petri Ihantola, and Ville Karavirta
Department of Computer Science and Engineering
Aalto University
Finland
juha.helminen@aalto.fi, petri.ihantola@aalto.fi, ville@villekaravirta.com

ABSTRACT

In this paper, we report on the analysis of a novel type of automatically recorded detailed programming session data collected on a university-level web programming course. We present a method and an implementation of collecting rich data on how students learning to program edit and execute code and explore its use in examining learners' behavior. The data collection instrument is an in-browser Python programming environment that integrates an editor, an execution environment, and an interactive Python console and is used to deliver programming assignments with automatic feedback. Most importantly, the environment records learners' interaction within it. We have implemented tools for viewing these traces and demonstrate their potential in learning about the programming processes of learners and of benefiting computing education research and the teaching of programming.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Human Factors

Keywords

Computing Education Research, Computer Science Education, Web Based Programming Environment, Programming Assignment, Programming Session, Python

1. INTRODUCTION

With the onset of MOOCs and the general trend towards web-based cloud computing, it is important to study which kinds of programming environments can be implemented in this mode of computing and what new opportunities it may offer in terms of computing education research. A web-based programming environment has, among others, the advantage of presenting learners with a common working environment that can be managed and updated in a rapid and centralized fashion. In MOOCs, this is one possible approach to creating a common classroom environment when participants are geographically dispersed and obviously cannot have access to shared lab equipment and, in fact, may not even always have adequate control over the computer they are using for their studies. Furthermore as we have control over the environment, we are actually, with relative ease, able to gain access to how learners work when solving programming assignments. Indeed, it is interesting and useful to study how and to what extent this could enable us to examine learners' working habits and processes without direct physical intervention such as human or video observation, in real time, and on a large scale. It would be valuable to get insight into how students work, identify struggling students, and give feedback on their process while discouraging bad practices right as they are working on the problem. Following this, in our research we are interested in:

- How to collect accurate data, on a large scale and with minimal effort, on what students actually do when they are programming?
- How to use this data to learn, in quantifiable means, about the difficulties that learners face and the behaviors they, for good or bad, exhibit?

In this work, we contribute to answering the questions listed by

- presenting an implementation of tools for collecting and viewing detailed data about Python programming sessions and
- demonstrating their use in analyzing students activities, such as, examining how they made use of the integrated interactive console and presenting common exceptions encountered by students.

2. RELATED WORK

With today's computing environments it is quite feasible to keep detailed logs of the interaction in software applications. In terms of programming environments, this means, for example, recording how and when the developer edits and executes code. Indeed, there is growing interest in research on analyzing automatically recorded programming sessions as a means of gaining insight into programming processes.

[*]Corresponding author.

Koli Calling '13, November 14–17, Koli, Finland
Copyright 2013 ACM 978-1-4503-2482-3/13/11 ...$15.00.
http://dx.doi.org/10.1145/2526968.2526970

Early Work

There exists already a long history of research dating back to the 80s that has tried to identify the kinds of mistakes students learning to program are prone to make and how their time is spent during development activities. Spohrer and Soloway used an instrumented version of the VAX 750 operating system to record each syntactically correct program compiled on an introductory Pascal programming course [21, 22]. In an in-depth analysis of the first syntactically correct programs for each of the around 50 students in 3 different problems, they concluded that just a few types of bugs accounted for a majority of the mistakes in students' programs, the implication for educators being that they can most effectively improve their students' performance by changing instruction to address and eliminate the high-frequency bugs.

Java Compiler Errors

Decades later, Jackson et al. recorded compiler errors during one semester with their custom-built Java IDE at the US Military Academy [7]. All errors from the on-site systems were collected including data from 583 students and 11 faculty members. They noted similarly that the top ten types of errors represented over half of the total number collected. Furthermore, they reported that there was a discrepancy between the errors they most observed with students and what faculty had believed to be the most common, thus adding to the value of this data. An interesting feature was that any cadet could access a web page with a detailed analysis of their most common errors.

Ahmadzadeh et al. also collected Java compiler errors, but with attached timestamps and source code, using an instrumented version of the Jikes compiler on a CS1 course as students solved 15 exercises in the JCreator environment [1]. They divided the errors into three categories: syntax errors dealt with the grammar of the language, semantic errors with the meaning of the code being inconsistent, e.g. using a non-static global variable inside a static method, and lexical errors with unrecognized tokens. They found the distribution in their data to be 63 % semantic error, 36 % syntax errors, and 1 % lexical errors. They also noted that only 6 of the 226 distinct semantic errors constituted more than half of the error occurrences in each unit of exercises dealing with a single concept. Additionally, they reported a weak negative correlation between time debugging and the mark achieved in the online exams[1]. In another experiment, the students were given a debugging task[2] in the form of a program with both compiler errors and logical mistakes. On closer inspection of the recorded compiler error traces, they concluded that many students with a good understanding of programming[3] still do not acquire the skills to debug programs effectively. Students successful at locating bugs were observed using print statements or a filtering approach of commenting out certain lines. They discovered that the majority of good debuggers are good programmers while less than half of the good programmers are good debuggers. In summarizing their work they suggest that because a good portion of good programmers did not seem to be aware of some bugs, and thus were unable to debug them, they actually lacked an understanding of the actual program implementation and suggest that the ability to read and understand other people's code is an important skill and different from writing one's own.

[1]Statistical significance was not discussed.

[2]Students did not have any high-level debugging tools available.

[3]Based on their marks.

BlueJ and Java Compilation Behavior

Jadud has studied novice programmers' Java compilation behavior with a more general focus investigating the complete edit-compile cycle of alternating between editing and compiling a program instead of just the errors [8]. He collected data from 63 students on their first university-level programming course working on programming assignments in classroom tutorial sessions. The BlueJ programming environment was modified to, among other metadata, record the time, the source code and, the possible compiler error[4] at every compile. He presents a frequency distribution of compiler errors by type which shows that the minority of different types of compiler errors account for the majority of errors dealt with by students. Indeed, the 5 most common errors accounted for 58 % of all errors generated by students. Furthermore, on closer inspection he notes that the most common error types are handled in less than 30 seconds and require adding or removing only a few characters. Overall, it was common to spend very little time before recompiling after an error. He reports also that instructors observed students often recompiling their programs without attempting to fix or otherwise understand the error they had received. The data seems to corroborate this incidence of students in some manner not believing in the error reported to them since for example 21% of the time for a repeated missing semicolon error the next error was the exact same error with source code unchanged[5]. On considering the students' behavior as individuals Jadud notes that the number of compilations in a single lab session seems to suggest a grouping of students into typical ones, those who compile more often than others, and students whose behavior cannot be adequately described by this one graph. Moreover, some students had compiled up to almost 60 times in a single one-hour lab session, much more than their peers, and he had no explanation for this. Finally Jadud points out that the environment could be modified to guide students past the common errors observed but cautions making this a crutch that students would rely on doing the work for them instead of learning from it.

In another paper, Jadud reports similar observations from students of several first-year programming courses [9]. Furthermore, the paper reports a case study of a single weaker student working on one Java assignment in BlueJ. The session description illustrates the student struggling with syntax errors as the compiler errors are somewhat misleading and cause the student to introduce more and more errors to the code. He ends up spending a significant amount of time dealing with syntax errors instead of the actual program design task. During the session, the student goes on to employ a pattern Jadud calles "remove the error" where the student removes or comments out lines in order to remove and possibly locate an error by way of elimination and maybe later add the code back in. Jadud notes that this strategy was observed frequently but less commonly led to success and

[4]BlueJ only provides the student with a single error for a compilation even if many exist in the code.

[5]This might also be due to the student just refreshing the error message that was cleared accidentally as noted by Jadud in [9].

seemed in general to be an indication of being lost and confused. Another common behavioral pattern Jadud describes is that students will move on from a particularly problematic piece of code without addressing the issue. Stronger students will do something else and fix the problem later but weaker students just tend to introduce more errors. Overall, he notes that the students exhibited similar behavioral patterns in struggling with the syntax as described by Perkins et al. in solving programming problems in general [15].

Indeed, in this early work from the 80s Perkins et al. discussed some powerful characterizations of the ways how students approach solving a programming problem [15]. They observed young students, high school students learning BASIC and elementary school students learning LOGO, with an experimenter providing help only when needed in a progression from general strategic prompts to specific advice. Where students were unable to proceed quickly, they observed two general behavioral patterns that they classified as *stoppers* and *movers*. Stoppers will simply stop overwhelmed with the belief that they cannot solve the problem on their own and show unwillingness to explore it any further, perhaps immediately moving on to the next one. Movers, on the other hand, will consistently try one thing after another without ever really seeming to be stuck. At the far end of this, *extreme movers* tend to try new fixes with hardly any reflection or apparent convergence to a solution, and end up abandoning promising ideas prematurely or even going in circles trying the same thing over and over. As Perkins et al. discuss, the students are in their own way disengaging themselves from the task as instead of dealing with their mistakes and the information this might yield, they avoid them by constantly moving on. Moreover, they reported that both of these less optimal behaviors, stoppers and extreme movers, were common.

Furthermore, they discuss that the students often followed an approach they call *tinkering*. Students first write some code and then end up trying to solve the problem by making many successive small edits in the hopes of fixing the program. They describe that tinkerers are movers and with sufficient tracing of their code, or close tracking as the authors call it, to localize the problem and some systematicity to avoid compounding errors, tinkering may lead to success. Indeed, Perkins et al. associate the effectiveness of tinkering to be closely related with the extent of tracing students perform on their code. Often tinkering works poorly because without sufficient understanding of their program tinkerers may assume that small changes will help when, in fact, a change in approach is needed. Some students may also allow these changes to accumulate untested or leave them in place when they have failed, ultimately, producing an incomprehensible tangle of code. Informed by these observations, the authors note that students could be explicitly taught the pitfalls of tinkering and encouraged in such practices as removing failed fixes and considering whether a completely different approach needs to be taken if several tinkers prove unsuccessful.

Going back to Jadud's work, he has also presented an HTML-based visualization of programming sessions in terms of compilation events [9]. The visualization highlights the types of errors with colour-coding, time spent and number of characters changed between compilations, and the locations of the edits and a possible previous compilation error. One can additionally view the associated source code for each compilation via a link. Finally, Jadud describes the error quotient (EQ) that aims to quantify how much a student struggles in a programming session based on the encountered compiler errors. Consecutive erroneous compilations add to the quotient and repeating the same error even more so. Jadud goes on to report a statistically significant, yet be it a weak, correlation between a student's EQ and assignment and exam grades[6]. Overall, Jadud notes that his tools and the EQ allow a teacher to easily view how students are doing when solving the assignments as opposed to simply looking at the end result, and plan interventions when appropriate.

In later work Jadud and Henriksen have published a reimplementation of the BlueJ data collection infrastructure that extends the capabilities by also collecting data about when methods on a class or object are invoked via BlueJ object diagrams [10]. The framework consists of a BlueJ extension and a server. Tabanao et al. have used this framework to collect data on a university CS1 course from a self-selected group of 143 students over 5 lab exercise sessions [23]. They found a statistically significant and moderate (R = -0.54) negative correlation between students' mean EQ scores and midterm exam scores. Consistent with earlier results, they also report similar errors as Jadud and that top ten error types accounted for 76% of all the compiler errors. In view of this, they suggest improving the debugging ability of students by informing them of the common errors encountered by beginning programming students and discussing why they occur and how to solve them. Continuing on this work with similar data, Tabanao et al. have also tried to identify characteristics of high-performing and at-risk students [24]. Based on their midterm exam score students were grouped into high-, average-, and low-performing, at-risk, students. They found statistically significant differences between the groups in the percentage of erroneous compilations (more errors when lower performance), the occurrence of a few of specific types of common errors (more occurrences with lower performance), and the distribution of time spent between compilations (more time with higher performance). Correlation tests revealed that there was a statistically significant relationship between these erroneous compilations, the time between compilations, and EQ, and the midterm exam score. They also built linear regression models to predict the midterm exam score where EQ proved most influential but ultimately the models failed to accurately place students into the at-risk group.

Rodrigo et al. have also collected similar BlueJ data as Tabanao et al. and supplemented this with human-observed affective states [18]. Besides investigating the relationship of affective states with midterm exam scores they found that the number of pairs of consecutive erroneous compilations and those with the same edit location have a statistically significant correlation with the scores, albeit a fairly weak. They failed to build a statistically significant regression model to predict exam score using compilation data. In continuing this work, Rodrigo and Baker generated a linear regression model of a student's frustration based on the average number of consecutive compilations with the same edit location, average number of consecutive pairs with the same error, and the average time between compilations [17]. They achieved statistically significant results in predicting a student's average level of frustration across all labs but the correlation was fairly weak. Per-lab frustration could not be predicted in this way. They envision that if frustration could be identified the

[6]The dataset only included students' work in the labs with no knowledge of how much or how they worked elsewhere.

student could receive a message from the system sympathizing with them and encouraging them to keep trying.

Continuing this line of research, Fenwick et al. have too collected similar Java compilation data in BlueJ, as in work discussed above, using their own extension called ClockIt [14]. In a survey about the value of insights that this type of data may provide, they found that both the large majority of CS students and faculty perceived it would be helpful for introductory CS students to know about the types of compilation errors encountered and how a student's habits compare to his or her peers[7]. A thing of note compared with much of the other work is that students could also view the few types of graphs visualizing their own activity through a web interface. As for observations, the top five errors recorded were in Jadud's top six and make up over half of of all the compiler errors [6]. They also presented similar descriptive statistics of the time between compilations and diagrams that seem to indicate that starting early leads to better grade and that incremental work pays off as well[8]. As students had inadvertently copied the event log file or a project without this history of creating it, they were also able to identify these as potential occurrences of plagiarism.

In recent work, Utting et al. describe a plan to have a similar data collection feature built into future BlueJ versions[9] in order to collect data about students' behaviour on a large scale as opposed to the previous work dealing with closed-lab sessions at a single institution [25]. They plan to include code-edits on a line-by-line basis, compilation events, and other events such as unit test, debugger, and version control use. Furthermore, they intend to anonymize and host this data in an SQL database and allow others access for research purposes while researchers could still also collect identifiable data from their own institutes. Overall, they believe that the large scale will now allow less often used tools like the debugger and rarer error messages to be studied.

Finally, Retina is a system that not only collects Java code snapshots at compilations like in the other work discussed but also makes suggestions to students based on this data via instant messages [13]. The recommendations are based on rules such as suggesting the student work in smaller increments if their rate of errors per compilation is higher than average, or to seek help if they are spending more time on the assignment than expected. Retina's data collection is implemented as both BlueJ and Eclipse extensions as well as a modified javac compiler.

Marmoset and Code Snapshots

Spacco et al. have also collected programming session data but at a finer granularity than compilations [20]. They have used Marmoset, an automated project snapshot and submission system that commits snapshots of a student's code to an individual CVS repository every time the student saves his or her work. The automated recording of student's intermediate work is implemented with an Eclipse plugin. Using this data source, they investigated the accuracy of their bug detectors that use static analysis but, pertaining to this work, they found the CVS-based representation inadequate for exploring this type of data and present the design of a relational database schema for storing it, thus allowing SQL queries on it. The schema is built around the idea of tracking individual lines across the different versions of the code in a file as it evolves. Lines in successive versions of the code are regarded equivalent, i.e. modified instances of the same tracked line, on the basis of ignoring changes in whitespace and comments and the lines having only a small edit distance and the same relative location[10]. The results of unit tests and static analyses are also included with each snapshot in the database. Mierle et al. have also investigated students' code from CVS repositories and implemented a system for storing this data in an SQL database [12]. Furthermore, they attempted to find features that would correlate with final course grades but were unable to find any strong predictors.

In a very recent continuation of the Marmoset work, Spacco et al. report on a dataset collected in a CS2 course consisting of submissions to 6 assignments by 96 students [19]. They visualized the distribution of snapshots against the time of day and found that their students most preferred to work around 4 to 6 pm. Additionally, they visualized the number of snapshots being produced relative to the deadline and found that the majority of work was being done 48 hours before the deadline. Furthermore, they tested for linear regression between the time of the first snapshot and the final score in an assignment. They report that starting early correlates with better scores[11]. Using a heuristic of counting any time between snapshots whose difference in time was less than 20 minutes into a total time of actively working on the assignment, they also found that this time spent coding had a relationship with the final score of an assignment[12]. Something of note too is that they used a submission scheme where the amount of more detailed feedback was limited within a period of time in an attempt to get students working earlier before the deadline and discourage procrastination.

In other very recent work, Balzuweit and Spacco have discussed a prototype web service for visualizing snapshot data like the one Marmoset records [3]. They have reduced the data points into tuples of an identifier, a timestamp, a score, and a label (e.g. an error message), and show a visualization of a student's activity with regard to date (x-axis) and time of day (y-axis) where each data point is additionally colored according to its correctness as per unit tests. They hope to start work on developing standard data formats for storing and analyzing this type of data.

Web-CAT and Submission Data

Web-CAT is another automated grading system that has been made use of in examining students' progress and behavior in programming assignments. In using Web-CAT students are usually allowed to submit their work for assessment an unlimited number of times before the deadline. Edwards et al. have

[7]The survey reporting lacked details such as how large the sample sizes were.

[8]These inferences were not subjected to any statistical evaluation. A single incremental session was defined as a period of time where all successive events were less than 60 minutes apart.

[9]Version 3.1.0 that now includes the data collection features appears to have been published in June 2013 at `http://www.bluej.org/`.

[10]The method for computing the edit distance or the threshold of small are not specified.

[11]The test was statistically significant but the strength of association was very weak ($F(1, 499) = 49.94, p < 0.001, R^2 = 0.09$).

[12]The test was statistically significant but the strength of association was very weak($F(1, 492) = 6.3, p < 0.05, R^2 = 0.01$).

examined five years of programming assignment submission data from their first three programming courses [4]. They partitioned each sequence of submissions to an assignment by a student to two groups, to sequences that resulted in a score above or below 80 % of maximum. Compared with other work, in order to provide greater confidence that the cause of differences in scores has to do with differences in student behavior, rather than some innate ability that particular students possess, they then went on to remove all students from the data that consistently placed in either of the groups. They were then left with 633 students. In analyzing this data, they were able to find statistically significant results suggesting that when students received scores placing them in the high-performing group, they started earlier and finished earlier than on assignments where they received lower scores. They did not appear to spend any more time on their work. Also, around two thirds of the higher scores were received by individuals who started more than a day in advance of the deadline, while around two thirds of the lower scores were received by individuals who started on the last day or later. They suggest a possible explanation to be that when students start earlier, they simply have more opportunities to get help and then go on to perform better.

In other recent work, analyzing a similar large dataset from their locally developed Web Submissions System and data ranging from introductory to advanced assignment work, Falkner and Falkner investigated the relationship of the timeliness of submissions to students' later timeliness of assignment submissions and their average grade from courses [5]. They define a measure of average timeliness that is the average across all assignments where each is either scored 1 for the final submission being on time, i.e. before the deadline, or -1 for being late[13]. Partitioning students into two groups based on their first assignment submission – on time or late – they found that students who submit their first piece of work late seem more at risk of submitting late for the rest of their career and that this behavior also seems to correlate with the grades[14]. They suggest that with the reasonable assumption of a late submission leading to reduced marks or cascading lateness, the timeliness of the final submission of the first assignment can provide an indicator of the future likelihood of under performing and thus allow some resources to be assigned for early intervention.

Data Mining and Hidden Markov Models

Taking a very different analysis approach, Allevato and Edwards have also applied the data mining technique of frequent episode mining to discover frequently occurring patterns in Web-CAT submissions [2]. Data from two assignments by 102 students taking a C++ course after two semesters of learning Java was codified into a time series of events, such as, the student made his or her first submission or first submission that compiled without errors, the number of methods in the code changed, or the cyclomatic complexity changed. In comparing the frequent episodes of events between the students that scored well and those who scored poorly, they found that the weaker students had a frequent pattern of removing entire methods from their code which the other group did not have. They suggest this to be due to deficiencies not only in the students implementation but also in their design which can be more difficult to resolve. Additionally, they compared the frequent episodes occurring early in the course, in the first assignment, and late in the course, in the final assignment, in order to see if they might see some changes but were unable to find indication of a change in habits.

In other recent work, Piech et al. have also used data mining and machine learning techniques to analyze programming session data [16]. They recorded the compilation events in Eclipse on their CS1 course from a self-selected group of students. A Karel the Robot assignment using a Karel language based on Java was analyzed more closely. They clustered the many states of code into more high-level milestones using a metric based on differences in the AST and the API calls of the programs and using these modeled each student's development path using a Hidden Markov Model. They further clustered the individual students' HMMs to create a graphical state machine model of the few different high-level development paths students undertook to solve the assignment. Finally, they showed that the resulting different paths correlated with students' performance and with more power than the score achieved in the assignment. They also report testing the approach on a more complex Java assignment, thus proving its general applicability.

Another line of research that makes use of Hidden Markov Models in capturing the development behavior of students is the work by Kiesmüller et al. [11]. They have studied students' problem solving strategies in the finite state machine -based visual microworld programming environment Kara. Using log data from the system they have implemented an application that is able to recognize four different types of problem solving strategies automatically in real-time.

Summary

There is a steadily growing body of research on investigating learners' programming patterns and behavior from different kinds of automatically captured log data. Previous work has investigated data at many different granularities ranging from snapshots recorded every time the student saves his or her work as in Marmoset and data collected at every compilation as in the many studies surrounding BlueJ, to the submissions of automated grading systems such as Web-CAT. Data collection at the finer granularities has been implemented into modified versions of compilers or as extensions to IDEs such as BlueJ and Eclipse. The great majority of this work has focused on Java. Analyses of this type of data have tried to quantify the occurrences of compiler errors, attempted to distill activity data into usable indicators for high or poor future performance, as well as, tried to identify some higher-level behavioral patterns. To mention few of the results that seem to be gaining evidence from several sources, it seems that a minority of top types of errors accounts for the majority of the errors students learning to program encounter, consecutive erroneous compilations may be a sign of a student struggling, and starting work on assignments early will on average lead to better performance while a lot of the students' work still takes place only close to the deadline. Recent work has applied data mining and machine learning techniques to successfully discover knowledge about students behavior. Overall, the work has been motivated by a need for both discovering early actionable indicators of students struggling that would allow intervention and a desire to test and quantify anecdotal beliefs such as that starting work early and small incremental work will lead to better performance and

[13]Late submissions were allowed but these could not receive the maximum score.

[14]No statistical analyses were performed.

common types of errors students struggle with. Following the findings, there have been suggestions to explicitly teach students about observed difficulties and behavioral patterns that seem less desirable but only a few have implemented such feedback mechanisms of presenting observations on a student's behavior back to them.

With regard to this work, we are not aware of any previous work, besides that upcoming with the new version of BlueJ, that has collected data at the fine granularity of edits that we have, or included interactive console use. Indeed, Jadud has pointed out as possible future work instrumenting the programming environment further to find out what students do when their program is syntactically correct in order to "lift us out of the purely syntactic view of programming that we have" [9]. Similarly, Rodrigo et al. have mentioned as possible future work to attempt developing a finer-grained detector for frustration and other affective states, using more detailed data, such as keystrokes and mouse movements, as well as the coarse-grained compilation data already utilized [17]. In discussing the BlueJ data collection initiative and opting to collect quite detailed data, Utting et al. noted that an important decision is the selection of the data to be captured because that will determine the nature of the research questions that may later be investigated using this data [25]. Finally, our work deals with Python and we are not aware of any previous frequency analyses of Python exceptions during programming sessions.

3. DATA COLLECTION

Assignments and Learners

We collected data from a 5 ECTS Web Software Development course in Fall 2012 at Aalto University. The data analyzed in this study is from a compulsory exercise round on Python. The round had one multiple choice questionnaire on syntax and execution and three small programming assignments.

All the assignments could be submitted an unlimited number of times. If the student got full points in the multiple choice questions, then completing only two of the programming assignments was enough to pass the round and there was no reward for extra points. Almost all students who attempted the programming assignments managed to solve them. Indeed, 150, 149, and 128 students solved the assignments 1, 2, and 3, respectively. All the assignments were graded to give either zero or full points.

The backgrounds of the students varied greatly. About half of the students were master's or bachelor's level CS majors and the rest were from various other engineering disciplines. Whereas some had previous experience with both Python and web programming, some had only little programming experience in general.

Method and Tool

The programming assignments were delivered via a newly implemented web-based programming environment shown in Figure 1. The environment was integrated to the learning management system used on the course. On the left there is the code editor. It is a fully web-based programming editor that uses the CodeMirror library[15]. It provides all the functionality of a basic text editor such as cut-copy-pasting. In addition, it does syntax highlighting and has support for smart indentation that reduces the amount of required typing by, for example, automatically moving the caret to match the indentation of the previous line when entering a newline. On the right there is a console area that provides a fully functional interactive Python console including a command history feature. The console interface uses the jqconsole library[16]. The divider between the two areas can be dragged to assign more space to either of them. The buttons on the left provide functionality for running the code in the editor, submitting it for assessment and feedback, switching to fullscreen mode, and reloading the template code of the assignment. When the code is run, any output or errors of the execution are shown in the console. Also, the console shares the same Python execution environment, so after running the code in the editor, its functions and classes can be accessed – and most importantly interactively tested – in the console. The buttons on the right provide functionality for clearing the console, terminating execution, and resetting the Python environment. The programming environment had never been used on a course before and, thus, none of the students had any previous experience with it.

Python code execution in the environment is done on the client-side, in-browser and is accomplished using the jsrepl library[17] and the C implementation of Python, CPython, compiled into JavaScript via LLVM[18] using the emscripten library[19]. HTML5 web storage feature[20] is used to save and restore the contents of the editor and console including command history of a programming session. HTML5 fullscreen API[21] is used for enabling the environment to occupy the whole screen space not unlike a native application.

The feedback consisted of the execution of unit tests whose code, expected results, and student's program's results were shown. Students passed the assignment when all the tests passed. Figure 1 shows a reproduced snapshot of a student's use of the environment. On the left, the student has written a program that tries to meet the requirements in assignment 2 as described in the previous section. From the console we see that the student has executed the program and then entered a few commands in the console to test and confirm that it functions correctly. Then the student has submitted the solution and received as feedback a description of a failed test.

In this study, the web-based programming environment was used to record a detailed trace of the students' work. Students were not, however, required to use the environment to create the solution, only to submit using it, but when they did, their development activities were logged. This includes all code edits, running the code in the editor, command executions in the console, or submitting and the outcomes of all these. Additionally, using the Page Visibility API[22] and the web document's `focus` events, the trace tries to include data on when the student exited from and returned to the environment. Overall, this approach of a web-based programming environment that integrates the editor and the execution environment allows us in a relatively simple manner

[15]`http://codemirror.net/`

[16]`https://github.com/replit/jq-console`

[17]`http://replit.github.io/jsrepl/`

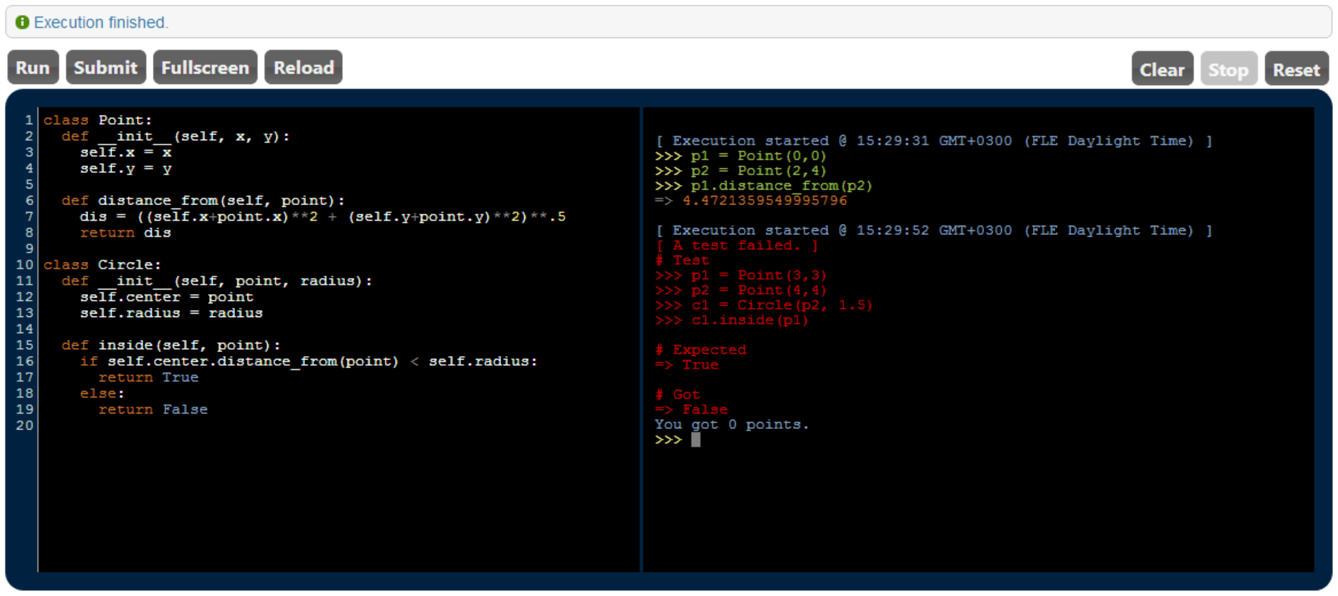
[18]`http://llvm.org/`

[19]`https://github.com/kripken/emscripten/wiki`

[20]`http://www.w3.org/TR/webstorage/`

[21]`http://www.w3.org/TR/fullscreen/`

[22]`http://www.w3.org/TR/page-visibility/`



```python
class Point:
  def __init__(self, x, y):
    self.x = x
    self.y = y

  def distance_from(self, point):
    dis = ((self.x+point.x)**2 + (self.y+point.y)**2)**.5
    return dis

class Circle:
  def __init__(self, point, radius):
    self.center = point
    self.radius = radius

  def inside(self, point):
    if self.center.distance_from(point) < self.radius:
      return True
    else:
      return False
```

```
[ Execution started @ 15:29:31 GMT+0300 (FLE Daylight Time) ]
>>> p1 = Point(0,0)
>>> p2 = Point(2,4)
>>> p1.distance_from(p2)
=> 4.47213595499958

[ Execution started @ 15:29:52 GMT+0300 (FLE Daylight Time) ]
[ A test failed. ]
# Test
>>> p1 = Point(3,3)
>>> p2 = Point(4,4)
>>> c1 = Circle(p2, 1.5)
>>> c1.inside(p1)

# Expected
=> True

# Got
=> False
You got 0 points.
>>>
```

Figure 1: The in-browser Python programming environment used in the assignments.

to collect rich data on students' programming processes. In fact, it provides a rather novel data source on students' testing sessions in the interactive console.

4. ANALYSIS AND RESULTS

The traces include a lot of data to process. To support the analysis, we implemented a web-based tool that allows us to explore the traces as they are recorded in the system. Processing of the traces is not done post-hoc but reports are immediately available for any new trace transmitted to the server. Figure 2 shows one of the views. This one focuses on what the code looks like when it is run or when the student pauses for a longer period of time after editing it – 10 seconds was the threshold value here. Other views include ones focusing on students' use of the console, students' use of the RUN button – similar to tracking compilations as in much of the previous work by others – students' edits of the code in detailed steps, and summary statistics of all students' traces in an assignment.

Testing Patterns

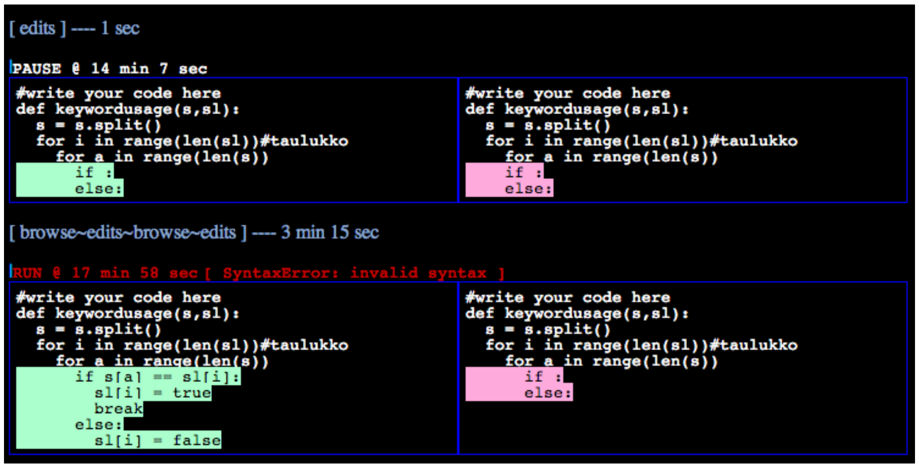
In perusing through the logs of programming sessions we observed two obvious main approaches to testing. Students would either attach test code at the end of their program and run it each time they ran their code or just run their program as is and then exercise its functionality from the console. The latter kind of data has rarely been analyzed before. Thus, we focused our investigative efforts on this and went through the console interaction in all the traces for assignments 2 and 3. The first assignment was left out of the analysis because there initially was an error in the assignment and it was modified mid-course. Also, you have to keep in mind that while we analyzed all the traces, similar to much of the previous work with students opting in to submit their data, our view is not complete because some of the students did also to varying extents work on their solution outside the web environment.

A great majority of the testing focused on the test cases given in the description of the assignments. Some students did nevertheless use test cases they had come up on their own. In an attempt to exclude such cases where the use of their own test input was not intentional but, for example, a typo, we only included in this category cases where the student had repeatedly used their own test cases (more than once). After submitting but failing to pass (which happened rarely) students would move on to using the test where their code failed during assessment. Finally, about a third of the students did not enter a single command to the console. More detailed statistics on the types of test cases used are found in Table 1. Each individual student's console testing behavior between assignments 2 and 3 was quite consistent in that they almost always exhibited the same patterns in the two assignments. In assignment 3, the test cases used to test students' solutions and give points were exactly the same as those given in the description of the problem. However, some students did much more thorough testing on their own in the console. An example of such a console session is shown in Figure 3.

Table 1: Students' use of the console for testing. The three latter categories are not mutually exclusive.

	Assign. 2	**Assign. 3**
Did not use	49 (32%)	52 (34%)
All the test from the assignment	82 (53%)	74 (48%)
All the tests from feedback	8 (5%)	- (-)
Own tests	26 (17%)	19 (12%)

In addition to examining how their program functions, some students would also use the console for more general exploration of language constructs and features before incorporating related code to their solution. In assignment 2, many students tried out the mathematical functions `sqrt` and `pow` that were needed in the assignment. An example of such behavior is shown in the console session in Figure 4. In



```
[ edits ] ---- 1 sec

PAUSE @ 14 min 7 sec
```

```
#write your code here
def keywordusage(s,sl):
  s = s.split()
  for i in range(len(sl))#taulukko
    for a in range(len(s))
      if :
      else:
```

```
#write your code here
def keywordusage(s,sl):
  s = s.split()
  for i in range(len(sl))#taulukko
   for a in range(len(s))
   if :
   else:
```

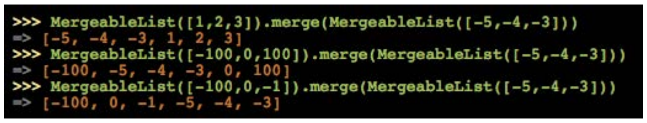
```
[ browse~edits~browse~edits ] ---- 3 min 15 sec

RUN @ 17 min 58 sec [ SyntaxError: invalid syntax ]
```

```
#write your code here
def keywordusage(s,sl):
  s = s.split()
  for i in range(len(sl))#taulukko
    for a in range(len(s))
      if s[a] == sl[i]:
        sl[i] = true
        break
      else:
        sl[i] = false
```

```
#write your code here
def keywordusage(s,sl):
  s = s.split()
  for i in range(len(sl))#taulukko
   for a in range(len(s))
     if :
     else:
```

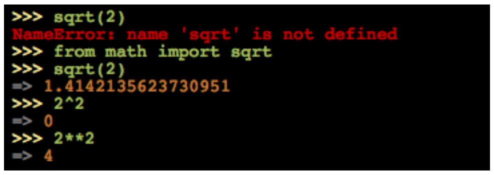
Figure 2: The view in the analysis tool showing a condensed reproduction of a student's programming session. Time progresses downwards and on the left we see the new versions of the code and on the right the previous state of it. In this line-based difference visualization, green lines signify additions (on the left) and red lines are deletions (on the right).



```
>>> MergeableList([1,2,3]).merge(MergeableList([-5,-4,-3]))
=> [-5, -4, -3, 1, 2, 3]
>>> MergeableList([-100,0,100]).merge(MergeableList([-5,-4,-3]))
=> [-100, -5, -4, -3, 0, 100]
>>> MergeableList([-100,0,-1]).merge(MergeableList([-5,-4,-3]))
=> [-100, 0, -1, -5, -4, -3]
```

Figure 3: Example of a student's console session of thorough testing.

total 15 students (10%) were found having done this kind of exploration.



```
>>> sqrt(2)
NameError: name 'sqrt' is not defined
>>> from math import sqrt
>>> sqrt(2)
=> 1.4142135623730951
>>> 2^2
=> 0
>>> 2**2
=> 4
```

Figure 4: Example of a student console session exploring math.

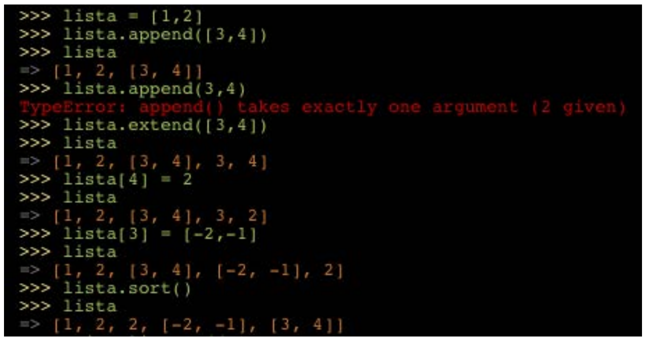
In assignment 3, it was typical for students to explore functionality of Python lists. An example is shown in Figure 5. A total of 19 students (12%) tested list functionality in the console. Furthermore, many (11) students tried the built-in function `sorted` which returns a sorted clone of a given list.

