AC 2011-831: JAVAGRINDER: A WEB-BASED PLATFORM FOR TEACHING EARLY COMPUTING SKILLS

James Dean Palmer, Northern Arizona University

Dr. Palmer is an assistant professor at Northern Arizona University where his research interests include undergraduate computer science education, language design, and computational storytelling.

Joseph Flieger
Eddie Hillenbrand

©American Society for Engineering Education, 2011

JavaGrinder: A Web-Based Platform for Teaching Early Computing Skills

Abstract

Even as Bureau of Labor Statistics predictions indicate unprecedented demand for software engineers in the next five years, nationwide retention rates of incoming majors are alarmingly low and interest in computer science remains stagnant. Many educators are reevaluating how we teach computer science in the critical first year of study and are questioning the emphasis of programming and tool mastery over more abstract computational thinking.

While specialized development tools and integrated development environments intend to simplify programming tasks they typically do little to support pedagogical development and evaluation of a broad range of problems at varying levels of computational abstraction. Worse yet, the languages and tools used in introductory courses often create barriers in the form of boiler plate code, complex build tools, and unintuitive interfaces that discourage students from engaging in directed and focused practice.

In this paper we review existing introductory computer science tools, enumerate barriers to student learning we have identified in our own classes, and introduce a new web-based pedagogical platform for teaching computer science that emphasizes problem solving and core computer science concepts while deemphasizing the role of specialized development tools. This is accomplished with JavaGrinder, a task specific web 2.0 environment where students can work either individually or as teams on bite-sized problems that focus on solid software engineering practices and concept mastery. Concepts are presented within real-world contexts that advocate computer science as an exciting multidisciplinary field, rather than as an abstract world of syntax and arcane codes. JavaGrinder is designed to facilitate problem-solving skills by exposing the salient aspects of a problem, providing guided practice, and immediate feedback. JavaGrinder teaches true Java programming, while shielding students from language and platform-specific minutiae. In this way, JavaGrinder addresses the critical gap between successful introductory programming environments and realistic functional programming and software engineering.

1. Introduction

According to the 2009-2010 Bureau of Labor Statistics' Occupational Outlook Handbook, computer software engineering is projected to be among the fastest-growing and highest-paid occupations from 2004 to 2014[1]. Even in a floundering economy, the Bureau of Labor Statistics found the computer systems design industry was among the top five performing industries between January 2008 and January 2009[2]. In 2006, Money Magazine ranked software engineering as the number-one occupation, based on salary and various quality-of-life factors[3]. Again in 2010 Money Magazine ranked "software architect" as the number-one occupation[4]. Yet, a crisis looms. Nationwide, CS enrollment

is nearly half of what it was in 2000[5] and Drop-Fail-or-Withdraw (DFW) rates for a first course in computing commonly hover between 35% and 50%[6]. Stories of even higher DFW rates in a Computer Science 1 (CS1) course are not uncommon. At the same time, there are unprecedented decreases in diversity as young women have fled the field; DFW rates for women have soared at an even greater pace than that of men[7].

Although the high DFW rate in CS1 is undeniably a complex phenomenon, two central factors are that students simply need more opportunities to practice basic skills and the experience needs to develop confidence and motivate long-term learning and career goals. Some students may not engage in sufficient practice due to low motivation, boring exercises, frustration with syntactic minutiae, and time management. If students fail to see a direct connection between learning and real problems and career paths, a perfect storm for poor retention is formed. Some research suggest a particular disconnect with young women thinking about the field when early exposure does not demonstrate how skills and problems relate to real world problems and careers or build confidence and self-efficacy[8,9].

JavaGrinder seeks to improve student learning for freshmen computer science majors by developing a highly interactive web-based training and practice system that may augment traditional curricula with exercises that emphasize the engaging interdisciplinary nature of computer science, modern software engineering practices, problem solving techniques, and instant feedback. We believe the key to JavaGrinder's success is making connections between students and realistic computing problems. Specifically, we promote the development of microworlds for designing engaging and relevant tasks to practice early computing skills. A microworld is a rich "virtual reality" environment with its own rules, actions and consequences. Microworlds give students the opportunity to master computer science skills on assignments associated with properties, phenomena and problems associated with microworld environments. Microworlds can use advanced simulation to embody the kinds of social and technical problems that must be addressed in realistic engineering problems. These simulated environments afford students access to unique simulated resources and situations that might otherwise be altogether unavailable.

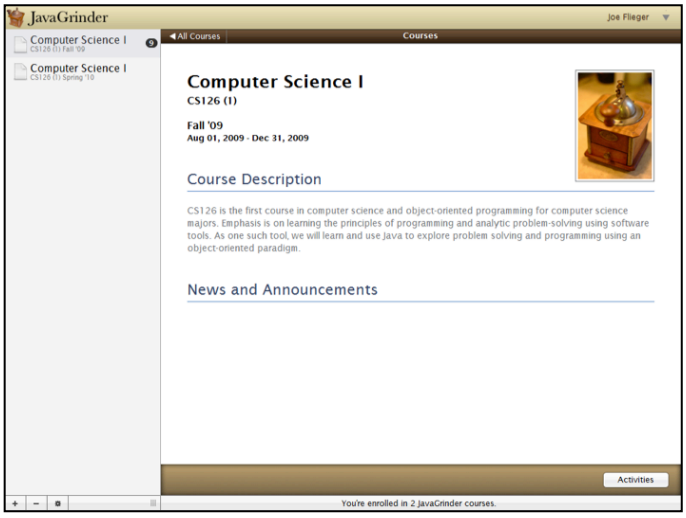

Figure 1. The JavaGrinder course information page links users to course activities.

2. Related Work

Against this backdrop a number of projects have emerged that have sought to improve the introductory programming experience.

2.1. The Kinder, Gentler IDE

Integrated Development Environments (IDEs) have become one of the mainstays of modern programming; providing powerful tools and editing capabilities for writing software. Simple text editors have given way to incredibly complex authoring software with syntax highlighting, code analysis, build tools, revision control tools and even diagramming and modeling features. Our reliance on IDEs has become such that some toolmakers often make it almost impossible to build or compile a program without using a sophisticated IDE. Although IDEs intend to make programming tasks easier they often come with a high learning curve.

Recognizing that there must be some middle ground between a full-fledged IDE and a simple text editor, several projects have sought to make an IDE specifically for new programmers.

Although a number of projects have taken up this task, DrScheme[8] and DrJava[9] have proven to be some of the more successful solutions. Several textbooks including Niño and Hosch's Introduction to Object Oriented Design use DrJava extensively.

Using a much different user interface approach, BlueJ (http://bluej.org/) has shown rapid adoption by a number of university CS programs and textbooks[10-12]. Some of BlueJ's innovations have been using a UML-like interface to represent files and including unit testing and debugging functionality while still maintaining a trim easy to learn user interface[13].

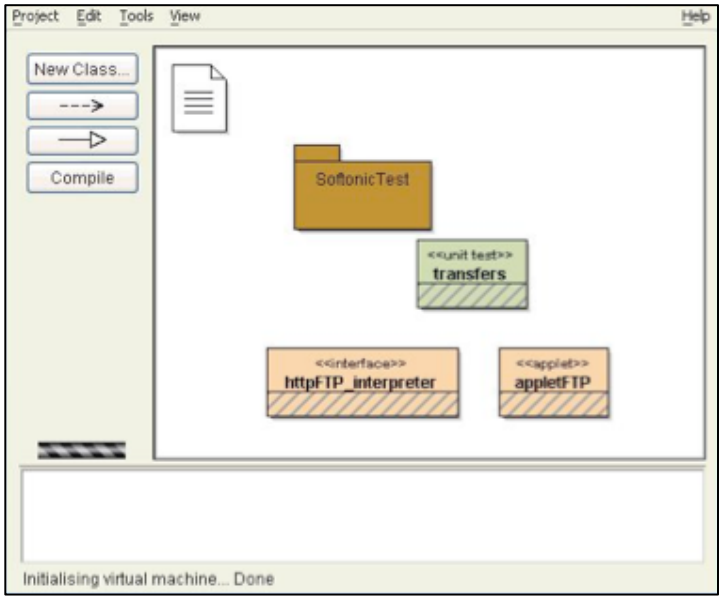

Figure 2. BlueJ uses UML idioms for representing files and basic class relationships.

BlueJ's success has left some wondering what is the best way to transition students from BlueJ to a heavier weight IDE. Netbean's BlueJ edition was one answer to this question. This product is a full-featured Netbeans installation with special modules to give it a BlueJ-like feel. A bit more than a pedagogical stepping-stone, the NetBeans BlueJ edition has been used in a number of successful commercial and open source projects.

Although these kinder, gentler IDEs have undoubtedly improved the beginning experience, a fair bit of any intro course will be devoted to learning these tools. Experience using BlueJ in our own classes has shown that it can still be difficult to setup and a daunting tool for students to use. While none of these projects address the issue of content quality, the Greenfoot project (http:/greenfoot.org/) certainly does[14,15]. Greenfoot supports the development of scenarios that students can use to learn programming skills. Most of these scenarios take the form of simple games with actor in the environment that must be manipulated. Often times these scenarios are open ended and there is no strong evaluation mechanism.

2.2. Virtual Learning Environments

Virtual learning environments represent a much different approach for learning early programming skills. Virtual learning environments often dismiss or deemphasize the role of the IDE and develop a virtual world where problems exist. Randy Pausch's Alice environment (http://alice.org/) is one of the best examples of this genre providing a drag-and-drop interface to learn programming concepts. A considerable amount of research has examined the efficacy of Alice for teaching early computing skills[16,17].

