

# Experiences Using Automated Tests and Test Driven Development in Computer Science I

Carol A. Wellington, Thomas H. Briggs, and C. Dudley Girard  
Shippensburg University  
*cawell@ship.edu, thbrig@ship.edu, cdgira@ship.edu*

## Abstract

*We are interested in how to expose our students to Test Driven Development (TDD) and have experimented with a variety of ways of leveraging testing technology to help our students learn to program in our first programming course. Initially, we developed a framework that allows the students to run tests that are developed by the faculty member. That experience led us to developing a JUnit plug-in that allowed the students to specify the tests without having to write the test code. As a result of these experiences, we have re-structured this class into these roughly sequential phases: learning to read code, learning to write code, and learning to program. Throughout this course, the students are using TDD, writing their own JUnit tests, and refactoring as they develop their code iteratively. This change has been made without dropping any of the required course content.*

## 1. Introduction

Recent years have brought significant changes in testing technologies and those technologies are changing the way software is developed. At Shippensburg University, we have been experimenting with how these technologies and techniques can impact our Computer Science I course.

Some research on using automated testing in Computer Science I has already been done. For example, the University of Kansas[1] encouraged the students to use asserts as embedded tests of their C++ code. Also, Stephen Edwards[2] has developed a framework to help students build thorough tests along with their code.

The course that this paper refers to is the first course our majors take and also serves as a service course to

math and science majors. Some of the students have had one or two programming courses in high school, but most of the students have no programming experience at all. We are using Java in the Eclipse IDE and have traditionally used an “objects-first” approach that very much matched the ACM guidelines for that approach. However, through the experiences this paper covers, we have migrated to more of an “objects early” approach.

We defined TDD as writing a test before you write the code to make the test pass. In other words, we follow the Red, Green, Refactor mantra. As this technique is relatively new to our faculty, it seemed too advanced for freshmen to undertake. In addition, this course is quite full with content that the second course depends on, so we didn’t think we had time to teach TDD in full. Therefore, we tried a number of strategies that used the tests and some of the philosophies of TDD before we fully embraced TDD for this course.

## 2. Blind Testing

In our first experiment, we were trying to provide the students with the ability to run automated tests without requiring them to know anything about JUnit. We built a framework that allowed the faculty member to build JUnit tests and let the students use those tests to evaluate their code.

In this approach, students were assigned a series of labs. Each lab defined a problem and a set of criteria. For example, one program involved making change using the minimum number of coins. The specifications were rigid in that the students had to implement a set of methods like *int getDimes()*.

The framework consisted of two types of tests: structural and functional. For structural tests, the framework used java’s introspection capabilities to ensure that the student’s code matched the specifications required by the lab. If these tests failed,

the framework reported how their code did not meet the specifications. For functional tests, the framework ran a series of tests checking on the values returned by particular methods under particular situations. For example, the lab about making change might have a test where it calculates the change for 45 cents and the `getDimes` method should return two. If these tests failed the framework would report the method that returned the incorrect value.

With the goal of understanding the types of errors the students were making, we wanted to track their usage of this framework. In addition, we wanted to let the students submit their code to this framework as many times as they liked. Therefore, students submitted their source code through a web page. Their code was compiled and the set of tests were run. Students received a status indicating whether their code passed or failed and indicated the result of each test. For each submission, the framework tracked who the student was and the results of that submission.

When we analyzed the statistics gathered by this framework, we saw some interesting results. First, the students were submitting their code multiple times and trying to get the tests to pass. Second, the number of structural test failures was significantly less than the number of functional test failures. In other words, the students understood the mechanics of writing code, but were having trouble with the problem solving strategies required to produce the necessary functionality. In addition, when they had functional test failures, the odds of them being repaired between any two submissions was surprisingly low. In the most extreme case, one student desperately submitted nearly identical code over one hundred times, each time, modifying the return value of `getDimes()` by one, in the hopes of passing the tests.

Observing the students working on a lab during class gave us an insight into the source of the problem. When they had a functional problem, the students were making changes without any reasoning justifying those changes. In these cases, we observed very little problem solving thought processes.

After this review, we decided that, because this blind approach did not allow the students to develop their own test cases, when their programs failed, the students had limited information about the causes. So, while the student had the tests before they started writing their code and could run them any number of times, they didn't understand how the tests were created or why their programs failed. In effect, they

were missing an important part of TDD: that writing the tests first makes you think about what you are going to code. This significantly limited their strategies for correcting the problems. As a result, the students tended to make changes to the code that seemed almost random.

Our conclusion from this experiment was that, if the students are going to reap the benefits of TDD, they had to at least understand the tests. While we had structured this blind framework to hide complexity from the students, we had made the tests almost magical and eliminated the students' ability to reason about how the tests were interacting with their code.

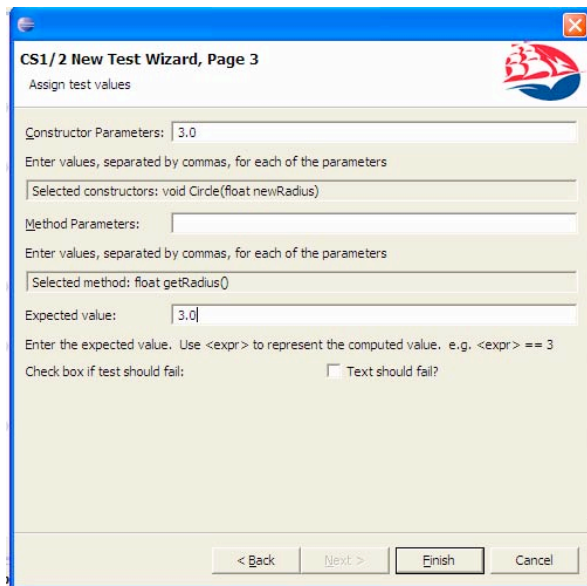
### 3. Eclipse Plug-in for Test Development

In our next attempt to introduce TDD to freshmen, we decided that the students needed to have exposure to the tests. As we considered writing tests to be an extra level of complexity, we developed some tools to help the students write the tests. We developed a set of plug-ins for the Eclipse environment that provide wizards to guide the students through the development of their code. The goal was to reduce the complexity of code generation, while not reducing the problem solving aspects of early CS courses. This way the students could think about the tests first without having to consider the details of writing the code.

The first wizard supplants the existing new class wizard that comes with Eclipse. This wizard asks students to plan out the class, including any variables and methods. The wizard generates an empty framework class, including the defined variables, each given a default value, an accessor and a mutator for each of these variables, and then any additional methods defined by the user.

The second wizard in our plug-in is shown in Figure 1. This wizard guides students through the creation of a single JUnit test case. The wizard first asks the user to select the class to test and a new or existing class to hold the test case. Then the wizard uses the JDT framework of Eclipse to query the compilation units defined in the selected class and presents the user with a list of known constructors and a list of known methods. The wizard allows the user to select a constructor and a method to test. Finally, the wizard prompts the user to enter values for the parameters to be passed to the constructor and the selected method. The wizard generates the necessary classes, modifies the project's build path to include JUnit, and automatically imports the JUnit packages into the test

class. It also generates the JUnit test code, constructing the object with the given parameters, calling the method and comparing the results with the given expected value.



**Figure 1 - Plug-In Wizard**

While the tests generated by this plug-in are of a very specific structure, they are sufficient for early CS1 projects. Using this wizard, students are free to generate tests without worrying about generating code. This helps our students to focus on problem solving and the behavior they want the code to exhibit before worrying about the implementation.

In developing this plug-in, we were trying to hide the complexities of the tests from students while allowing the students to develop the tests. However, when the plug-in was complete, it became apparent that the students had to know almost as much to use the plug-in as to write the tests even though the types of tests the plug-in could build were limited. We did learn that we could limit the types of tests students would need because the types of programs students write in this class are limited. This means that, instead of needing to learn all of the complexities of JUnit, we could control that scope pretty significantly. Lastly, in the longer term, our students are going to need to write true JUnit tests, so teaching them a limited intermediate framework seemed cumbersome. At the completion of this exercise, we were finally ready to expose all of JUnit to the students.

## 4. True TDD by the Students in CS1

In fall of 2006, we were finally ready to allow students to experience TDD in full. However, in order to do that, we had to make some significant changes to the structure of the course.

### 4.1 Learning to Read Code First

In order to give our students the perspective required by TDD, we started by teaching them to read code. In fact, in the second day of class, they were in the debugger in class, stepping through code to see how the code and variables work. The example that we use for this activity is code that outputs the lyrics to “One Hundred Bottles of Sprite on the Wall.” The students step through a while loop and conditionals and play with the values of variables to change the behavior.

We spend the first two weeks reading and modifying a number of example programs. By the end of this period, the students have a good understanding of how the control structures of Java work, the differences between primitive and reference variable types, and the use of Strings.

We had one section of this course that attempted TDD without teaching the students to read code first. Without this understanding, the teaching of TDD becomes a catch-22. For example, it is very difficult to introduce loops using TDD when the students do not understand loops or how they are used. In these cases, the tests tended to show up after the code (or in subsequent labs) and this inconsistency undermined the “test first” part of the philosophy.

### 4.2 Learning to Write Code

While the students get a good overview of control structures and variables by reading code, only by actually writing code using those constructs do the students really begin to see the subtleties of how those structures work. In the past, our students have practiced developing code with these control structures in the context of a particular problem. Our students completed a number of labs where they built small systems that required the use of these control structures. However, because they were building semi-functional systems (with user input and output), the number of control structure they actually authored was pretty small.

With an eye toward TDD, we decided to use JUnit tests to address this problem. Because students had not yet

```

/**
 * For this test, we are calculating taxes based on incomes. There are
 * three tax rates (0%, 5%, and 7% depending on the size of the income.
 */

public void testThreeRanges()
{
    ConditionalTricks ct = new ConditionalTricks();
    assertEquals(0, ct.threeRange(29999.99, 0.001);
    assertEquals(30000*0.05, ct.threeRange(30000), 0.001);
    assertEquals(49999.99*0.05, ct.threeRange(49999.99), 0.001);
    assertEquals(50000*0.07, ct.threeRange(50000), 0.01);
}

```

**Figure 2 - Example Drilling Test**

written any code, we wanted the faculty to write the tests. However, learning from our previous experience, we wanted the students to be able to see the details of the tests. Therefore, we developed a series of labs in which the students write code to make faculty-developed JUnit tests pass. These tests are designed to make the students develop increasingly complex combinations of the control structures. Figure 2 shows an example of a test from the conditionals lab:

Clearly, these labs will require the students to write significantly more examples of the control structures than they had in the past. In addition, the students became very comfortable with Eclipse, its debugging tools and JUnit. The early labs explicitly instruct the students on how to use these tools and the students develop strong diagnostic skills.

However, there are some challenges that we have not yet been able to resolve with these labs. First, the students quickly find the “Quick Fix” capability in Eclipse when they are trying to build enough code to make the tests compile. Unfortunately, they usually just pick the first fix Eclipse proposes instead of really thinking about which fix is most appropriate. This often leads them astray. Sometimes they even let Quick Fix change the code in the test instead of in their code

In addition, Quick Fix often makes the methods used in assertEquals statements return Objects (because the primitive variables are being auto-boxed into the parameters of assertEquals). Getting inexperienced freshmen to pay attention to these details can be quite a challenge.

These labs take about two weeks to complete, so by the end of four weeks, students are comfortable with

primitive and reference variables and writing control structures. However, these exercises are somewhat contrived; the students still need some of the traditional labs to see where control structures fit into programs as a whole.

### 4.3 Learning to Write Programs

The drilling labs expose students to classes and objects. The next challenge is inheritance. At this point, the students are comfortable enough with the IDE, reading JUnit tests, and writing code snippets, so we decided they were ready to be responsible for all aspects of TDD. Again, they develop these skills through a series of carefully structured labs. The early labs in this sequence lead the students through the TDD process and show the red, green, refactor cycle in detail. As the labs progress, the students become increasingly responsible for that process.

The first TDD lab has some characteristics that are critical to the success of TDD for these students. In the lab, the students develop two classes: Song and Album. Songs have names and durations and an Album has a name and an array of songs. More experienced developers would be able to code this directly, but our students need more detailed plans in order to know where they are going. In class, we lead the students through the creation of the design of these classes and develop a UML diagram showing the methods and fields of each of the classes. Notice that we are demonstrating that iterative development does not mean that you start without a plan. Developing that architectural vision and vocabulary will be an ongoing part of our students’ education.

In the lab, the students are practicing these skills:

The first test to write is the one that tests the constructor. First, declare the method that will hold the test:

```
public void testConstructor()  
{  
}  
}
```

When we create an album, we only need to check that the title and that the album contains no songs. Here is a line-by-line description of the test:

1. Create an instance of an album whose title is "Heavier Things" which can contain up to 3 songs
2. check the title of the album
3. check that the album contains no songs
4. Create another instance of an album whose title is "A Night at the Opera" which can contain up to 5 songs
5. check the title of the album
6. check that the album contains no songs

Once you have the test written, use quick fix to build enough of the Album class to make it compile (remember to be careful about which fix you choose, the return types, and the names of the variables). Run the test and watch it fail.

#### **Make it pass - one step at a time**

Look at the error message the test gave you. It should be something like this:

```
junit.framework.ComparisonFailure: expected:<Heavier Things> but was: <null>
```

This is because we aren't storing the title or returning it in the getter (getTitle). Fix those two problems and re-run the test.

**SURPRISE** - it passes! Now, we know that we haven't really built everything the constructor needs (we haven't declared or allocated the array that will hold the songs) and we didn't write the correct code for getNumberOfSongs (it always returns zero which I hope won't always work!). We have tested the parts of the constructor that are externally visible. We'll add the rest of the code when we have a test that needs it.

**Figure 3 - Example of Early TDD Instructions**

- Use of instance variables by constructors, getters, and setters,
- Manipulating strings and numbers in toString methods, and
- Using arrays and for loops in the Album.

Since the only JUnit tests the students have seen up to this point are from the drilling exercises, we provide them with the tests for the Song class. The first part of the lab is spent getting those tests to pass.

The second part of the lab is where the true TDD begins. In these labs, we are leading the students through the TDD process very carefully. The

instructions for the first test they develop are shown in Figure 3. We specifically point out the functionality that should be exercised by each test and lead them through any refactorings required. At this point in their development, they do not have the skills to break development up into small slices or to detect when refactoring is necessary. These labs also make sure that the students see how Eclipse can assist in refactoring, code generation, and code formatting.

The instructions for these labs also contain a significant amount of instruction about the TDD process. For example, Figure 4 shows how the first lab ends. We are explicitly pointing out the structure and benefits of TDD.

### What's the Point?

Review the strategies we've used to develop this code. First, we worked on things a little bit at a time. Thinking about the whole system might have been overwhelming and there'd be a lot of details to keep track of. We can keep things in control by only worrying about smaller pieces. Second, writing the code has followed a specific pattern: write a test, watch it fail and then write the code to make it pass. This let's us be sure that the little pieces of code that we are writing really works. Also, since we're always running all of the tests, we know that the code we just added didn't break any of the code we had previously written.

**Figure 4 - Example of Refactoring Instructions**

As the semester progresses, the labs become less structured, providing less support, and helping the students develop their own responsibility for the TDD process.

#### 4.4 Maturing TDD

Throughout the course, we move between covering material in class and laboratory activities, but some of the class material is presented in the context of demonstrating TDD. For example, as part of covering inheritance and polymorphism in class, the instructor leads the class through a TDD exercise in the development of a banking system with a variety of account and transaction types. These activities are free flowing with the students participating while the code is projected for everyone to see.

These open class activities are a great way to demonstrate the thought processes behind the development of code. In addition, it lets the students see that code is not written linearly – that all systems evolve and code changes are not unexpected. This is a revelation to many students. One student said, “I thought I didn’t know what I was doing because I couldn’t write the code without making mistakes. Then I watched you do the same thing in front of class, but you didn’t think you had made any mistakes.”

Using TDD to demonstrate new concepts in CS1 is key for the students. As one professor found, if they don’t see you using TDD when you write code, they don’t see it as an integral part of writing code and that undermines their motivation for using it.

#### 4.5 Refactoring

Since our labs are using iterative development, it is critical that our students learn about refactoring. Early in the labs, they perform simple refactorings like renaming variables, but the labs encourage them to use

Eclipse’s tools to complete those refactorings. This lays the groundwork for the students to be able to complete more complex refactorings later in the semester. Figure 5 has an example of the instructions related to more complex refactorings.

Refactoring exercises also help reinforce the usefulness of having a complete test suite. The students begin to understand that they can clean up the code with confidence because the tests will catch any mistakes they make.

### 5. Conclusions

While we have only offered this version of CS1 once, we have made some important observations about its effects. One of the challenges of this course is the range of capabilities that the students have at the beginning of the semester. For the students who have some programming experience in high school, TDD gives them something new to learn. In addition, for many of them, this is the first time that they are exposed to the concept of design quality. That underscores the image that a computer science degree has something significant to offer them.

For our weaker students, the surprising result was that writing the tests gave them something they were good at. The logic in the tests is generally simpler than the logic in the code and many of the students who were struggling with the rest of the projects were exceptional at developing strong tests. In addition, getting tests to pass shows these students they are making progress even if they do not complete the entire lab.

It was interesting to see the students carry the concepts of iterative development into other areas of their lives. When struggling with a paper, one student decided to write it “test first.” She developed a rubric that she thought represented what the professor wanted in the

Moving the variable named "value" into the Transaction class certainly improved our code because it reduced some duplication. However, we still have one batch of duplication to clean up. Look at the constructors of Withdrawal, Interest, and Deposit. They set the value of the field named "value." Since that field has been moved into the superclass, setting it in the constructor ought to be in the superclass, too. Use Refactor->Change Method Signature to add a parameter to the constructor of Transaction. Make the default value be "amount" (the name of the parameter we're going to want to pass when we call this constructor from the subclasses). That will make every subclass constructor pass both the date and the amount to the superclass constructor. Notice that your javadoc comments have a TODO - fix that up! Re-run the tests to make sure they are still green.

**Figure 5 - Refactoring Example Instructions**

paper and then wrote the paper until she got an A on that rubric. She said that it gave her many of the benefits that TDD gave her: it kept her focused and gave her direction when she was floundering and it let her know when the paper was finished. Similarly, another student talked about "refactoring" a draft of a paper instead of "editing" it.

At the end of the first semester, our students have to take a programming competency exam (PCE) that is a pre-requisite for the next course. Even if they pass CS1, they cannot enroll in Data Structures until they pass the PCE. In the semester this course was offered, the PCE gave the students two options for each question: their code could be driven by user input or by automated tests. We think it is significant that every student in the TDD section chose to develop automated tests for their solutions. Many of those students are continuing to use JUnit and TDD even though it is not explicitly required in the subsequent courses.

The PCE results show one more significant result. Even though we have covered a significant amount of material about unit testing, TDD, iterative development, and refactoring, the students in that section performed similarly on the PCE to the students in the other sections. This is strong evidence that we

covered all of the other required course content. In other words, since TDD was integral to the course and presented as a tool to learn the other content of the course, we continued to be able to cover the same amount of normal CS1 content.

## 6. References

[1] David S. Janzen, Hossein Saiedian, "Test-Driven Learning: Intrinsic Integration of Testing into the CS/CE Curriculum", *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 2006, pages 254-258,

[2] Stephen H. Edwards, "Rethinking computer science education from a test-first perspective", In *Addendum to the 2003 Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (Educator's Symposium)*, ACM, 2003, pp. 148-155.

[1] A.B. Smith, C.D. Jones, and E.F. