Rethinking Computer Science Education from a Test-first Perspective

Stephen H. Edwards
Virginia Tech, Dept. of Computer Science
660 McBryde Hall, Mail Stop 0106
Blacksburg, VA 24061 USA
+1 540 231 5723
edwards@cs.vt.edu

ABSTRACT

Despite our best efforts and intentions as educators, student programmers continue to struggle in acquiring comprehension and analysis skills. Students believe that once a program runs on sample data, it is correct; most programming errors are reported by the compiler; when a program misbehaves, shuffling statements and tweaking expressions to see what happens is the best debugging approach. This paper presents a new vision for computer science education centered around the use of test-driven development in all programming assignments, from the beginning of CS1. A key element to the strategy is comprehensive, automated evaluation of student work, in terms of correctness, the thoroughness and validity of the student's tests, and an automatic coding style assessment performed using industrial-strength tools. By systematically applying the strategy across the curriculum as part of a student's regular programming activities, and by providing rapid, concrete, useful feedback that students find valuable, it is possible to induce a cultural shift in how students behave.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.5 [**Software Engineering**]: Testing and Debugging—testing tools.

General Terms

Verification.

Keywords

Pedagogy, test-driven development, laboratory-based teaching, CS1, extreme programming.

1. INTRODUCTION

Many educational institutions are undergoing significant curriculum changes as they embrace object orientation, often opting for an aggressive objects-first strategy for its pedagogical value [25, 32, 26, 6]. Yet, while such changes offer the promise of eliminating the paradigm shift that would face students who receive initial training in procedural programming, other age old difficulties remain [28, 15]. Particularly during freshman and sophomore courses, and occasionally much later, a student may believe that once the code she has written compiles successfully, the errors are gone. If the program runs correctly on the first few runs she tries, it must be correct. If there is a problem, maybe by switching a few lines around or tweaking the code by trial and error, it can be fixed. Once it runs on the instructor-provided sample data, her program is correct and the assignment is complete. Even worse, students are often able to succeed at simpler CS1 and CS2 assignments without developing a broader view, which only reinforces approaches that will handicap their performance in more advanced courses.

The reason for this, as described by Buck and Stucki [9, 10], is that most undergraduate curricula focus on developing program *application* and *synthesis* skills (i.e., writing code), primarily acquired through hands-on activities. In addition, students must master basic *comprehension* and *analysis* skills [8]. Students must be able to read and comprehend source code, envision how a sequence of statements will behave, and predict how a change to the code will result in a change in behavior. Students need explicit, continually reinforced practice in hypothesizing about the behavior of their programs and then experimentally verifying (or invalidating) their hypotheses. Further, students need frequent, useful, and immediate feedback about their performance, both in forming hypotheses and in experimentally testing them.

To this end, I propose a new vision for laboratory and programming assignments across the entire CS curriculum inspired by test-first development [4, 3]. From the very first programming activities in CS1, a student should be given the responsibility of demonstrating the correctness of *his or her own* code. Such a student is expected and required to submit test cases for this purpose along with the code, and assessing student performance includes a meaningful assessment of how correctly and thoroughly the tests conform to the problem. The key to providing rapid, concrete, and immediate feedback is an automated assessment tool to which students can submit their code. Such a tool should do more than just give some sort of "correctness" score for the student's code. In addition, it should:

- Assess the validity of the student's tests, giving feedback about which tests are incorrect.

- Assess the completeness of the student's tests, giving an indication of how to improve.
- Assess the style of the student's code, giving feedback about where improvements can be made.
- Assess the quality of the student's code, giving suggestions for improvement or drawing attention to potential problems.

This paper describes a vision of a test-first-inspired educational strategy: systematically supporting test-first programming from the beginning to ensure students acquire the necessary comprehension and analysis skills needed to support effective programming. It also describes a practical, feasible approach to providing automated feedback that students can really use. This approach will work even for very early programming assignments in CS1 classes, and naturally meshes with existing tools for teaching in an objects-first style. By systematically adopting such an assessment approach across the curriculum, it will be possible to induce a cultural shift in how students behave when completing programming assignments and what they expect to get out of the process.

Section 2 lays out the details of test-first assignments and their assessment, while Section 3 uses this foundation to describe a new vision for CS education. Related work is described in Section 4, with conclusions appearing in Section 5.

2. TEST-FIRST ASSIGNMENTS

Others have suggested that more software engineering concepts in general [29, 30] and software testing skills in particular [38, 20, 21, 22, 16] should be integrated across the undergraduate CS curriculum. Providing upper-division elective courses on such topics is helpful, but has little influence on the behaviors students practice throughout their academic endeavors. Instead, a student can easily view the software engineering practices in most student-oriented texts as something that professional programmers do "out in the real world" but that has little bearing on—and provides little benefit for—the day-to-day tasks required of a student.

Practicing test-driven development (TDD) across the curriculum is an interesting alternative. In TDD, one always writes a test case (or more) before adding new code. New code is only written in response to existing test cases that fail. By constantly running all existing tests against a unit after each change, and always phrasing operational definitions of desired behavior in terms of new test cases, TDD promotes incremental development and gives a programmer a great degree of confidence in the correctness of their code. While TDD is a practical, concrete technique that students can practice on their own assignments.