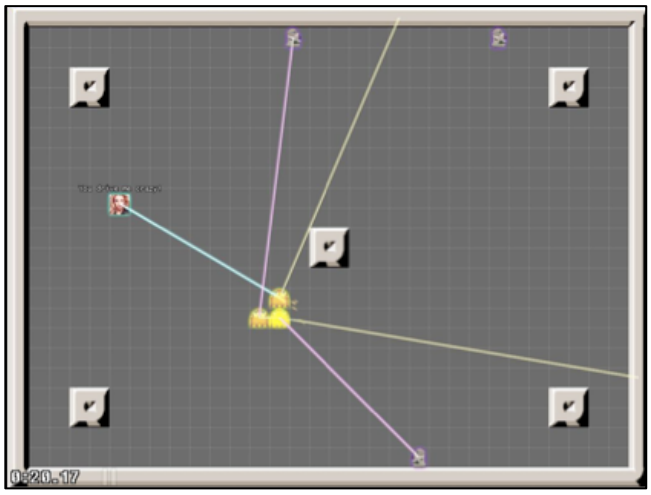

Figure 3. Violent robot island provides a virtual world where programmable robots fight it out in order to demonstrate path finding and strategic planning.

While Alice is an impressive tool it tends to be targeted at kids and true computer novices. Alice also tends to emphasize gaming terminology, constructs and memes. This entertainment and younger demographic focus will likely be even more apparent in the upcoming Alice 3.0 release that will sport Disney themed characters and situations. While Alice problems are indeed fun to solve, they don't feel realistic or emphasize engineering approaches for problem solving. Even more problematic, is that it lacks any bridge to work in a general-purpose programming language like Java making it a difficult system to work into a CS1 course that teaches Java.

Virtual environments that are based on Java include Robocode[18,19] and our own Violent Robot Island (See Figure 3). Projects like these tend to be very special purpose with specific learning goals in mind. A commonality of both Robocode and Violent Robot Island, for example, is that programmers must carefully weigh winning gaming strategies against resource utilization. Another common thread with tools in this genre is that programming tends to be gaming or entertainment oriented. There are a number of ties with serious and educational gaming research. For an in depth survey of virtual learning environments and beginning languages developed since the 1960s see also[20].

2.3. Web-based Learning Environments

Many early web based learning systems for computer science were essentially on-line multiple choice. Fortunately, a number of environments have been developed which provide more advanced experiences with instant feedback, hints, supplementary instruction, and automated grading. Web based systems typically are more portable and require no installation and students can quickly access problems and make progress without many of the traditional hurdles associated with IDEs or large complex learning frameworks.

The JavaBat project (http://javabat.com/) is one such example of a web based learning environment. JavaBat provides a series of small "bite-sized" programs for students to work through in order to learn basic concepts associated with arrays, iteration, strings, Boolean logic,

and recursion. Unfortunately, JavaBat provides only about forty problems for students and is difficult to modify or extend, often making it impossible to design exercises that are relevant to a particular course. The user interface sports a simple HTML design.

A commercial alternative, Codelab, has similar features but supports languages other than Java and allows instructors to add their own problems. Codelab checks correctness by checking the state of the machine against the desired machine state.

3. JavaGrinder and the New Computer Science

Today's computing is not your father's computing. Interaction design, empirical studies of user experience, project management, understanding social impacts of technology, and much more are the new faces of academic computing.

Bonnie A. Nardi
University of California, Irvine

It's seems clear that at least since the dot-com bust, Computer Science has had difficulty attracting new students. The percentage of college freshmen planning to major in computer science dropped from 3.4% in 1998 to 1.4% in 2004. Further, data from the National Center for Education Statistics shows that computer and information sciences conferred fewer degrees than either the visual and performing arts or the social sciences and history. Downward trends for women entering the field date back to the 90s. After a brief resurgence in interest in computer science between 1999 and 2000, the percentage of female computing students has dropped from 19% to 12% in 2006[21]. The picture seems particularly grim when you consider that NSF data shows the number of women in science and engineering held steady at 50-51% during the same period. At odds with the down turn in enrollment is the fact that demand for Computer Science majors continues to grow. A 2006 ACM report on job growth notes, "Despite all the publicity in the United States about jobs being lost to India and China, the size of the IT employment market in the United States today is higher than it was at the height of the dot.com boom."[22] Computer Science and Software Engineering continue to make career top 10 lists where the careers are connected with good salaries and flexible career opportunities[3,4,23].

Numerous studies and analysts have considered why the field has failed to attract new students. Often cited causes include high job loss rates, negative portrayals of the field as "nerdy", impressions that computing professionals need extraordinary proficiency in math and a poor understanding of the field by high school guidance counselors[24]. As early as 1985, computer science practitioners began to worry that the field was becoming too closely associated with programming and not the full range of problem solving and engineering skills that represents the reality of the discipline[6]. The Association for Computing Machinery (ACM) has responded to these charges by mounting a campaign to a) take computer science to the high schools, b) increase the visibility of computing as a career, and c) develop curriculum and studies on how to convey an appealing message that describes the opportunities and challenges of the field[24].

Compounding this problem is the fact that once we have prospective majors in the classroom, their prospects for success are not great. Low enrollments and high DFW rates are obviously connected. Although computer science advocates claim computer science is not about mindless abstract programming done by lone hackers late at night, that is almost exactly what most

introductory computer science classes asks students to do! Introductory CS problems are notoriously disconnected with textbooks mindlessly echoing similar banking problems to demonstrate "transactional" concepts or vending machines that represent state. Historically, introductory CS courses have not emphasized teamwork and instead emphasize independent programming skill and tool mastery leaving students to work long and often frustrating hours fighting syntax and compiler minutiae. We might as well tell our students where to get the Jolt cola and Cheetos to perfect the negative stereotype we are supposedly fighting.