Execution Errors

The number of errors in students' executions in both running the code in the editor and running commands in the console are summarized in Table 2. As can be seen, the most common error depends on the type of the assignment. In assignment 1, there are much more indentation errors than in the later assignments. We assume this to be explained by students not being familiar with Python. Still, syntax errors along with indentation errors that may be regarded as such too were quite common overall even though Python is often praised for its relatively simple and readable syntax.

In assignment 2, the high number of NameErrors is mostly explained by students trying to call `sqrt` function from `math` package without properly importing it. Furthermore, a lot of the TypeErrors stem from students not knowing how to specify arguments when defining functions of a class. Missing `self` argument in the function definition was a typical mistake, which resulted in TypeError as the number of arguments does not match what is defined. In Python, the `self` argument is implicitly passed when calling an instance function, but needs to be explicitly specified in the method signature.

In assignment 3, the most common TypeError was caused by overwriting the built-in `sorted` function by assigning an



```
>>> lista = [1,2]
>>> lista.append([3,4])
>>> lista
=> [1, 2, [3, 4]]
>>> lista.append(3,4)
TypeError: append() takes exactly one argument (2 given)
>>> lista.extend([3,4])
>>> lista
=> [1, 2, [3, 4], 3, 4]
>>> lista[4] = 2
>>> lista
=> [1, 2, [3, 4], 3, 2]
>>> lista[3] = [-2,-1]
>>> lista
=> [1, 2, [3, 4], [-2, -1], 2]
>>> lista.sort()
>>> lista
=> [1, 2, 2, [-2, -1], [3, 4]]
```

Figure 5: Example of a student's console session of exploring the functionality of Python lists.

Table 2: Types of errors in students' code in the assignments.

Error	Assign. 1	Assign. 2	Assign. 3
SyntaxError	394 (35%)	195 (17%)	202 (23%)
NameError	274 (24%)	484 (42%)	218 (25%)
IndentationError	173 (15%)	66 (6%)	62 (7%)
TypeError	141 (12%)	252 (22%)	223 (25%)
AttributeError	66 (6%)	135 (12%)	118 (13%)
IndexError	40 (4%)	1 (0%)	25 (3%)
UnboundLocalError	26 (2%)	1 (0%)	15 (2%)
ValueError	21 (2%)	9 (1%)	0 (0%)
ImportError	0 (0%)	15 (1%)	3 (0%)
Other	1 (0%)	7 (1%)	12 (1%)

object into a variable with the same name, and then trying to use the sorted function in the code. This error was, however, probably mostly our fault, as the example test cases initially used sorted as a variable name.

Students' Perceptions

The web-based programming environment had not been used on a course before. In order to gauge students' attitudes towards this kind of web environment and get some measure of how large a portion of students' development activities the recorded traces include, we issued a short web questionnaire to the students. Students were given some points on the course for answering it. The students were asked whether they had used any other applications in editing and running code when solving the assignments, what was good and what was bad about the environment, and finally whether it should be used on the course in the future. 116 students filled the questionnaire. As 151 students submitted assignment 1, this gives a response rate of around 77 %.

A little less than half of the respondents (44 %) reported having only used the web environment for editing or execution, about a fifth had sometimes used other tools, and about a third reported having edited and run code elsewhere often or almost always. The most commonly reported choices were Eclipse IDE[23] and Notepad++[24] for editing, and command line Python interpreter and Eclipse with PyDev[25] extension for running. Still, relatively few had used any integrated development environments and apparently done without more advanced tools like visual debuggers. A great majority of the students, 79 %, mostly or strongly agreed with the statement that the web-based programming environment should be used in the course assignments in the future (mostly 35 %, strongly 44 %). Overall, students thus seem to have generally felt positively about the system.

On closer inspection of this data, we found no correlation between the use of our tool and students' performance on the course. For the students who answered the questionnaire, we compared their reported use of the in-browser editor and execution environment to their final grade (i.e. exercise + exam + project work in groups of three), sum of points from all exercises, the sum of points from Python exercises divided by the number of submissions per student (i.e. average points), and finally the number of resubmissions needed for the Python multiple choice questions. In the corresponding eight cases, the Spearman's rank correlation was low ($\rho < 0.2$) and not significant ($p > 0.05$).

103 students left feedback in the two free text questions about the pros and cons of the environment. We went through all the feedback and placed the comments into categories of common themes that emerged. 32 students gave general positive comments about the system with opinions ranging from "worked as intended" to "I can't think of anything that was bad about the environment. I think the web-based programming environment was brilliant!". About as many students mentioned the lack of setup as a specific advantage of the environment going as far as saying that "This made Python really look very simple and fun". 25 students commented on the user interface being clear and easy to use. Quite a few students also responded having liked how editing, executing, and submitting code was so conveniently integrated.

The most common complaint dealt with being used to a different programming environment which was mentioned by 28 students. Another common issue had been the editor and the interface in general being too small when integrated to the learning management system used on the course. The system could however still be switched to a fullscreen mode but students found this feature lacking and inconvenient because in the Firefox browser after switching to another window the mode would have to be reactivated again and again (but not in Google Chrome). On the other hand, it would also appear that not all students realized that the divider's position between the editor and console was adjustable. Quite a few students (16) also mentioned about the environment being "a bit sluggish" or "laggy". Indeed, the in-browser Python execution environment is noticeably slower than running a natively compiled Python interpreter directly in your operating system instead of the browser. Furthermore, some students missed having an explicit way of saving their code and mentioned even having been afraid of losing their progress if accidentally closing their browser. This would not have been the case unless browser's web storage had specifically been disabled. However, there was no clear mention of this feature being there. To mention just one of the less frequent issues, a few students did not like the color theme of having a black background.

5. DISCUSSION

Overall, students used automatic feedback rather sparingly and preferred to test their code themselves. This is rather surprising because the previous experience is that an unlimited availability of automatic feedback tends to encourage some trial-and-error behavior using the automatic system as a tester. Most of the testing was still based on code given in the assignment. Still, it is interesting that so many students used the console in their testing. It is difficult to say whether the availability of the console as an integral part of the system had an effect on students' willingness to exercise their code before submitting but this may very well be the case. On the other hand, it could be that just the availability of the test cases in the descriptions of the problems was the sole underlying factor in the students' behavior. Maybe it would be beneficial to provide students with some tests in the first few programming assignments and then omit the tests with the assumption that then they would have got used to the idea and would be more likely to continue testing their code but now with tests they have designed on their own.

The group of students who did not use the console may not have benefited much from our browser-based environment. The answers to the feedback questionnaire also revealed that students used a wide range of editors and environments to write and execute their code. This could be at least partially explained by the course being intended for 3rd year students. Especially computer science students are likely to have their favorite code editor and execution environment selected by the time they take this course. Novice students might benefit from this kind of environment more. Indeed, many of the free text answers in the feedback highlighted the ease of the environment due to not needing to install any tools.

The web-based programming environment had never before been used on a course and it still lacked some features that some students would have liked. Its interface did confuse

[23]http://www.eclipse.org/
[24]http://notepad-plus-plus.org/
[25]http://pydev.org/

some too. For example, we could infer from the traces that there were occasions where students expected the interpreter to function differently. How it did work is that whenever a student would run their code, the Python environment was first cleared and then the new code was evaluated and a message saying “Initializing Python environment” would inform about this too. However, it appears some students expected that if they had, for example, executed an assignment in the console that this definition would stick and they could make use of it across different runs. It could of course work like this but we felt that this would too easily lead to hard-to-interpret errors resulting from some old definitions that stuck in the execution environment. As another example, some students had often copied their code in the editor and then pasted it into the console in order to evaluate it into the Python environment. However, they could have just as well clicked the run-button above the editor since the console shares the same execution environment.

6. CONCLUSIONS

In this paper, we have reviewed and investigated the collection and use of programming session data in supporting educational research about programming. We have presented a web-based Python programming environment that integrates an editor, an execution environment, and an interactive Python console. The environment includes functionality for collecting and analyzing fine-grained data on how learners edit and execute code when solving programming assignments. We used the tool to examine traces of university students solving three tasks assigned as part of coursework. We reported some observations from this data, such as, how students used automatic feedback rather sparingly and made active use of the console for both testing their code and, less frequently, for exploring language features and libraries. Our results also confirm the findings of previous work that a minority of error types account for the majority of error occurrences. As a method to both replicate and extend the many previous studies dealing with more coarse-grained data, the programming environment and the access to and views of the traces can help propel future research into students' difficulties in programming assignments. Ultimately, added understanding of how the traces reflect students' progress and difficulties could enable us to provide automatic feedback and guidance based on students' observed behavior in the environment.

7. REFERENCES

[1] M. Ahmadzadeh, D. Elliman, and C. Higgins. An Analysis of Patterns of Debugging among Novice Computer Science Students. *ACM SIGCSE Bulletin*, 37(3):84–88, 2005.

[2] A. Allevato and S. H. Edwards. Discovering Patterns in Student Activity on Programming Assignments. In *2010 ASEE Southeastern Section Annual Conference and Meeting*, 2010.

[3] E. Balzuweit and J. Spacco. SnapViz: Visualizing Programming Assignment Snapshots. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 350–350, 2013.

[4] S. H. Edwards, J. Snyder, M. A. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola. Comparing Effective and Ineffective Behaviors of Student Programmers. In *Proceedings of the fifth international workshop on Computing education research workshop*, pages 3–14, 2009.

[5] N. J. Falkner and K. E. Falkner. A Fast Measure for Identifying At-Risk Students in Computer Science. In *Proceedings of the ninth annual international conference on International computing education research*, pages 55–62, 2012.

[6] J. Fenwick Jr., C. Norris, F. Barry, J. Rountree, C. Spicer, and S. Cheek. Another Look at the Behaviors of Novice Programmers. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, pages 296–300, 2009.

[7] J. Jackson, M. Cobb, and C. Carver. Identifying Top Java Errors for Novice Programmers. In *Proceedings of the 35th Annual Conference on Frontiers in Education*, 2005.

[8] M. Jadud. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*, 15(1):25–40, 2005.

[9] M. Jadud. Methods and Tools for Exploring Novice Compilation Behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*, pages 73–84, 2006.

[10] M. Jadud and P. Henriksen. Flexible, Reusable Tools for Studying Novice Programmers. In *Proceedings of the fifth international workshop on Computing education research workshop*, pages 37–42, 2009.

[11] U. Kiesmüller, S. Sossalla, T. Brinda, and K. Riedhammer. Online Identification of Learner Problem Solving Strategies Using Pattern Recognition Methods. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, pages 274–278, 2010.

[12] K. Mierle, K. Laven, S. Roweis, and G. Wilson. Mining Student CVS Repositories for Performance Indicators. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[13] C. Murphy, G. Kaiser, K. Loveland, and S. Hasan. Retina: Helping Students and Instructors based on Observed Programming Activities. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, pages 178–182, 2009.

[14] C. Norris, F. Barry, J. B. Fenwick Jr, K. Reid, and J. Rountree. ClockIt: Collecting Quantitative Data on How Beginning Software Developers Really Work. *ACM SIGCSE Bulletin*, 40(3):37–41, 2008.

[15] D. N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons. Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research*, 2(1):37–55, 1986.

[16] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling How Students Learn to Program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 153–160, 2012.

[17] M. Rodrigo and R. Baker. Coarse-Grained Detection of Student Frustration in an Introductory Programming Course. In *Proceedings of the fifth international workshop on Computing education research workshop*, pages 75–80, 2009.

[18] M. M. T. Rodrigo, R. S. Baker, M. C. Jadud, A. C. M. Amarra, T. Dy, M. B. V. Espejo-Lahoz, S. A. L. Lim, S. A. Pascua, J. O. Sugay, and E. S. Tabanao. Affective and Behavioral Predictors of Novice Programmer Achievement. *ACM SIGCSE Bulletin*, 41(3):156–160, 2009.

[19] J. Spacco, D. Fossati, J. Stamper, and K. Rivers. Towards Improving Programming Habits to Create Better Computer Science Course Outcomes. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, pages 243–248, 2013.

[20] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[21] J. C. Spohrer and E. Soloway. Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM*, 29(7):624–632, 1986.

[22] J. G. Spohrer and E. Soloway. Analyzing the high frequency bugs in novice programs. *Empirical Studies of Programmers*, 1986.

[23] E. Tabanao, M. Rodrigo, and M. Jadud. Identifying At-Risk Novice Programmers through the Analysis of Online Protocols. In *Philippine Computing Society Congress 2008*, 2008.

[24] E. Tabanao, M. Rodrigo, and M. Jadud. Predicting At-Risk Novice Java Programmers Through the Analysis of Online Protocols. In *Proceedings of the seventh international workshop on Computing education research*, pages 85–92, 2011.

[25] I. Utting, N. Brown, M. Kölling, D. McCall, and P. Stevens. Web-Scale Data Gathering with BlueJ. In *Proceedings of the ninth annual international conference on International computing education research*, pages 1–4, 2012.