The idea of using TDD in the classroom is not revolutionary [2]. Instead, the real issue is how to overcome its potential pitfalls: the approach must be systematically applied across the curriculum in a way that makes it an inherent part of the programming activities in which students participate, and students must receive frequent, directed feedback on their performance that provides the student with clear benefits. The key to resolving these issues is a powerful strategy for assessing student performance.

2.1 Automated Grading

Providing appropriate feedback and assessment of student performance is the critical factor in the success of this vision. Instructors and teaching assistants are already overburdened with work. Doubling their workload by requiring them to assess test data as well as program code will never work. This issue is even more critical for a curriculum-wide transformation. The only practical answer is automation.

Many educators have used automated systems to assess and provide rapid feedback on large volumes of student programming assignments [19, 23, 31, 35, 18]. While these systems vary, they typically focus on compilation and execution of student programs against some form of instructor-provided test data. Indeed, Virginia Tech uses its own automated grading system for student programs and has seen powerful results.

In spite of its classroom utility, an automatic grading strategy like the one embodied in the Curator also has a number of shortcomings. Most importantly, students *focus on output correctness* first and foremost; all other considerations are a distant second at best (design, commenting, appropriate use of abstraction, testing one's own code, etc.). This is due to the fact that the most immediate feedback students receive is on output correctness, and also that the Curator will assign a score of zero for submissions that do not compile, do not produce output, or do not terminate. In addition, students are not encouraged or rewarded for performing testing on their own. In practice, students do less testing on their own, often relying solely on instructor-provided sample data and the automated grading system. Clearly, existing approaches to automatic grading of student programs will not work.

2.2 TDD-oriented Assessment

Instead of automating an assessment approach that focuses on the *output* of a student's program, instead we must focus on what is most valuable: the student's testing performance. The assessment approach should require a student test suite as part of every submission, and encourage students to write thorough tests. It should also support TDD by encouraging the rapid cycling of "write a little test, write a little code."

Virginia Tech has developed a prototype grading system to explore the possibilities in this direction, and has experimented with these techniques in the classroom with positive results. The prototype is a service provided by Web-CAT, the Web-based Center for Automated Testing.

Suppose a student is developing a programming assignment in Java. The student can prepare test cases in JUnit format [24]. The source files for the program and tests can be submitted to the Web-CAT Grader. Upon receipt, the student's submission is compiled and then assessed along four dimensions: correctness, test completeness, test validity, and code quality.

Assessing "correctness" is entirely the student's responsibility, and the percentage of student-written tests passed by the student's code is used for this measure. Student code is also instrumented to gather code coverage instrumentation, using a tool such as Clover [14]. The instructor can choose an appropriate coverage metric for the difficulty level of the course, and code coverage can be used as a measure of how thoroughly the student as tested the submitted code. Further, the instructor may wish to provide a separate reference test set—the percentage of tests in this reference set that are passed by the student submission can be used as an indicator of how thoroughly the student has tested all the behavior required in the problem.

Test validity is assessed by running the student tests against an instructor-provided reference implementation. In cases where the class design for the student's submission is tightly constrained, this may include unit-level test cases. As students move on to more comprehensive assignments, the test cases can be partitioned into those that test top-level program-wide behavior and those that test purely internal concerns. Only top-level test cases that capture end-to-end functionality are validated against the instructor's reference implementation.

Finally, industrial quality static analysis tools such as Checkstyle [11] and PMD [34] can assess how well the student has conformed to the local coding style conventions as well as spot potentially error-prone coding issues. Together, Checkstyle and PMD provide many dozens of fully automated checks for everything from indentation, brace usage, and presence of JavaDoc comments to flagging unused code, inappropriate object instantiations, and inadvisable coding idioms like using assignment operators in sub-expressions. The instructor has full control over which checks are enabled, which checks result in scoring deductions, and more.

To support the rapid cycling between writing individual tests and adding small pieces of code, the Web-CAT Grader will allow unlimited submissions from students up until the assignment deadline. Students can get feedback any time, as often as they wish. However, their score is based in part on the tests they have written, and their program performance is only assessed by the tests they have written. As a result, to find out more about errors in their own programs, it will be necessary for the student to write the test cases. The feedback report will graphically highlight the portions of the student code that are not tested so that the student can see how to improve. Other coding or stylistic issues will also be graphically highlighted.

2.3 But Can It Be Used Across the Board?

While the idea of automatically assessing TDD assignments is exciting, it also raises questions when one proposes to apply it curriculum-wide. The two biggest questions are: can beginning students use it from the start of their first class, and will it work on graphically-oriented programs?

First, consider beginning students. Most automated grading systems, including the current system in use at Virginia Tech, were designed to help cope with the large volumes of students in introductory-level classes. The previous Curator system has been in use in our CS1 course for many years and has not caused issues in that regard. So the real question is whether or not students can *write test cases* from the start of CS1.

Interestingly, DrJava [1], which is designed specifically as a pedagogical tool for teaching introductory programming, provides built-in support to help students write JUnit-style test cases for the classes they write. Similarly, BlueJ [25, 26, 27], another introductory Java environment designed specifically for teaching CS1, also supports JUnit-style tests. BlueJ allows students to interactively instantiate objects directly in the environment without requiring a separate main program to be written. Messages can be sent to such objects using pop-up menus. BlueJ's JUnit support allows students to "record" simple object creation and interaction sequences as JUnit-style test cases. Such tools make it easy for students to write tests from the beginning.

```
import cs1705.*;

/**
 * MyRobot adds three basic capabilities to a
 * robot: the ability to turn right, turn com-
 * pletely around, and pick up a row of beepers.
 */
public class MyRobot
    extends VPIRobot
{
    //-------------------------------------------
    /** Construct a new MyRobot object.
     */
    public MyRobot()
    {
    }

    public void turnRight()
    {
        turnLeft();
        turnLeft();
        turnLeft();
    }

    //-------------------------------------------
    /** Reverse direction with a 180-degree turn
     */
    public void turnAround()
    {
        turnLeft();
        turnLeft();
    }

    //-------------------------------------------
    /** March along a line of beepers, picking up
     *  each in turn.
     */
    public void collectBeepers()
    {
        while ( nextToABeeper() );
        {
            pickBeeper();
            if ( frontIsClear() )
            {
                move();
            }
        }
    }
}
```

Figure 1. A simple student program.

Further, the scoring formula used to grade introductory assignments by beginners will most likely be different than that used for more advanced students. To start, the instructor may wish to only require method-level coverage of beginning students (i.e., each method is executed at least once). As students grasp the concept and develop experience applying the feedback they receive, grading stringency can be gradually increased.

But will this technique work for graphically-oriented programs? As long as a batch-oriented test execution scheme can be devised, the solution is appropriate. Buck and Stucki describe a simple approach for achieving the same end with graphically-oriented student programs [9]. By fixing the interface between the GUI and the underlying code, the GUI can be replaced by an alternate driver during testing. Instructors who use custom GUI libraries designed for educational use can augment them with additional support for test automation if needed. We have successfully applied automated grading techniques to a variety of courses from the freshman through the junior level with success, including some courses that use graphically-oriented projects.

2.4 An Example

To show how TDD assignments work, consider a case that pushes the boundaries: a freshman in CS1 is learning the basics of programming on a graphically oriented assignment. Many institutions use variations of Karel the Robot because of the consistent and intuitive metaphor it provides to introductory students. There are several Java versions of Karel the Robot [5, 7, 10], some of which allow student to "program" Karel by writing pure Java.

Karel is a simple mobile robot that navigates in a two-dimensional grid-based world. Karel supports a simple set of messages to move forward, detect walls directly in front of him, turn left, and pick up or put down small beepers in his environment. Students can easily grasp the concept of Karel as well as the basic operations he provides, and their programs are easily animated in a graphical window to visualize the robot's actions.

Figure 1 shows the source code for a hypothetical Karel assignment: create a robot that provides three new capabilities: turning right (the base robot only knows how to turn left!), reversing direction, and picking up a sequence of beepers. A student completing this assignment may begin with a sample robot class in a text book or provided by the instructor.

What kind of test case might a CS1 student write for this assignment? Suppose the student is working on gathering beepers first. Figure 2 shows a simple JUnit-style test case that might be created as a student works on `collectBeepers()`. The student might even create this sequence interactively and record it as a test case using their educational IDE. The student could then submit code and test case together for assessment. The student could continue to develop test cases for each new feature or change, using repeated submissions to get feedback on his or her progress.

Figure 3 depicts the feedback report the student would receive from the Web-CAT Grader. This report is for a submission where all of the student's tests pass. It shows a summary of the correctness and testing assessment, which in this example is taken from the Clover code coverage measure—the number of methods executed in this case, since students for this assignment are not yet ready for more stringent requirements. The bar graphs in the report were inspired in part by JUnit's GUI TestRunner: "when the bar is green the code is clean."

Figure 3 also shows a summary of the stylistic assessment, where points have been deduced for stylistic or coding errors. There is also room for a design and readability score from the TA or instructor. In this example, the code has not yet been manually assessed. Further, a more detailed breakdown lists each class in the submission separately, showing the number of comments or remarks on the corresponding source file, the points lost attributable to that class, and a summary of how thoroughly that particular class has been tested. By showing the basic testing coverage achieved for each component in this way, the top-level summary indicates to the student where more effort can be productively spent to improve their understanding of the code and to ensure it operates correctly. This list is initially sorted by the number of comments received, although the student can resort the list using other criteria if desired.

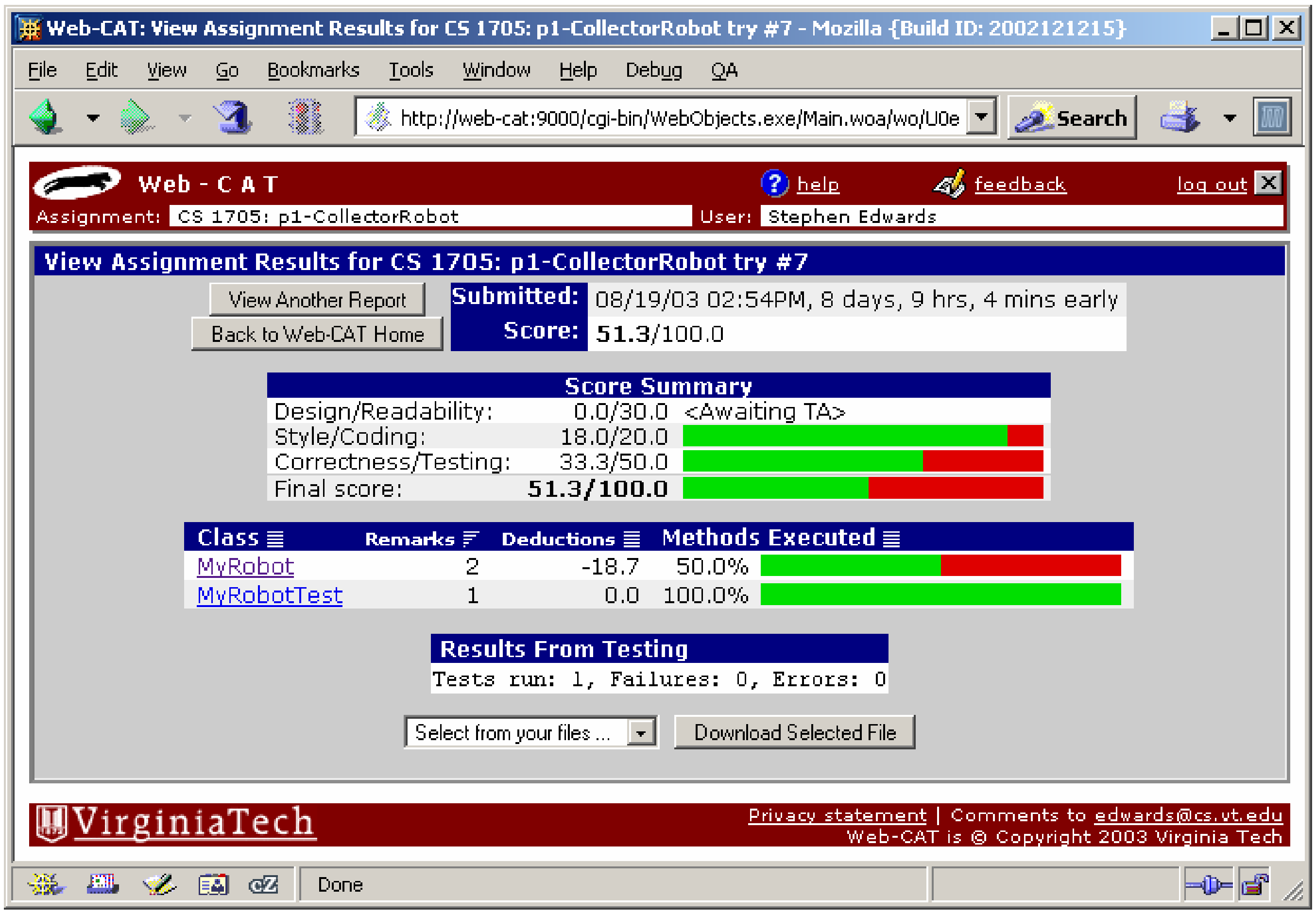
```
import cs1705.*;

public class MyRobotTests
    extends junit.framework.TestCase
{
    MyRobot karel;
    World   world;

    protected void setUp()
    {
        // Read in a world config containing
        // a line of beepers at karl's start loc
        World.startFromFile( "beeperTest.kwld" );
        karel = new MyRobot();
        world = karel.getWorldAsObject();
    }


    //-------------------------------------------
    /** Check that after calling collectBeepers(),
     *  there are no more beepers left.
     */
    public void testCollectBeepers()
    {
        karel.collectBeepers();
        karel.turnAround();
        karel.turnOff();
        karel.assertBeepersInBeeperBag();
        world.assertNoBeepersInWorld();
    }
}
```

Figure 2. A simple test case for `MyRobot`.

The student can click on a class name to view the suggestions and comments on that portion of his or her code. Figure 4 shows an example screen shot of "marked up" source code that the student will see. The basic form of the report is produced by Clover, and each source file is viewble in pretty-printed form with color-highlighted markup and embedded comments or remarks. This top-level summary shows the basic testing coverage achieved for each component, indicating to the student where more effort can be productively spent to improve their understanding of the code and ensure it operates correctly.

From this summary, individual reports for each file in the submission can be obtained, as exemplified in Figure 4. Clover automatically highlights lines that have not been executed during testing in pink to graphically indicate where more testing needs to be performed. In addition, an execution count for each line is listed next to the line number on the left. Hovering the mouse over such lines pops up more detailed information about the amount of full or partial coverage achieved on troublesome lines.