The situation for computer science is dramatic but it's part of a larger picture of declining leadership in engineering education. In Moving Forward to Improve Engineering Education[25], the National Science Board (NSB) has called attention to "how engineering education must change in light of changing workforce demographics and needs." The NSB has posed three key challenges: 1) we must respond to the changing global context of engineering, 2) we must reconsider the perception of engineering and 3) we must work to retain engineering students. In the context of computer science, the NSB's call to action seems especially poignant. The NSB charges that basic engineering skills and knowledge have become a commodity and nowhere has this been evidenced more clearly than IT outsourcing. Gartner analysts predict that by 2011 India will become a leading "megavendor" of IT services competing for deals with values worth more than $1 billion. At first blush it might seem American workers simply can't compete with cheaper foreign labor but other sources suggest the success in outsourcing is due in part to a lack of qualified American workers. In 2007, Bill Gates pleaded to both congress and the public for more green cards and H-1B visas[26] to feed the growing need for computer science professionals. Improving engineering recruitment and perception was the second key challenge posed by the NSB. Engineering is simply not attracting enough people to the field and computer science seems to be a bellwether as an increasingly negative and misunderstood profession. The third challenge laid by the NSB was to improve retention of engineering students. Attrition is formidable in most engineering programs, but attrition in computer science is often among the worst (especially during the first year). These challenges set forth by the NSB complement and echo the ACM's own response to the computer science education crisis.

3.1. Problems with the "old computer science"

In our own work on retention and recruitment we identified several very specific aspects of the CS1 experience that often frustrates students and teachers alike in CS1 courses and directly relate to the NSB and ACM's observations on undergraduate education:

1. *Unrepresentative of real-world problems.* Coming up with new and engaging material can be difficult, but it often seems that computer science is plagued with abstract mathematical examples and bank transactions, which have little to do with the diverse problems real-world computer scientists encounter. Consequently, material doesn't connect students with potential career options that could provide early inspiration and motivation.
2. *Poor exposure quality.* Students often spend too much time writing monotonous "boiler-plate" code and getting stuck on syntax minutiae instead of exploring core issues.
3. *Lack of interactivity.* Students only know how well they performed on an exercise after the assignment has been graded and long after the "teachable moment" has passed. Often, the only interactive feedback students receive is cryptic compiler errors.

4. *Complex development environments.* Assignments often require using and configuring a compiler and development environment. Students often fight incompatible versions, dependencies and IDE-specific workflows to make any progress.
5. *Lack of teamwork.* Most textbook assignments and CS programs we are familiar with focus on individual mastery of skills, neglecting the positive and necessary role of teamwork.
6. *Difficult assessment.* As a teacher working with large sections, it's often difficult to provide comprehensive feedback, execute every program, and look at every line of code.

3.2. Our Approach

In an effort to address these goals we developed JavaGrinder as a web based learning system. JavaGrinder is designed to facilitate problem-solving skills by providing appropriate amounts of guided practice and immediate feedback through small problems designed to be completed quickly while emphasizing specific concepts.

