Flipping Introductory Programming Classes using Spinoza and Agile Pedagogy

Fatima Abu Deeb
Computer Science Department
Brandeis University
Waltham, MA 02453, USA
Email: abudeebf@brandeis.edu

Timothy Hickey
Computer Science Department
Brandeis University
Waltham, MA 02453, USA
Email: tjhickey@brandeis.edu

***Abstract*—In this paper we present a new approach to flipping large introductory programming classes that we call the Solve-Then-Debug approach. This is a Computer Supported Agile Teaching methodology in which students solve problems using a web-based IDE we created, Spinoza, and then start reviewing failed attempts by their peers to classify the errors and comment on them. The classification and comment information is then made available as a hint to those students still trying to solve the problem when they encounter a similar error. Spinoza provides a wide variety of visualizations and dashboards that allow the instructor to closely monitor the progress of the students in this activity and to pivot to another phase of the activity at the appropriate time. It also has features that allow the instructor to easily detect and intervene with students who have failed to demonstrate mastery of the skills and concepts covered in that lesson. Spinoza builds on the ideas behind several other recent systems and this paper demonstrates that the Solve-Then-Debug approach can successfully keep all students actively engaged in learning coding skills even when there is a large range of skills in the class.**

I. Introduction

In the past few years with greater governmental emphasis on increasing the STEM workforce and the increasing demand for college graduates with STEM expertise, the enrollments in many engineering courses have expanded rapidly. In Computer Science in the US, the number of majors has nearly doubled in the past 10 years [1]. Growth in engineering and other STEM majors has also expanded recently creating a need for more effective strategies for teaching large classes in computer science and related fields. Active Learning is one pedagogical approach that has been shown to increase the effectiveness of classroom instruction, especially for large classes [2], [3] and most of these techniques involve having students engage in problem solving activities during the class. One of the challenges in teaching such a large interactive class is assessing the level of understanding of the students during the interactive activities.

In this paper we introduce the **Solve-Then-Debug** activity and argue that it is a highly effective approach to engaging students in large introductory programming classes. In the Solve-Then-Debug activity students use our web-based IDE, Spinoza [4], to solve a programming problem created before class by the instructor. Some students will solve the problem very quickly, others will take much longer, and some will be unable to solve it at all during the class period.

The main innovation of the Solve-Then-Debug approach is that students are instructed to work on debugging problems as soon as they solve the initial programming problem. Spinoza creates a large supply of these problems by looking at the most common errors, up to that point, which the students themselves are generating while trying to solve the current problem. In the debugging activity, students are shown an incorrect attempt and asked to classify the error (syntax, logic, incomplete, or "I don't know") and to write a short description of the error.

After they submit their analysis of the error they are allowed to see all of the other students' analyses. This is helpful for those students who were unable to classify the error or who incorrectly classified it. For students who are still working on the problem, Spinoza will allow them to ask for a hint and see all of the comments that other students have made for that kind of problem.

When the instructor decides it is appropriate s/he switches to the Code Review phase of the Solve-Then-Debug activity and (anonymously) discusses the code students have submitted during the exercise both for the most common errors and for the correct solutions. At the end of this activity the instructor can either choose to assign another Solve-Then-Debug activity for the same skill/concept or move on to a different skill/concept.

Spinoza records every attempt every student makes on a problem and provides a variety of views of this data to help the instructor teach more effectively. This includes views that show the current status of all students on the particular problem and on the debugging tasks. It also includes views the instructor can use after class to determine who has and who has not mastered the skills and concepts covered that day. Finally, it has views that give insight into the general problem solving strategies of the class as a whole as well as for individuals.

In the rest of this paper we will describe the way in which we flipped a large (158 student) Python programming class using Spinoza and other tools. We will then explain in detail the way in which Spinoza supports the Solve-Then-Debug activity. We conclude with an overview of the ways in which Spinoza can be used outside of class to help improve the

978-1-5090-5920-1/17/$31.00 ©2017 IEEE

instructor's effectiveness with his/her particular class.

II. Flipping CS1

In this section we describe the context of a course we taught using Spinoza in the Spring 2017 semester. The main learning objective of the course was to support all students in mastering the basic skills and concepts needed to solve problems using the Python programming language. This was a course for general students, not necessarily those who were interested in majoring in Computer Science. The course closely mirrored the demographics of the university as a whole in terms of gender, race/ethnicity, class year and major.

As in most flipped classes, the students are assigned reading to familiarize themselves with the content before class. We used the social reading website Nota Bene (nb.mit.edu) [5] to facilitate their interaction with the text. This site allows students to highlight a part of the reading and make comments about it as well as to ask or answer questions. These textual annotations are visible to all students in the class and the instructor. Our experience is that this kind of social reading stimulates a high level of engagement.

During the class students were required to log in to several web sites and engage in activities using those sites. The general audience response system we used was the TeachBack system from Discovery Teaching [6] which allowed us to ask students non-programming questions and to support a TA-monitored back-channel to answer any questions students might have without having to interrupt the class as a whole to answer one student's question.

Each class was screen recorded and live-streamed using the Echo360 system. Students were allowed to attend class virtually if they wanted, but they were required to participate using the Audience Response Systems and this participation became part of their grade (5% of the final grade). Typically about one third of the class attended virtually and participated actively using the online tools.

After each class, the teaching staff used Spinoza and Teach-Back to determine which students did not master the learning objectives for that day, and provided appropriate interventions, e.g complete the in-class problems at home, see a TA or a tutor, etc. We also sent email to those students who did not attend the class either virtually or in person.

Although many instructors are concerned about requiring the use of computers in class, we have shown in earlier work that the use of Spinoza and other online tools does not unduly distract the students in large flipped classes [7], [8] provided the class employs a highly interactive pedagogy that keeps students engaged at all times. We've also shown that the data collected by Spinoza can be used effectively for early detection of At-Risk students [9]

A. Computer-Supported Agile Teaching

The general approach we used in the class was Computer-Supported Agile Teaching[10] (CSAT). This is a teaching method which assumes the instructor has access to one or more audience response systems that provide a rich collection of learning analytics that can be used in real-time to select appropriate activities during the class as well as off-line to help detect at-risk students and analyze the effectiveness of the current pedagogy.

In CSAT the instructor teaches a concept or skill to a class by intermixing teaching activities with frequent formative assessments throughout a lesson and then, critically, uses the information gained from the real-time assessments to decide whether to continue with that concept or skill or to move on.

In a large class, it is unrealistic to assume that all students will be able to master all of the material during the activity, so some threshold must be used to decide when to move on. We have found that 60-80% is a useful threshold. The students who have not mastered the material are given additional opportunities to learn those skills and concepts after class and the instructor typically makes these interventions immediately after class using the audience response system log to determine who needs the additional help. Students can be asked to complete the problem solving activities at home and/or to meet with a member of the teaching staff to review the skills/concepts involved.

Every week or so in the CSAT model, the teaching staff reviews the learning analytics and makes an estimate of how effective the pedagogy has been. This provides an opportunity to pivot to another pedagogy that the instructor might find is more effective with this particular class. The key idea is that these decisions are based on the learning analytics collected through the formative assessments of the various audience response systems used in class.

B. Spinoza

The main tool we used for this class however was Spinoza [11], [7], [8], [9], [12], a web-based IDE which allows instructors to create and assign programming problems and to monitor in great detail the way students are (and have been) attempting to solve that problem. One of our design goals for Spinoza was to have it support activities that would keep all students in class fully engaged in problem solving with Python at all times during the class.

Spinoza allows instructors to easily create programming problems by providing a description of the program to be written, a unit test generator, and a solution which is used to test the correctness of the students attempted solutions. Spinoza also provides a wide variety of views for the instructors which facilitate agile teaching.

III. Solve-Then-Debug

In this section we describe the way in which Spinoza can be used to effectively oversee the Solve-Then-Debug activity and we provide evidence that it can be effective at keeping most of the students in a large class fully engaged in programming problem solving activities during the class.

A. TDD Programming Problems

The core activity in Spinoza is the Test Driven Development programming problem. The instructor creates such a problem by providing

- a precise description of the program the student is required to write including one function which will be subjected to a battery of unit tests
- some scaffolding code (possibly empty) for the problem
- the instructor's solution to the problem including the specified function
- the unit tests that will be applied to the student's function and compared to the results of the instructor's solution

The unit tests can include randomly generated parameters (but the pseudo-random generator must use a particular seed passed into the unit test generator). The instructor can also choose whether or not the student should be shown the results of the unit test (allowing for instant grading and revision) or whether the results should be hidden (allowing their work to be auto-graded!)

In the early stages of the course, the instructor can create a problem with a lot of scaffolding and students are expected to make only minimal changes to the code to produce the desired result. In later parts of the course the instructor can leave the initial scaffolding code empty to test if students can write a correct method signature and start from no code to write the complete code by themselves.

When students work on the problem they will write code to solve the problem by extending the scaffolding code. When they evaluate the code they will see the results of the unit tests in a table which shows the inputs, the expected results, and their actual results. Each row is color coded with green (for correct tests) and red (for incorrect tests). The instructor can choose to hide the unit tests which forces the students to create their own testing code and makes it easy for the instructor to grade their final programs.

Fig. 1 shows the student's view of one particular programming problem, not shown is the editing pane containing the scaffolding code which the student modifies before hitting the "run" button. Fig. 2 show an incorrect attempted solution of this problem in which they neglect to return "invalid" for negative scores – this particular error was made by 20% of the students. Fig. 3 shows an example of the feedback that students received for this incorrect attempted solution. The initial few values are instructor provided edge cases while the later tests are randomly generated. The editing window provides syntax highlighting and line numbers as well as showing the "invisible" characters which is important in Python where the way lines are indented plays a critical role.

One interesting approach is for the instructor to leave the description empty and to provide just the signature of the function as a stub function for the scaffolding. Students must then run the scaffolding code to see the unit tests and the expected results. From this data, they form a hypothesis about what the program is doing and then try to implement that algorithm. Each time they run their program, Spinoza provides both fixed edge cases as well as randomly generated cases, so they can't just write a single if-then-else statement encapsulating all of the cases they see. This form of problem is called an inductive programming problem and was the primary