In addition, comments from static checking tools (e.g., Checkstyle and PMD) have been folded into this unified report. Lines highlighted in red indicate stylistic or coding issues resulting in point deductions. In Figure 4, line 18 is so marked, and the corresponding message is shown immediately below the line, in this case indicating that the method is missing a descriptive comment. Alternate colors and icons are used to denote warnings, suggestions, good comments from the TA or instructor, and extra credit items.



Web-CAT: View Assignment Results for CS 1705: p1-CollectorRobot try #7 - Mozilla {Build ID: 2002121215}

http://web-cat:9000/cgi-bin/WebObjects.exe/Main.woa/wo/U0e

Web - C A T

Assignment: CS 1705: p1-CollectorRobot User: Stephen Edwards

View Assignment Results for CS 1705: p1-CollectorRobot try #7

Submitted:	08/19/03 02:54PM, 8 days, 9 hrs, 4 mins early
Score:	51.3/100.0

Score Summary		
Design/Readability:	0.0/30.0	<Awaiting TA>
Style/Coding:	18.0/20.0	
Correctness/Testing:	33.3/50.0	
Final score:	**51.3/100.0**	

Class	Remarks	Deductions	Methods Executed
MyRobot	2	-18.7	50.0%
MyRobotTest	1	0.0	100.0%

Results From Testing

```
Tests run: 1, Failures: 0, Errors: 0
```

VirginiaTech

Privacy statement | Comments to edwards@cs.vt.edu
Web-CAT is © Copyright 2003 Virginia Tech

Figure 3. The score summary a student receives for a submission.

In Figure 4, line 40 is also highlighted as an error, with two associated messages. The execution count next to the line number indicates that lots of processing time was spent here—the accidental infinite loop was terminated by the execution time limit imposed for this assignment. The messages draw attention to the misplaced semicolon, helping to solve the issue in this case.

The Web-CAT Grader also provides an interface for TAs to review assignments. Using a direct manipulation interface, comments resulting from manual grading can be directly entered via a web browser. TA comments entered this way will be visible to the student just as tool-generated comments.

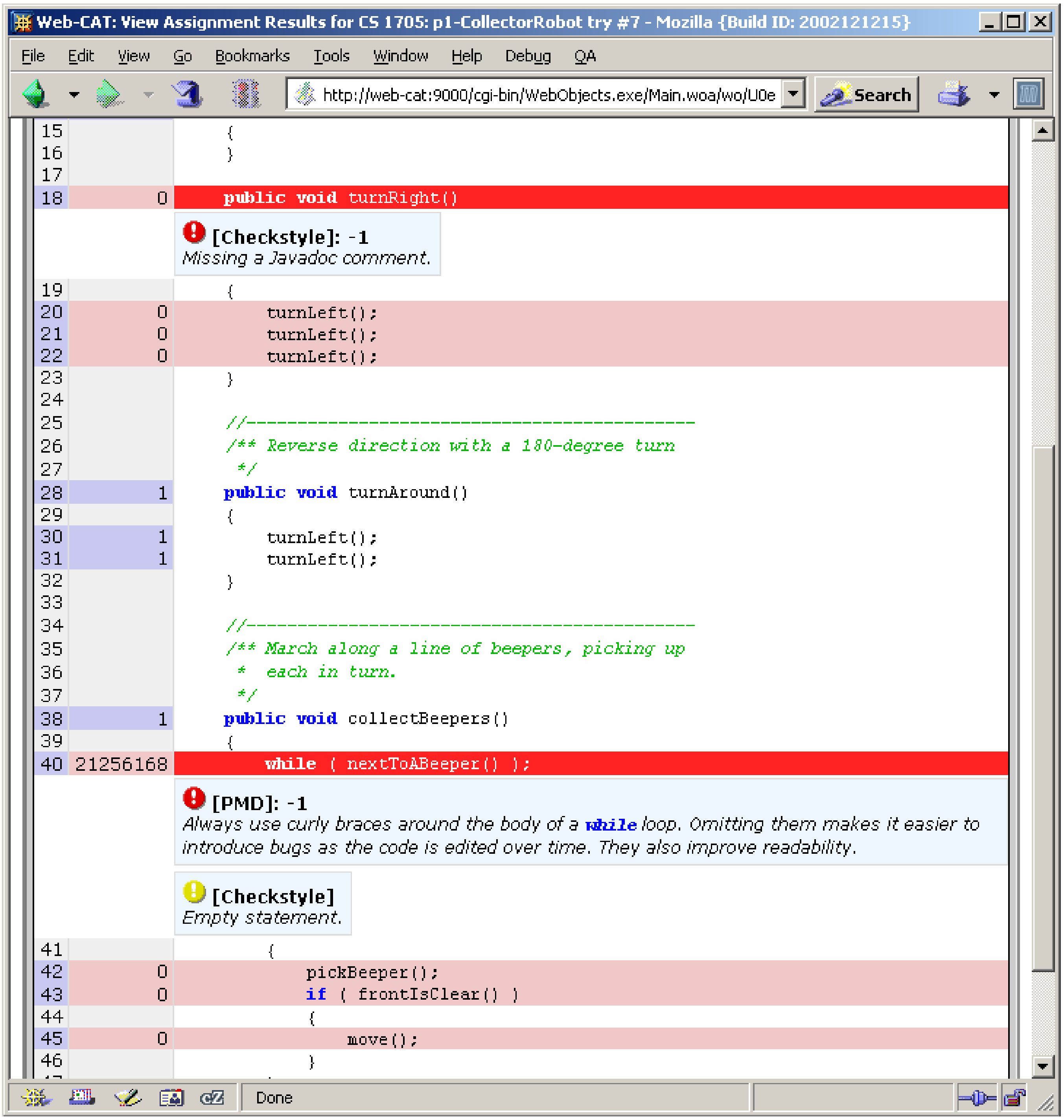
2.5 How Are Students Affected?