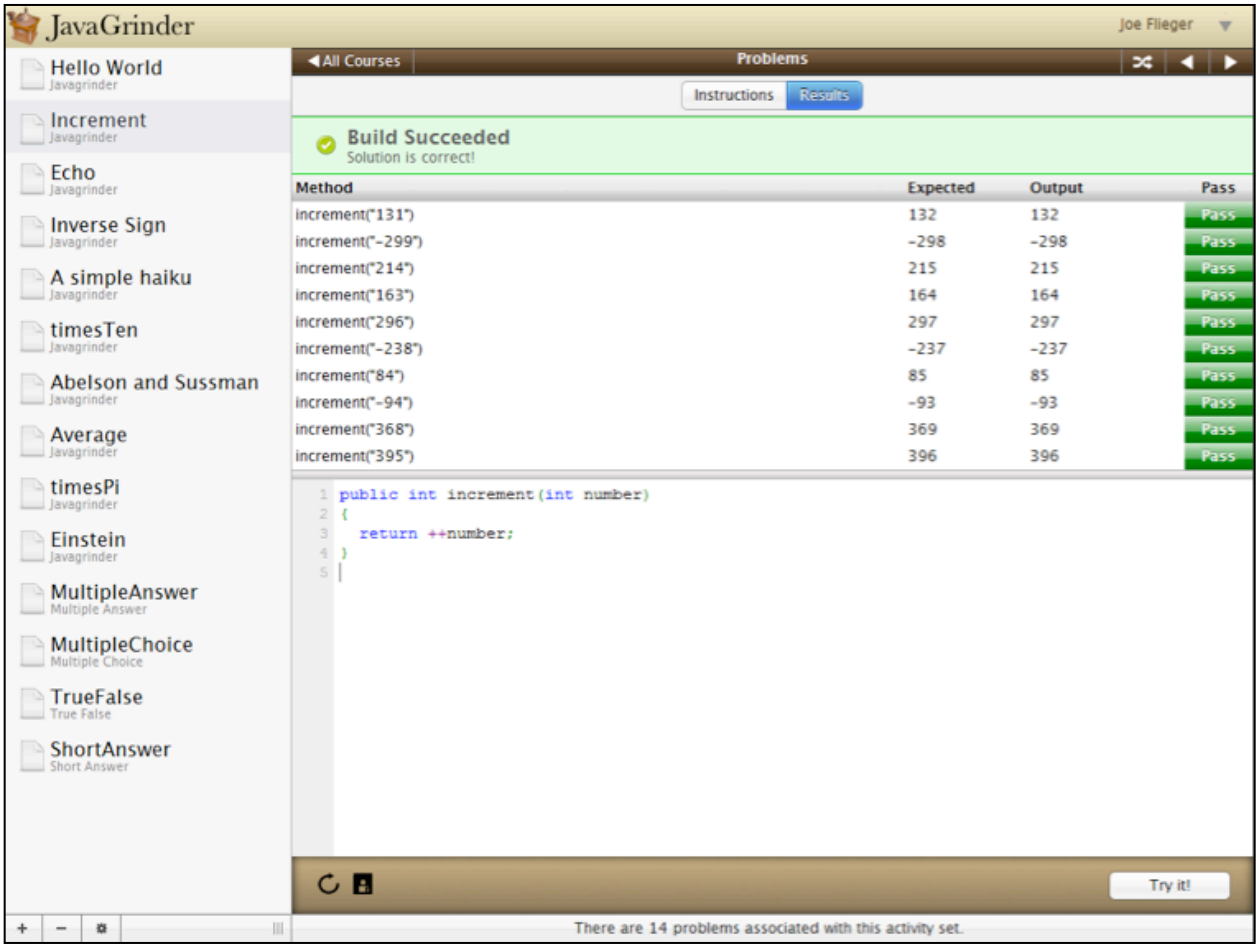

Figure 4. JavaGrinder is a web-based application that focuses on short targeted problems. In this warm-up example the student must implement a function that increments a number but doesn't have to implement any other classes or framework code.

JavaGrinder's design is a reaction to the problems indentified in the last section. We tackle the problem of unrepresentative material by hiding boiler plate framework and exposing just the interesting and salient aspects of a problem. While studying a complex system like a fire model simulator would be beyond the scope of an introductory class, JavaGrinder lets us pull small parts of the system out so they can be manipulated and studied. Thus, we connect abstract programming techniques and concepts with realistic roles in software systems.

Since only a small part of the problem is exposed the exposure quality is improved. Boiler plate and framework code are hidden so that students don't spend time learning frameworks and instead spend time learning core language and problems solving concepts.

A traditional lack of interactivity is addressed by allowing students to check their solutions without penalty against a suite of unit tests that provide instant feedback and instant scoring. Since it is a web-based application, there is no special software to install or complex development environment to learn. JavaGrinder problems may ask students to define whole classes, functions, or just code fragments. Thus, a broad array of problems that may have been difficult for instructors to pose becomes easy, making it possible to integrate problems that address the most salient aspects of a problem. Special attention has been paid to make problems engaging and realistic for a diverse audience.

4. Designing Problems in JavaGrinder

Problems are described using a wiki-like language. This markup language supports the inclusion of text formatting, images, multimedia, tables, and even UML diagrams. With this rich markup, we can give graphical representations of algorithms like finding an angle in a triangle. UML diagrams are used to describe classes which students are asked to then implement. JavaGrinder supports a host of different problem types including multiple-choice, multiple-answer, true-false, short answer, and coding exercises. Using a broad array of problems and activities, we can support a full complement of CS1 coursework through JavaGrinder. Built-in survey functionality supports student assessment to guide course improvement and motivate future work.

Most of the standard learning activities like true-false and multiple-choice behave much like any other learning system. Coding exercises are slightly more complex. Each exercise has a description, a skeletal "starter" solution, a reference solution and a unit test.

The starter solution is the code the student sees when he or she begins the activity. The reference solution is the code that the student solution is compared against. The unit test then provides the glue that compares the results of the reference solution with the results of the student solution.

Lets consider one of the JavaGrinder warm up exercises. The problem description reads, "Take the integer input, x, and return its value multiplied by 10." The instructor that created this problem would begin by inputting the problem description and then providing this starter code:

```
int timesTen(int x) {

}
```

The starter code generally provides just enough of the framework or method signature for students to get started and have working code to run against the unit test. In this example, the

student only needs to add a return statement for the code to compile correctly. Next, we need to write a reference solution. The reference solution will be executed after the student solution and should duplicate the student's results if their solution is correct. The reference solution for this problem would be:

```
int timesTen(int x) {
  return x * 10;
}
```

We embed this solution as part of the test. Here's the test we wrote for this problem:

```
class MyTest extends UnitTest {
  MyTest() {
    Random gen = new Random($SEED);
    for (int i = 1; i <= 10; i++) {
      int r = gen.nextInt(1000);
      r -= 500;
      unit_test("timesTen(" + r + ")",
                timesTen(r),
                timesTen$SEED(r));
    }
  }

  int timesTen$SEED(int x) {
    return x * 10;
  }

  $USERCODE
}
```

There are two special variables in this code. $SEED will be substituted with a random number created by the calling code. This $SEED can be used to seed the random number generator but is also used as part of the name for the reference solution. This prevents clever students from determining the name of the reference solutions and simply calling that code from their own. $USERCODE then contains the student code that should be evaluated and is a straightforward substitution.

The UnitTest class then provides several test methods that communicate the results of the tests from the Java environment to the learning management system. In this particular example, 10 tests will be performed that will randomly select values between -500 and +500 and compare the result of multiplyByTen with multiplyByTen$SEED. The student never sees the reference solution or the test framework. Instead, when the student clicks "Try It" the test framework is executed with the student result and the student sees the results of the unit_test calls as output:

Unit Test	Expected	Observed	Result
`timesTen(19)`	190	190	Passed
`timesTen(-290)`	-2900	2900	Failed
`timesTen(420)`	4200	4200	Passed
`timesTen(211)`	2110	2110	Passed
`timesTen(-56)`	-560	560	Failed
`timesTen(89)`	890	890	Passed
`timesTen(-308)`	-308	308	Failed
`timesTen(-72)`	-720	720	Failed
`timesTen(344)`	3440	3440	Passed
`timesTen(238)`	2380	2380	Passed

In this example the student has made an unlikely mistake and taken the absolute value of x before returning it. By inspecting the expected and observed values students can fix their own code and also get used to the idea of unit and regression testing. While testing does not prove correctness, it provides confidence that the student implemented a correct solution without having to carefully inspect the student code. Thus, the effectiveness of our evaluation system is tied to the effectiveness of the test.

5. Student Response

We have been using JavaGrinder in our CS1 class for the last two years. Each semester we have solicited feedback from students in order to improve the system. While much of the early feedback concerned improving the code editor or other user interface elements, some feedback was quite substantive. The first revision of JavaGrinder only allowed students to try problems during the window before the due date but we received overwhelming feedback from students that they wanted all attempted coding activities available to them such that they could study for exams with the tool.

From an instruction perspective we have been able to give students 4-5 times as much homework and at the same time have drastically reduced the amount of code we have to hand evaluate. In our own courses we have an attached lab where students do weekly projects. Over the course of a semester students will complete roughly 12 lab projects which are graded by lab instructors and will complete 100-200 coding assignments in JavaGrinder. Our pedagogical strategy with JavaGrinder has been simple – practice is the key to success in CS1.

With a policy where students can attempt a problem as many times as they like we initially expected all students would receive 100% credit for JavaGrinder coding exercises and appropriately weighted these exercises as representing only 5% of a students overall grade.

We were surprised to find that some students would complete only a few (and often no) problems in assigned JavaGrinder sets. Ability to complete all of the JavaGrinders was closely linked with success in the class. We interviewed students who didn't do the JavaGrinders and

the story was uniformly the same – they didn't understand how to do the problems. These students would not put in time practicing the early problems and as the course becomes more difficult, when they attempted later problems these students found they were unable to complete them. Since JavaGrinder provides some simple event logging, it was easy to determine how often students would attempt sets and this reinforces our opinion that practice is key.

An open question this raises is that with such accessible practice resources available how do we motivate students that think they don't need to practice to do it? One thing we have done is make the JavaGrinders worth more in hopes that students who think they can "blow it off" and it won't hurt their grade will take it seriously.

Beyond these qualitative interviews and quantitative assessments of course grade and JavaGrinder scoring, we are currently involved in a multi-semester study of the effectiveness of JavaGrinder with respect to core skills and self-efficacy. One feature JavaGrinder support is teaming and pair-programming. Pair programming is a form of extreme programming where one team member plans and guides solution development while the other participant actively implements the solution[27]. JavaGrinder can directly support this kind of *teamwork* with on-screen instructions describing the methodology and directions telling students when to switch roles. JavaGrinder is also "aware" that more than one student is working on the project and each student gets credit. One of our initial studies suggest a positive student response to using the tool for pair programming and that it is effective in promoting positive self-efficacy and improved grades echoing similar findings in other studies[28,29].

6. Service Based Education

JavaGrinders design is based on a web service model (specifically using RESTful interfaces). While this helps support our rich AJAX web interface, it also enables a philosophy that supports taking education into other contexts. While the JavaScript version of JavaGrinder is our primary educational product, we have also developed an Objective-C client designed for iOS based devices such as the iPad (See Figure 5). The iOS version of the product includes its own Java interpreter, which supports evaluating projects using our testing framework so students have the same instant feedback experience that they do using the web based version of JavaGrinder even when disconnected from the Internet.

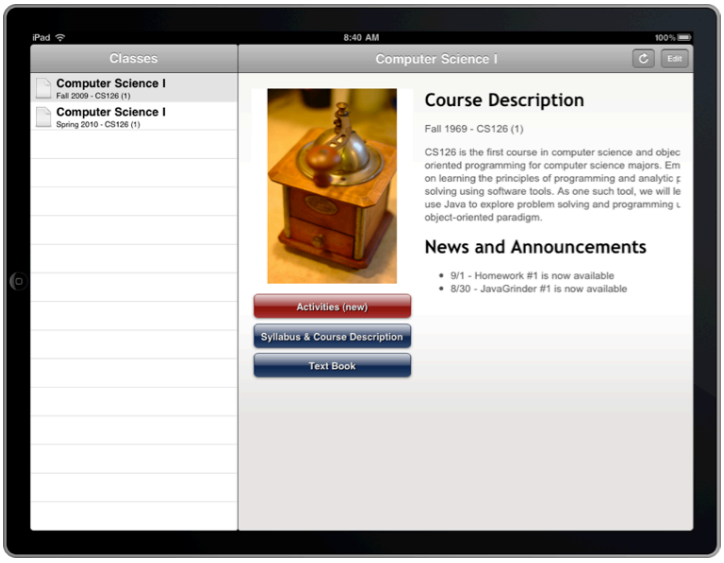

Figure 5. JavaGrinder for the iPad offers similar functionality to the JavaScript client but using iPad specific metaphors and interfaces.

We have also taken advantage of this service based approach to develop an interactive textbook for the class that links problems in the text directly to the JavaGrinder environment. Students can practice the problems as they read the book. See Figure 6.

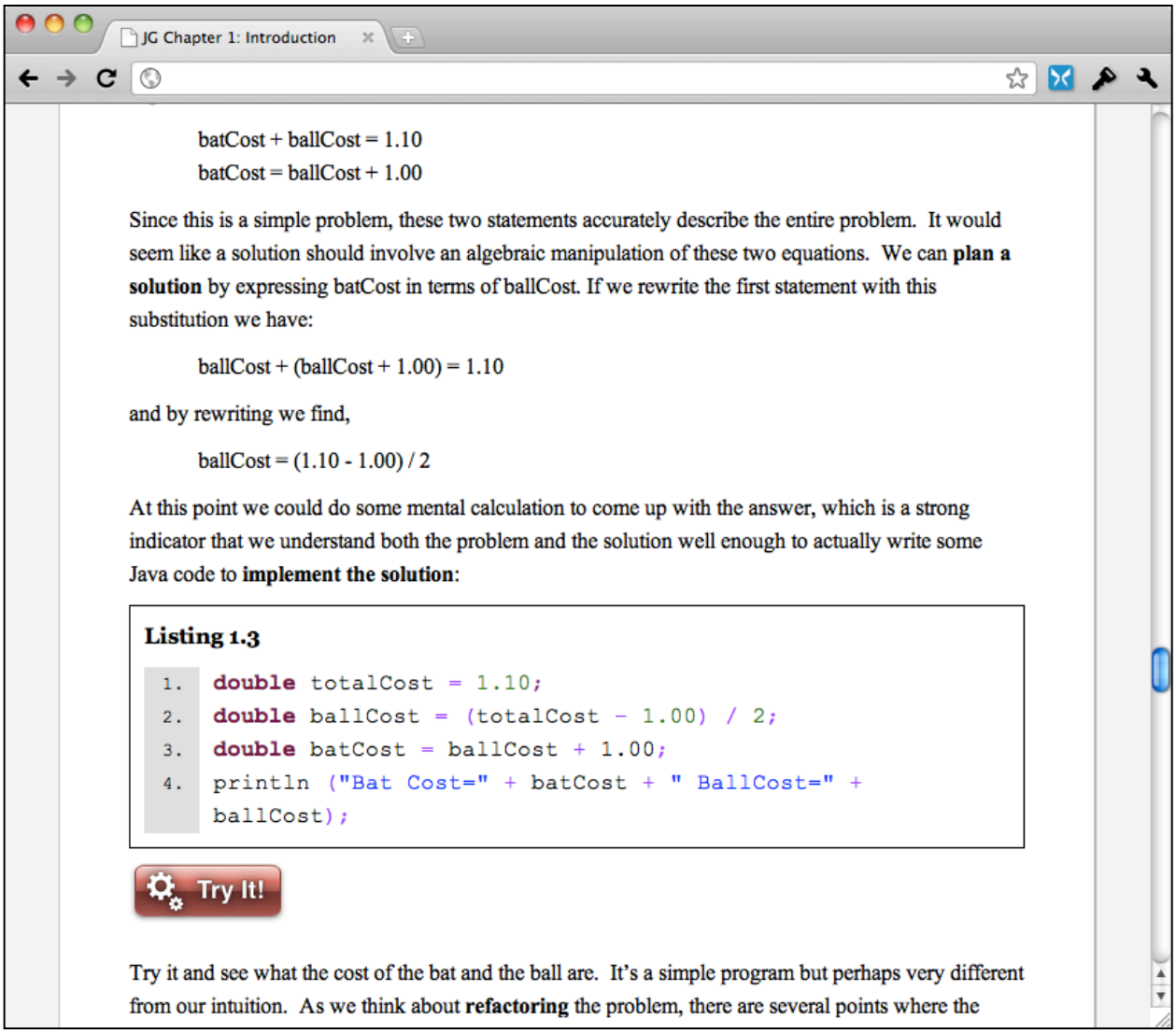

Figure 6. "Learning Computer Science with JavaGrinder" is a textbook we have developed that links the text to the JavaGrinder environment.

7. Future Work

The work described in this paper is ongoing. We are in the process of analyzing student self-efficacy and in measuring student success and satisfaction in the context of JavaGrinder. While our core mission with JavaGrinder is to support a web-based approach to learning Java, we have begun developing a mobile off-line solution for devices like the iPad. What is emerging is an ecosystem of JavaGrinder related service that can span across the client and the cloud to support different capabilities for different learning platforms.

In the very near future we plan to open the system up for other universities and organization to use both the infrastructure and learning materials we have developed. We hope to then leverage this to do a more comprehensive multi-site evaluation of our platform.

National Science Foundation Grant IEECI 08-610 provided funding for some of JavaGrinder's development and support for research in this paper. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Bibliography

1. Department of Labor, Bureau of Labor Statistics. *Occupational Outlook Handbook*. 2010th ed. Government Printing Office; 2010.

2. Department of Labor, Bureau of Labor Statistics. *Occupational Outlook Quarterly*. 2009th ed. Government Printing Office; 2009.

3. Kalwarski T, Mosher D, Paskin J, Rosato D. 50 Best Jobs In America. *Money*. 2006;35(5).

4. Best Jobs in America. *Money*. 2010;39(11).

5. Zweben S. Computing Degree and Enrollment Trends. 2010.

6. Denning PJ. The field of programmers myth. *Commun. ACM*. 2004;47(7):15–20.

7. Margolis J, Fisher A. *Unlocking the Clubhouse: Women in Computing*. MIT Press; 2003.

8. Felleisen M, Findler RB, Flatt M, Krishnamurthi S. The DrScheme project: an overview. *SIGPLAN Not.* 1998;33:17–23.

9. Allen E, Cartwright R, Stoler B. DrJava: a lightweight pedagogic environment for Java. In: *ACM SIGCSE Bulletin*. SIGCSE '02. New York, NY, USA: ACM; 2002:137–141.

10. Barnes DJ, Kölling M. *Objects first with Java: a practical introduction using BlueJ*. Pearson/Prentice Hall; 2009.

11. Kölling M. Using BlueJ to Introduce Programming. In: Bennedsen J, Caspersen M, Kölling M, eds. *Reflections on the Teaching of Programming*.Vol 4821. Lecture Notes in Computer Science. Springer Berlin / Heidelberg; 2008:98-115.

12. Kouznetsova S. Using BlueJ and Blackjack to teach object-oriented design concepts in CS1. *J. Comput. Small Coll.* 2007;22:49–55.

13. Patterson A, Kölling M, Rosenberg J. Introducing unit testing with BlueJ. In: *ACM SIGCSE Bulletin*.Vol 35. New York, NY, USA: ACM; 2003:11–15.

14. Henriksen P, Kölling M. greenfoot: combining object visualisation with interaction. In:

Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. OOPSLA '04. New York, NY, USA: ACM; 2004:73–82.

15. Kölling M. Greenfoot: a highly graphical ide for learning object-oriented programming. In: *ACM SIGCSE Bulletin*.Vol 40. New York, NY, USA: ACM; 2008:327–327.

16. Moskal B, Lurie D, Cooper S. Evaluating the effectiveness of a new instructional approach. In: *ACM SIGCSE Bulletin*.Vol 36. New York, NY, USA: ACM; 2004:75–79.

17. Cooper S, Dann W, Pausch R. Teaching objects-first in introductory computer science. In: *ACM SIGCSE Bulletin*. SIGCSE '03. New York, NY, USA: ACM; 2003:191–195.

18. Hartness K. Robocode: using games to teach artificial intelligence. *J. Comput. Small Coll.* 2004;19:287–291.

19. Hartness K, Culver B. Robocode: a fun way to learn to program. *J. Comput. Small Coll.* 2004;19:348–348.

20. Kelleher C, Pausch R. *Lowering the barriers to programming: a survey of programming environments and languages for novice programmers*. School of Computer Science, Carnegie Mellon University; 2003.

21. Vegso J. Enrollments and Degree Production at US CS Departments Drop Further in 2006-07. *Computing Research News*. 2008;20(2).

22. Job Migration Task Force. Globalization and Offshoring of Software Aspray W, Mayadas F, Vardi MY, eds. 2006.

23. Morsch L. What America's Top Jobs Pay. 2008.

24. Denning PJ, McGettrick A. Recentering computer science. *Communications of the ACM*. 2005;48:15–19.

25. National Science Board. Moving Forward to Improve Engineering Education. 2007.

26. Gates B. How to Keep America Competitive. *The Washington Post*. 2007.

27. Williams LA, Kessler RR. All I really need to know about pair programming I learned in kindergarten. *Commun. ACM*. 2000;43(5):108-114.

28. McDowell C, Werner L, Bullock HE, Fernald J. Pair programming improves student retention, confidence, and program quality. *Commun. ACM*. 2006;49(8):90-95.

29. Werner LL, Hanks B, McDowell C. Pair-programming helps female computer science students. *J. Educ. Resour. Comput.* 2004;4(1):4.