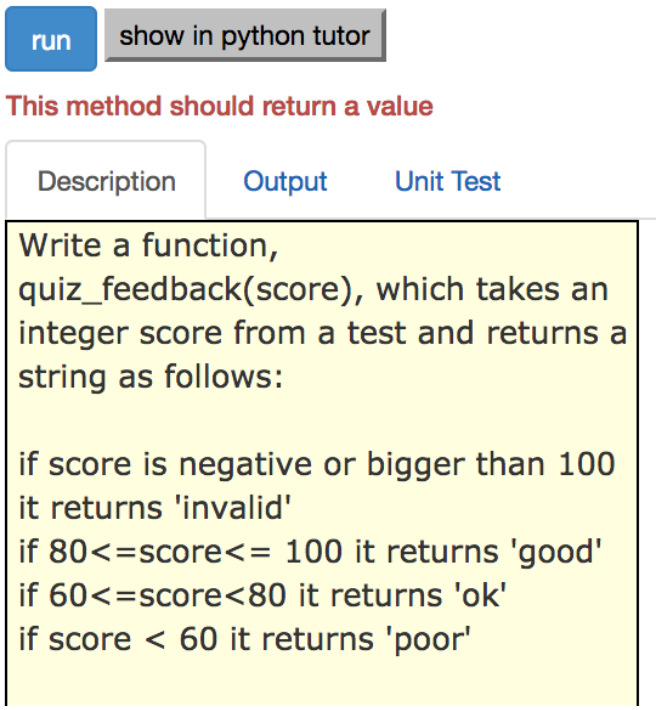


Fig. 1. student problem view

```
1 def quiz_feedback(score):
2     if 80<=score and score<=100:
3         return 'good'
4     if 60<=score and score<=80:
5         return 'ok'
6     if score<60:
7         return 'poor'
8     else:
9         return 'invalid'
```

Fig. 2. student problem view

feature of Microsoft's CodeHunt application [13]. Spinoza can easily duplicate that functionality.

B. Debugging Problems

In the Solve-Then-Debug model, students who solve a problem then immediately move on to the debugging activity. Spinoza only allows them to move on if they have correctly solved the problem. In the debugging activity, they are shown examples of incorrect attempted solutions that other students have submitted for this problem. They are asked to categorize the bug as a syntax error, logic error, or incomplete program and to write a short description of the error. They can also select the "I don't know" option, Fig. 4.

Note that when a student submits a solution that passes all of the tests they are given credit for correctly solving the problem and they are then allowed to make debugging comments on

parameters	expected	Your result	match	comment
-1	invalid	poor	False	not the same
0	poor	poor	True	
60	ok	ok	True	
61	ok	ok	True	
80	good	good	True	
81	good	good	True	
100	good	good	True	
101	invalid	invalid	True	
22	poor	poor	True	
15	poor	poor	True	
8	poor	poor	True	
81	good	good	True	
74	ok	ok	True	

12 from 13 Tests Passed

Fig. 3. student problem view

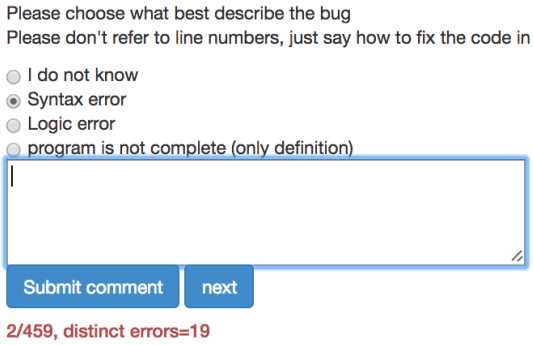
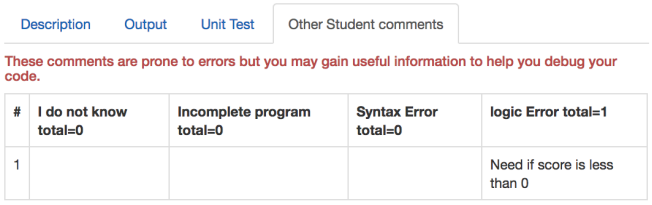


Fig. 4. student problem view

other students code. The assumption is that once they know how to solve the problem, then will be more confident in spotting other students syntax and logic errors.

When students press the ”Submit comment” button in the debugging activity, they are able to see all the responses that other students have made for this particular incorrect attempt. If they are the first to comment on that attempt, as in Fig. 5, then they see only their own response. This usually only happens for the students who solve the problem first, and they are usually the students who understand the concepts and have mastered the skills for that problem.



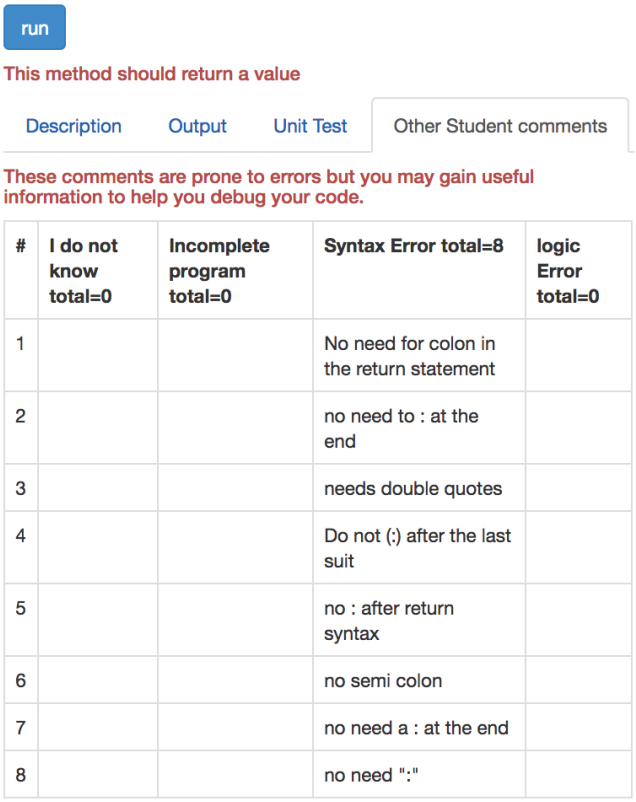
#	I do not know total=0	Incomplete program total=0	Syntax Error total=0	logic Error total=1
1				Need if score is less than 0

Fig. 5. Student Comments View for first responder

Fig. 6 shows an example of responses to a syntax error for a different problem

```
def same_suit(a,b):
    return a['suit'] == b['suit']:
```

where the student incorrectly added a colon at the end of the last line. Students who are still working on a problem and haven't yet found the correct solution can click on the ”Other Student Comments” tab and see these comments. The students are warned that these comments may not be correct, but in many cases they are quite accurate.



#	I do not know total=0	Incomplete program total=0	Syntax Error total=8	logic Error total=0
1			No need for colon in the return statement	
2			no need to : at the end	
3			needs double quotes	
4			Do not (:) after the last suit	
5			no : after return syntax	
6			no semi colon	
7			no need a : at the end	
8			no need ":"	

Fig. 6. Student Comments View for later responder

C. The Unit Test Equivalence Relationship

Spinoza assigns each attempted solution to an equivalence class by generating a set of unit tests with a fixed randomization seed and then hashing the vector of all results returned by the student's program on that set of unit tests. Thus, two programs are said to be equivalent if they produce the same answers (including error messages) on the canonical unit tests. This allows Spinoza to keep track of the most common errors and to keep a list of all student attempts associated to each such equivalence class.

The algorithm Spinoza uses for selecting the next debugging example is to take the first attempt from the largest equivalence class so far that the student has not yet debugged. Only those errors that have at least a minimum number of attempted solutions (currently 5) are shown. Once examples of all errors

have been debugged the system cycles through them again but picking the second attempt and so on.

This algorithm guarantees that each student sees the most common errors first and that the students who take the longest time to solve a problem are guaranteed to see the responses of all students who came before them for each bug and so this is a learning experience for them. The debugging practice itself is a vital skill for learning how to code and it is especially important for novice programmers to learn how to recognize the most common bugs.

D. Monitoring student progress

Once the problem has been assigned to the class the instructor needs to decide how long to let the students work on the problem. In the usual case, students will make steady progress and after a few minutes most students will have solved the problem.

Spinoza provides several instructor views that show how many students have completed the problem, how many are dealing with syntax errors and how many are dealing with logic errors. Fig. 7 shows the grouped solution view which shows every current attempt being tried by the students together with the number of students who have just submitted that attempt and the percentage of the unit tests which are correct. At the bottom of the list are those attempts that have syntax errors and hence no unit tests have been run. These are ordered by the syntax error. Recall that two attempts are considered to be the same if they produce the same values on the canonical unit tests (which are run with a specified seed for the random number generator).

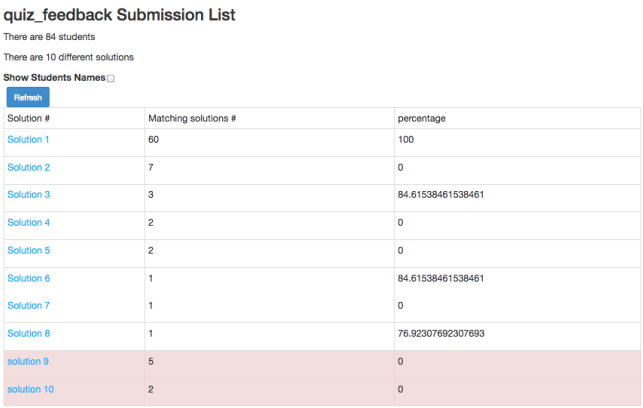


Fig. 7. Instructor grouped solutions view

The instructor can also use the Spinoza Markov Model [12] view (SMM) that shows all attempts students have made on the problem, not just the most recent attempts(Fig. 8) Each node in SMM represents an equivalent class of attempted solutions and each edge represent a transition from this solution to another in one step. Node size represents the total number of students who have submitted this attempt so far. The SMM shows all attempts the students have made, not just the current attempt; so the sizes of the nodes represent the most common attempts. Hopefully, the correct solution will eventually be the largest, but initially the syntax error attempts are larger. Additionally each edge is labeled with the number of students who transit from one node to the next. The SMM can be viewed as a probabalistic model of a theoretical random student in the class working on this problem. For each attempt, the neighbors of that node in the SMM are all the attempts that anyone in the class made after that one, and each edge is labelled with the probability that such a transition was made.

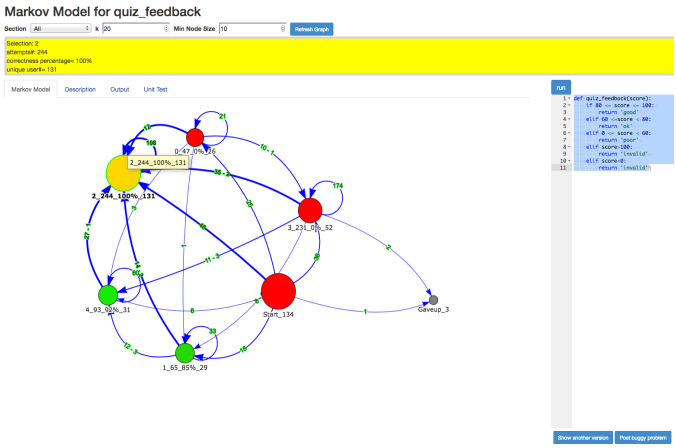


Fig. 8. Instructor Spinoza Markov model view

Another important monitoring view for the instructor is the Debug Peer View (Fig. 9) which shows all of the attempts that have been analyzed by students in the Debug phase of the Solve-Then-Debug activity. The attempts are ordered by the number of debug comments on each. This view also shows the total number of students that have analyzed at least one attempt as well as the total number of analyses that have been recorded.

The instructor can see the history of how students have progressed through the Solve and Debug stages of this activity with the Solve-Then-Debug history view shown in Fig. 10. The bottom line is the number of students who have started the Debug phase of the Solve-Then-Debug activity at any time since the activity began. The next line is the number of students (including the debuggers) who have correctly solved the problem, and the top line is the number of students who have submitted at least one attempt on the problem.

E. Reviewing the solutions and bugs

When the instructor thinks that enough people have solved the problem(typically 60-80%), he/she can begin discussing the problem with the class. The instructor can use these different views to share with the class the most common misconceptions as well as the most common successful strategies to solve the problem. Students who were not able to correctly solve the problem can use this review to follow along with the instructor and get a correct version of the code working.

For this phase of the Solve-Then-Debug activity, we find the Spinoza Markov Model to be the most useful view as it shows all of the attempted solutions with the most common mistakes

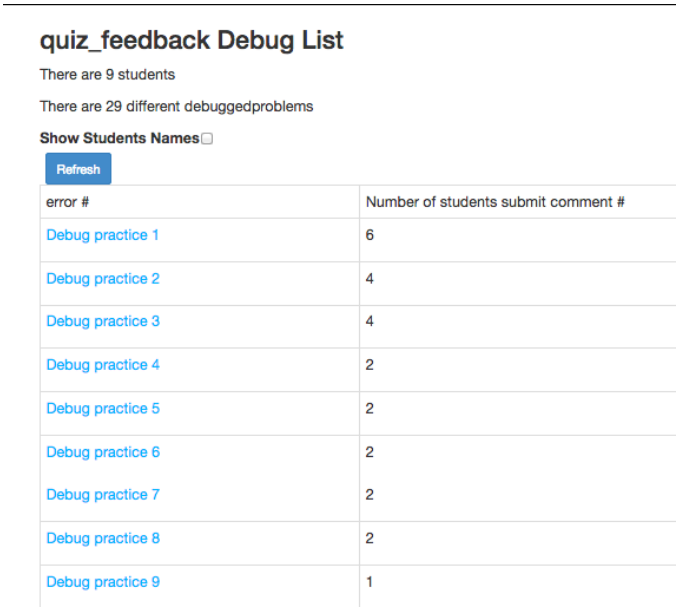


Fig. 9. Instructor Debug status view

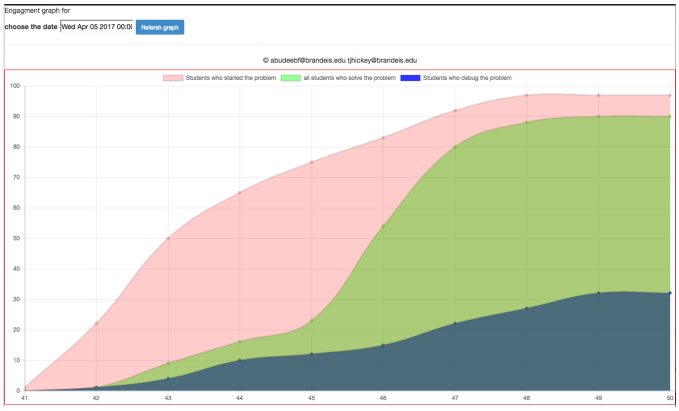


Fig. 10. Solve-Then-Debug history view

represented by the largest nodes. Moreover, by clicking on an SMM node we can see the code for the first attempt in that equivalence class. This SMM code view has forward and backward buttons that allow the instructor to view all student attempts in that equivalence class. This can be used as a group debugging activity where the students are asked to try to find the bug, or it can be used by the instructor to point out the most common errors and comment on them. Finally, it is pedagogically useful to look at all of the correct solutions and make comments about them, discussing their differences and ranking the quality of the solution. For example, the instructor can discuss choice of variable names or clarity of the control flow structure of the code.

F. Maintaining High Engagement

Fig. 11 shows a more detailed breakdown of the activity of the students on a Solve-Then-Debug activity. This figure demonstrates the potential for the Solve-Then-Debug activity to keep more students engaged than just a Solve activity by itself.

At minute 40, the instructor asked the students to solve the problem and at minute 41, some of the students started submitting attempted solutions, and at minute 42, the first few students solved the problem correctly and started debugging their peer incorrect submission. The yellow line represents the number of unique students who have debugged at least one of their peers' attempts, while the purple one represent the total number of debugging comments. This shows that many of the students who correctly submitted a solution for the problem are still engaged in the class.

Toward the end of minute 45, the instructor started discussing the solution of the problem by looking over many correct attempts. We see that the number of students submitting correct solutions increases rapidly as they are following along with the instructor and trying out the various correct solutions to make sure they understand.

In minute 48, the instructor shifts to conclude the class (which ends at minute 50), but several students continue working on the problem.

G. Moving on

If the students are not solving the problem quickly enough, the instructor can easily see this using the monitoring views and can stop the activity, discuss the concept, and assign another similar problem. It typically requires 5-10 minutes to run each instance of the Solve-Then-Debug activity in large class.

The example in Fig. 11 is an instance of this situation. We see that only about 20 students were able to get a correct solution by themselves in the first four minutes and so the instructor switched to the code review phased of the Solve-Then-Debug activity. If there were more time in the class, he could have moved to a second similar problem to validate that they learned how to work with the particular skills and concepts of that problem. Since it was the end of class, the next day picked up on those skills/concepts.

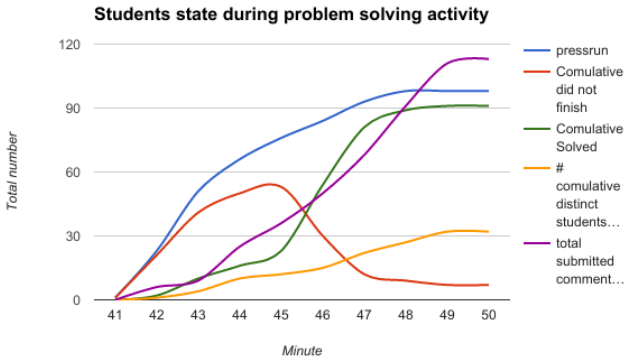


Fig. 11. Students state in the problem solving activity

IV. At-Risk Student Detection

Spinoza provides several dashboards to rank students according to their demonstrated level of mastery, either over the entire course or just for a specified day. It has been designed to make it easy for the instructor to keep in contact with students who seem to be disengaged or struggling with the material. At the end of every class, the instructor can move to the attendance view and easily send a pre-written email to all students who didn't use Spinoza during that class. Another email can be sent, with the touch of button, to all students who correctly solved less than some instructor-provided number of Spinoza problems that day. We have found that many students respond well to an email expressing the instructors concern about their performance in the class.

Every week or so the instructor can also review the overall level of mastery of the students using Spinoza from the first day of class to the current day. This view shows the number of Spinoza problems attempted, the number solved correctly, the average number of times the "run" button was pushed for each attempted problem, the average number of minutes taken for each correctly solved problem, etc. This information can be used to identify students who are systematically struggling in the course and to direct resources toward them, e.g. more office hour visits or recommendations for tutoring or additional recitations.

V. Related Work

Over the past decade there have been many new web-based IDEs which, similar to Spinoza, allow the students to work on programming problems using only their browser and which provide the students with the results of unit tests on their code.

One of the earliest of these systems was Nick Parlante's CodingBat system out of Stanford [14]. This system provided about 100 programming problems in 10 different categories. The students could specify an instructor in their settings and this would grant the instructor the ability to see the student's progress on individual problems. CodingBat was initially setup for Java problems but later included a Python section. Although CodingBat could in principle be used in class, it provided none of the class-wide monitoring tools that Spinoza specializes in, and this makes it hard for the instructor to decide when to move to the Review phase of a programming activity.

Since CodingBat first came out over a decade ago, there have been several other systems similar to CodingBat but none of these systems provide the same level of real-time monitoring that is needed to effectively run a Solve-Then-Debug activity.

The simplest versions of these web-based IDEs simply grade for correctness based on a set of instructor supplied tests [14], [15], [16], [17], [18]. More sophisticated systems also attempt to test for coding style and the student's ability to create good test cases for their code, as well as checking for any restrictions the instructor may have made on the problem (e.g. students might be required to not use the math package for a particular problem) [19], [20]

Some of these recent systems require students to upload their code to a server which grades it and gives them feedback [15], [19], while other systems are formed as web applications, like Spinoza, where students code and run their programs entirely in a browser [14], [13], [18], [17]

All of the systems besides Spinoza run the students code on a server to test correctness and this limits their scalability by increasing the server costs for larger numbers of users. Spinoza runs all code using a javascript-based Python interpreter and only sends the results to the server which stores the attempt and its meta-data in a database.

The two systems which are closest to Spinoza are the Python Response System [18] and the Informa System [21], [22].

The Python Response System [18] (PRS) is a web based clicker system that allows the instructor to assign multiple choice and code writing questions in class time, similar to Spinoza. It was developed mainly to include code writing in a Peer instruction (PI) pedagogy approach in computer science [23].

The instructor designs the code writing questions and a small number of fixed unit tests to catch certain misconceptions he suspects will be common among the students, in the same way one designs multiple choice questions where the incorrect choices correspond to commonly made mistakes. The PRS system shows the instructor, in real-time, a bar chart for each unit test, which shows the number of current student attempts that have passed (respectively, failed) that unit test.

The instructor can choose a student's solution from each each category (success/fail) for any unit test to modify and discuss. Students can also debug their code or step through their code one instruction at a time using the Online Python Tutor [24] that presents students with a visualization of the python memory model to help in the debugging process. Spinoza also provides access to the Python Tutor as a step-by-step debugger in the TDD code writing window.

Spinoza improves on the PRS experience by providing much more detailed real-time information about where the students are in the process of solving the problem. It also has the Debug Phase that PRS is lacking.

Informa is a clicker-style desktop application designed for in-class activities in a flipped classroom, just like Spinoza, but it is not a web-based application and needs to be installed in every students computer [21], [22]. Moreover, students will use the tool to write a proposed solution to the given Java coding problems in the text editor and the solution will be sent to Instructor app when they click the submit button, but the app does not run the students code so students don't get immediate feedback. Neither the students nor the instructor get any feedback about whether their attempted solutions are correct.

Informa was designed to support the evaluate-discuss-reveal conversational pedagogy approach which is superficially similar to our Solve-Then-Debug strategy. In their approach, students submit their proposed solution then are given submitted solutions from their peers to review.

Students are asked in this review/evaluation part to indicate if their peer's solution is correct or not and to rate how confident they are about their feedback (0-4). This data is displayed for the instructor in a matrix view which allows the instructor to see which students have or have not submitted a solution. The matrix also shows, for each proposed solution, which students thought it was correct and which thought it was incorrect. This is an (imperfect) proxy for having an interpreter determine correctness through compilation and unit tests.

After the students review enough of their peers' attempts, the instructor using Informa is supposed to move into a discussion phase and lead a class discussion about selected student's proposed solutions. Finally, he reveals the correct solution.

Informa was used for small classes of not more than 20 students where the matrix view makes sense, but it is not ideal for large classes since the instructor depends on students evaluations to decide when to switch to the discuss and reveal phases. In a large class, students will not able to evaluate all of their peer solutions and relying on the individual student list to show solutions would be impractical. further more some unevaluated code could have amazing learning opportunities that could be missed by this module.

Spinoza shares the Informa property of keeping all students engaged during the problem solving activity. It differs in that it provides immediate feedback as to the correctness of their solutions and it gives them practice in recognizing and classifying the most common bugs using the Spinoza Markov Model to identify the most common mistakes.

Our work provides a new approach to Contributing Student Pedagogy (CSP) [25], [26]. CSP encourages students to contribute to each other's learning and has an emphasis on valuing the contribution of others. Peer Review is the best known example of the CSP pedagogy [27], [28]. In a peer review exercise, students are asked to review other students solutions and give them feedback about the correctness and the style of the solution. The student whose work was reviewed receives the feedback and then can use that feedback to improve their solution.

Thus the main difference between our approach and Peer Review is that Peer Review engages students in a kind of reviewing dialog where the reviews directly contribute to improving that particular document being reviewed. In our approach, the purpose of the reviews is to learn how to debug programs and the original person that made that mistake won't necessarily see those comments and hence wouldn't use them to improve their code.

VI. Future Work

We plan on expanding the variety of visualizations and dashboards to give the instructor an even deeper understanding of student performance. For example, Fig. 12 shows a plot of all of the attempted solutions in a particular Solve-Then-Debug exercise. Each point represents an attempted solution, the x-coordinate is the time at which is was submitted and the y-coordinate represents the number of times that particular class of attempt had been previously submitted. This particular visualization is not yet generated in real-time as part of Spinoza, but it provides interesting information about which attempts are the most common at different times during the activity and could be useful. The blue line on the left represents the "empty" attempt from students that press eval before writing any code to see what the unit tests are. The three big lines that go up to the upper right corner are the correct solutions, the general syntax errors, and a logic error, respectively.

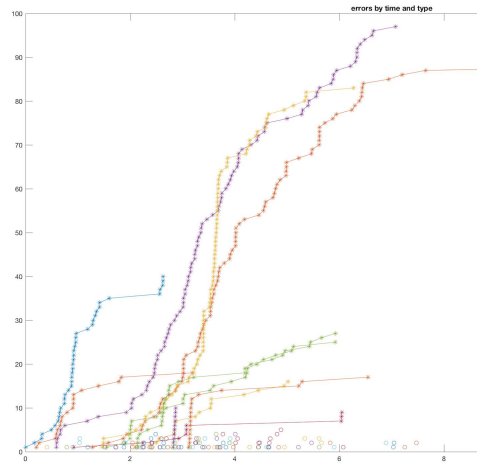


Fig. 12. Cumulative Frequency of Attempted Solutions. Each line represents an equivalence class of attempted solutions and shows the cumulative number of times that equivalence class was attempted as a function of time.

In the future we also plan to use the data collected by Spinoza to study the learning process of Novice Programmers. There have been many studies of the most common types of programming errors among novices, but Spinoza provides for a much deeper level of understanding of the problem solving strategies used by Novices.

We also plan on trying to get other faculty to use Spinoza for their Python programming classes. To do this we will create an open source version of Spinoza with installation instructions. Spinoza can be used in any class just by giving the students the URL of the server. It does not require any additional TAs or any additional software. Students do need to bring their laptops to class however, and they need to use the Chrome browser for the best experience.

VII. Conclusions

We have shown that a large introductory Python class can be successfully flipped using a Problem Solving Learning Environment for Coding, like Spinoza. The Solve-Then-Debug activity allows the instructor to keep all students engaged, from the novices who are struggling with the problem to the more expert students who solve it quickly. The Debug phase uses the incorrect attempts of the struggling students to improve the debugging skills of the more expert students and it uses the debugging analyses and comments of the expert students to provide hints to the struggling students and also provide an "answer key" for the debugging problems themselves.

REFERENCES

[1] S. Zweben and B. Bizot, "CRA Taulbee Survey Report 2014," May 2015. [Online]. Available: http://cra.org/crn/2015/05/2014_taulbee_survey/

[2] J. M. Fraser, A. L. Timan, K. Miller, J. E. Dowd, L. Tucker, and E. Mazur, "Teaching and physics education research: bridging the gap," *Reports on Progress in Physics*, vol. 77, no. 3, p. 032401, Mar. 2014. [Online]. Available: http://stacks.iop.org/0034-4885/77/i=3/a=032401?key=crossref.2338152e2386ee2984e1e1c8c44add75

[3] D. C. Haak, J. HilleRisLambers, E. Pitre, and S. Freeman, "Increased Structure and Active Learning Reduce the Achievement Gap in Introductory Biology," *Science*, vol. 332, no. 6034, pp. 1213–1216, Jun. 2011. [Online]. Available: http://science.sciencemag.org/content/332/6034/1213

[4] F. Abu Deeb and T. Hickey, "The Spinoza Code Tutor: Faculty Poster Abstract," *J. Comput. Sci. Coll.*, vol. 30, no. 6, pp. 154–155, Jun. 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2753024.2753055

[5] H. G. f. MIT, *Nota Bene (NB) About NB*, accessed: 2016-02-18. [Online]. Available: http://nb.mit.edu/about/

[6] W. T. Tarimo, F. A. Deeb, and T. J. Hickey, "A Flipped Classroom with and Without Computers," in *Computer Supported Education*. Springer International Publishing, 2015, pp. 333–347. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-29585-5_19

[7] ——, "Computers in the cs1 classroom." in *CSEDU (2)*, 2015, pp. 67–74.

[8] ——, "A flipped classroom with and without computers," in *International Conference on Computer Supported Education*. Springer, 2015, pp. 333–347.

[9] ——, "Early detection of at-risk students in cs1 using teach-back/spinoza," *Journal of Computing Sciences in Colleges*, vol. 31, no. 6, pp. 105–111, 2016.

[10] W. T. Tarimo, "Computer-supported agile teaching." [Online]. Available: http://gradworks.umi.com/10/15/10157897.html

[11] F. A. Deeb and T. Hickey, "The spinoza code tutor: Faculty poster abstract," *J. Comput. Sci. Coll.*, vol. 30, no. 6, pp. 154–155, Jun. 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2753024.2753055

[12] F. A. Deeb, K. Kime, R. Torrey, and T. Hickey, "Measuring and visualizing learning with markov models," in *Frontiers in Education Conference (FIE), 2016 IEEE*. IEEE, 2016, pp. 1–9.

[13] J. D. H. Nikolai Tillmann and J. B. Tao Xie, "Code hunt: Gamifying teaching and learning of computer science at scale," *Proceedings of the first ACM conference on Learning at scale conference*, pp. 221–222, 2014.

[14] N. Parlante. Codingbat code practice. [Online]. Available: http://codingbat.com/

[15] M. Sherman, S. Bassil, D. Lipman, N. Tuck, and F. Martin, "Impact of auto-grading on an introductory computing course," *J. Comput. Sci. Coll.*, vol. 28, no. 6, pp. 69–75, Jun. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2460156.2460171

[16] O. Gotel, C. Scharff, and A. Wildenberg, "Extending and contributing to an open source web-based system for the assessment of programming problems," *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, 2007.

[17] D. H. Andrei Papancea, Jaime Spacco, "An open platform for managing short programming exercises," *Proceedings of the ninth annual international ACM conference on International computing education research*, pp. 47–52, 2013.

[18] D. Zingaro, Y. Cherenkova, O. Karpova, and A. Petersen, "Facilitating code-writing in pi classes," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 585–590. [Online]. Available: http://doi.acm.org/10.1145/2445196.2445369

[19] S. H. Edwards and M. A. Perez-Quinones, "Web-cat: Automatically grading programming assignments," *SIGCSE Bull.*, vol. 40, no. 3, pp. 328–328, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1597849.1384371

[20] S. H. Edwards, "Teaching software testing: Automatic grading meets test-first coding," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 318–319. [Online]. Available: http://doi.acm.org/10.1145/949344.949431

[21] M. Hauswirth and A. Adamoli, "Solve & evaluate with informa," *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, 2009.

[22] ——, "Teaching java programming with the informa clicker system," *Sci. Comput. Program.*, 2013.

[23] B. Simon, M. Kohanfars, J. Lee, K. Tamayo, and Q. Cutts, "Experience report: Peer instruction in introductory computing," in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '10. New York, NY, USA: ACM, 2010, pp. 341–345. [Online]. Available: http://doi.acm.org/10.1145/1734263.1734381

[24] P. J. Guo, "Online python tutor: Embeddable web-based program visualization for cs education," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 579–584. [Online]. Available: http://doi.acm.org/10.1145/2445196.2445368

[25] G. L. Herman, "Designing contributing student pedagogies to promote students' intrinsic motivation to learn," *Computer Science Education*, vol. 22, no. 4, pp. 369–388, 2012.

[26] K. Falkner and N. J. Falkner, "Supporting and structuring contributing student pedagogy in computer science curricula," *Computer Science Education*, vol. 22, no. 4, pp. 413–443, 2012.

[27] H. Søndergaard and R. A. Mulder, "Collaborative learning through formative peer review: pedagogy, programs and potential," *Computer Science Education*, vol. 22, no. 4, pp. 343–367, 2012.

[28] D. Clarke, T. Clear, K. Fisler, M. Hauswirth, S. Krishnamurthi, J. G. Politz, V. Tirronen, and T. Wrigstad, "In-flow peer review," in *Proceedings of the Working Group Reports of the 2014 on Innovation & Technology in Computer Science Education Conference*. ACM, 2014, pp. 59–79.