TDD is attractive for use in education for many reasons. It is easier for students to understand and relate to than more traditional testing approaches. It promotes incremental development, promotes the concept of always having a "running (if incomplete) version" of the program on hand, and promotes early detection of errors introduced by coding changes. It directly combats the "big bang" integration problems that many students see when they begin to write larger programs, when testing is saved until all the code writing is complete. It increases a student's confidence in the portion of the code they have finished, and allows them to make changes and additions with greater confidence because of continuous regression testing. It increases the student's understanding of the assignment requirements, by forcing them to explore the gray areas in order to completely test their own solution. It also provides a lively sense of progress, because the student is always clearly aware of the growing size of their test suite and how much of the required behavior is already "in the bag" and verified.

Most importantly, students begin to see these benefits for themselves after using TDD on just a few assignments. The Web-CAT Grader prototype and TDD have been used in a junior-level class. Compared to prior offerings of the class using a more traditional automated grading approach, students using TDD are more likely to complete assignments, are less likely to turn assignments in late, and receive higher grades. Empirically, it also appears that student programs are *more thoroughly tested* (in terms of the branch coverage their test suites achieve on a reference implementation) than when using the previous automated grading system.

3. A NEW VISION FOR CS EDUCATION

Given the example in Section 2.4, it is clear that TDD-based assignments with comprehensive, automated assessment are feasible, even for introductory students. In addition, this strategy can be combined easily with many recent advances in CS pedagogy.



Web-CAT: View Assignment Results for CS 1705: p1-CollectorRobot try #7 - Mozilla {Build ID: 2002121215}

http://web-cat:9000/cgi-bin/WebObjects.exe/Main.woa/wo/U0e

```
                {
                }

        0       public void turnRight()
```

[Checkstyle]: -1
Missing a Javadoc comment.

```
                {
        0           turnLeft();
        0           turnLeft();
        0           turnLeft();
                }

                //------------------------------------------------
                /** Reverse direction with a 180-degree turn
                 */
        1       public void turnAround()
                {
        1           turnLeft();
        1           turnLeft();
                }

                //------------------------------------------------
                /** March along a line of beepers, picking up
                 *  each in turn.
                 */
        1       public void collectBeepers()
                {
 21256168           while ( nextToABeeper() );
```

[PMD]: -1
Always use curly braces around the body of a **while** *loop. Omitting them makes it easier to introduce bugs as the code is edited over time. They also improve readability.*

[Checkstyle]
Empty statement.

```
                    {
        0               pickBeeper();
        0               if ( frontIsClear() )
                        {
        0                   move();
                        }
```

Figure 4. Style and coding suggestions for one student source file.

Students can be taught using an objects-first style [5, 6, 12, 13, 37, 32], and introduced to programming using metaphorical systems like Karel the Robot [7, 13, 37]. Role-playing activities [7] can be used to introduce OO concepts and act out testing tasks. Closed laboratory sessions can be used to provide more hands-on learning. Pair programming can be used in closed labs to increase peer-to-peer learning and also to foster comprehension and analysis skills [33, 39]. Bloom's taxonomy can be used to plan the order in which topics are introduced and the manner in which programming tasks are framed as students progress in their abilities [9, 10].

As students gain more skill from early courses, requirements for test thoroughness can be increased. Unlike prior automated grading systems that tend to inhibit student creativity and enforce strict conformance to an unwavering assignment specification, the TDD approach more readily allows open-ended assignments such as those suggested by Roberts [36]. If a student wishes to do more work or implement more features, they can still write their own internal tests. As long as they also implement the minimum requirements for the assignment as embodied in the instructor's reference test suite, their submission will be graded on the thoroughness of their own testing against their enhanced solution.

After students have used TDD techniques across several classes, it will become the cultural norm for behavior, not just an extra requirement that one instructor imposes and that can be "thrown away" after his or her class has been passed. The goal is to foster this cultural shift for pedagogical ends. By continually requiring students to test in the small, every time they add or change a piece of code, they are also continually practicing and increasing their skills at hypothesizing what the behavior should be and then operationally testing those hypotheses. This will truly bring the "laboratory" nature of computer science training to the fore if this vision is adopted across an institution's curriculum.

4. RELATED WORK

The vision described here builds on a large body of prior work. Infusing software engineering issues and concerns across the undergraduate curriculum has been discussed at SIGCSE on several occasions [17, 29, 30]. TDD and other extreme programming ideas have even been used in the classroom [2]. This idea is complementary to the test-first assignment strategy described here. The main difference is that the TDD strategy focuses on operational techniques that provide clear benefits to students in a way that is natural part of the programming process and that can be applied across the curriculum.

The idea of including software testing activities across the curriculum has also been proposed by others [16, 20]. Jones has described some experiences in this direction [21, 22]. While Jones has used a traditional automated grading system for assessing student work [23], his system is similar to others in that it focuses on assessing program correctness first and foremost. This paper proposes TDD rather than more traditional testing techniques and focuses specifically on the unique assessment issues necessary for fostering a positive cultural change in student behavior.

Automated grading has also been discussed in the educational literature [19, 35, 18]. Unfortunately, most such systems are of the "home brew" variety and see little or no use outside their originating institution. Further, virtually all focus on output correctness as the sole assessment criterion. Mengel describes experiments in using metrics-based techniques to assess style [31]. Here, the intent is to use of industrially proven tools. By installing and configuring these tools on a server and combining them with a unified feedback format, students can readily take advantage of the information they provide without being exposed to the hassles of installing and learning to use the tools.

5. CONCLUSION

Despite the best efforts of computer science educators, CS students often do not acquire the desired analytical skills that they need to be successful until later than we would like, if at all. Reassessing typical computer science education practices from a test-first perspective leads one to focus on programming activities and how they are carried out. It is possible to infuse continual practice and development of comprehension and analysis skills across the programming assignments in a typical CS curriculum using TDD activities. Providing a system for rapid assessment of student work, including both the code and the tests they write, and ensuring concrete, useful, and timely feedback, is critical. In addition to assessing student performance, students can get real benefits from using the approach, and these benefits are important for students to internalize and use the approach being advocated.

Using TDD across the board can serve as the core for a broader vision of re-engineering programming practices across the CS curriculum. The goal is to develop a culture where students are expected to test their own code (that is, apply analytical and code understanding skills on a daily basis), and where it is an accepted part of life across all of a student's courses. Instead of being the exception—i.e., testing is something students do in one class focused on the topic—testing one's own code will become the norm. As students become inculcated with this expectation, it is possible to emphasize testing across the curriculum as a natural part of existing classes, without requiring extra class time or lecture materials. The hope captured in this vision is that students will acquire better skills for a variety of programming tasks, that instructors and TAs will be able to devote more attention to design assessment (because simple stylistic, correctness, and testing issues are automatically assessed), and thus more teaching time and effort can go into the deeper issues that all students must master once they conquer their programming fundamentals.

6. ACKNOWLEDGMENTS

This work is supported in part by the Virginia Tech Institute for Distance and Distributed Learning and by the National Science Foundation under grant DUE-0127225. Any opinions, conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the NSF. I wish to acknowledge the feedback provided by Manuel Pérez-Quiñones on these ideas, and the students who have worked on the project: Anuj Shah, Amit Kulkarni, and Gaurav Bhandari.

7. REFERENCES

[1] Allen, E., Cartwright, R., and Stoler, B. DrJava: a lightweight pedagogic environment for Java. In *Proc. 33rd SIGCSE Technical Symp. Computer Science Education*, ACM, 2002, pp. 137-141.

[2] Allen, E., Cartwright, R., and Reis, C. Production programming in the classroom. In *Proc. 34th SIGCSE Technical Symp. Computer Science Education*, ACM, 2003, pp. 89-93.

[3] Beck, K. Aim, fire (test-first coding). *IEEE Software*, 18(5): 87-89, Sept./Oct. 2001.

[4] Beck, K. *Test-Driven Development: By Example*. Addison-Wesley, Boston, MA. 2003.

[5] Becker, B.W. Teaching CS1 with Karel the Robot in Java. In *Proc. 32nd SIGCSE Technical Symp. Computer Science Education*, ACM, 2001, pp. 50-54.

[6] Bergin, J., et al. Resources for next generation introductory CS courses: report of the ITiCSE'99 working group on resources for the next generation CS 1 course. *ACM SIGCSE Bulletin*, 31(4): 101-105.

[7] Bergin, J., Stehlik, M., Roberts, J., Pattis, R. Karel J. Robot: A Gentle Introduction to the Art of Object-Oriented Pro-

gramming in Java. http://csis.pace.edu/~bergin/KarelJava2ed/

[8] Bloom, B.S., et al. *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*. Longmans, Green and Co., 1956.

[9] Buck, D., and Stucki, D.J. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. In *Proc. 31st SIGCSE Technical Symp. Computer Science Education*, ACM, 2000, pp. 75-79.

[10] Buck, D., and Stucki, D.J. JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. In *Proc. 32nd SIGCSE Technical Symp. Computer Science Education*, ACM, 2001, pp. 16-20.

[11] Checkstyle home page. http://checkstyle.sourceforge.net/.

[12] Comer, J., and Roggio, R. Teaching a Java-based CS1 course in an academically-diverse environment. In *Proc. 33rd SIGCSE Technical Symp. Computer Science Education*, ACM, 2002, pp. 142-146.

[13] Cooper, S., Dann, W., and Pausch, R. Teaching objects-first in introductory computer science. In *Proc. 34th SIGCSE Technical Symp. Computer Science Education*, ACM, 2003, pp. 191-195.

[14] Clover: a code coverage tool for Java. http://www.thecortex.net/clover/.

[15] Decker, R. and Hirshfield, S. The top 10 reasons why object-oriented programming can't be taught in CS 1. In *Proc. 25th Annual SIGCSE Symp. Computer Science Education*, ACM, 1994, pp. 51-55.

[16] Goldwasser, M.H. A gimmick to integrate software testing throughout the curriculum. . In *Proc. 33rd SIGCSE Technical Symp. Computer Science Education*, ACM, 2002, pp. 271-275.

[17] Hilburn, T.B., and Towhidnejad, M. Software quality: A curriculum postscript? In *Proc. 31st SIGCSE Technical Symp. Computer Science Education*, ACM, 2000, pp. 167-171.

[18] Isong, J. Developing an automated program checker. *J. Computing in Small Colleges*, 16(3): 218-224.

[19] Jackson, D., and Usher, M. Grading student programs using ASSYST. In *Proc. 28th SIGCSE Technical Symp. Computer Science Education*, ACM, 1997, pp. 335-339.

[20] Jones, E.L. Software testing in the computer science curriculum—a holistic approach. In *Proc. Australasian Computing Education Conf.*, ACM, 2000, pp. 153-157.

[21] Jones, E.L. Integrating testing into the curriculum—arsenic in small doses. In *Proc. 32nd SIGCSE Technical Symp. Computer Science Education*, ACM, 2001, pp. 337-341.

[22] Jones, E.L. An experiential approach to incorporating software testing into the computer science curriculum. In *Proc. 2001 Frontiers in Education Conf. (FiE 2001)*, 2001, pp. F3D7-F3D11.

[23] Jones, E.L. Grading student programs—a software testing approach. *J. Computing in Small Colleges*, 16(2): 185-192.

[24] JUnit home page. http://www.junit.org/.

[25] Kölling, M. and Rosenberg, J. Guidelines for teaching object orientation with Java. In *Proc. 6th Annual Conf. Innovation and Technology in Computer Science Education*, ACM, 2001, pp. 33-36.

[26] Kölling, M. and Rosenberg, J. BlueJ—the hitchhiker's guide to object orientation. Maersk Mc-Kinney Moller Institute for Production Technology, Univ. Southern Denmark, Tech. Report 2002, No. 2, ISSN No. 1601-4219. http://www.mip.sdu.dk/~mik/papers/hitch-hiker.pdf.

[27] Kölling, M. BlueJ—The Interactive Java Environment. http://www.bluej.org/.

[28] Krause, K.L. Computer science in the Air Force Academy core curriculum. In *Proc.13th SIGCSE Technical Symp. Computer Science Education*, ACM, 1982, pp. 144-146.

[29] McCauley, R., Archer, C., Dale, N., Mili, R., Robergé, J., and Taylor, H. The effective integration of the software engineering principles throughout the undergraduate computer science curriculum. In *Proc. 26th SIGCSE Technical Symp. Computer Science Education*, ACM, 1995, pp. 364-365.

[30] McCauley, R., Dale, N., Hilburn, T., Mengel, S., and Murrill, B.W. The assimilation of software engineering into the undergraduate computer science curriculum. In *Proc. 31st SIGCSE Technical Symp. Computer Science Education*, ACM, 2000, pp. 423-424.

[31] Mengel, S.A., Yerramilli, V. A case study of the static analysis of the quality of novice student programs. In *Proc. 30th SIGCSE Technical Symp. Computer Science Education*, ACM, 1999, pp. 78-82.

[32] Mitchell, W. A paradigm shift to OOP has occurred … implementation to follow. *J. Computing in Small Colleges*, 16(2): 95-106.

[33] Nagappan, N., Williams, L., Ferzli, M., Wiebe, E., Yang, K., Miller, C., and Balik, S. Improving the CS1 experience with pair programming. In *Proc. 34th SIGCSE Technical Symp. Computer Science Education*, ACM, 2003, pp. 359-362.

[34] PMD home page. http://pmd.sourceforge.net/.

[35] Reek, K.A. A software infrastructure to support introductory computer science courses. In *Proc. 27th SIGCSE Technical Symp. Computer Science Education*, ACM, 1996, pp. 125-129.

[36] Roberts, E. Strategies for encouraging individual achievement in introductory computer science courses. In Proc. 31st SIGCSE Technical Symp. Computer Science Education, ACM, 2000, pp. 295-299

[37] Sanders, D., and Dorn, B. Jeroo: a tool for introducing object-oriented programming. In *Proc. 34th SIGCSE Technical Symp. Computer Science Education*, ACM, 2003, pp. 201-204.

[38] Shepard, T., Lamb, M., and Kelly, D. More testing should be taught. *Communications of the ACM*, 44(6): 103–108, June 2001.

[39] Williams, L., Upchurch, R.L. In support of student pair-programming. In *Proc. 32nd SIGCSE Technical Symp. Computer Science Education*, ACM, 2001, pp. 327-331.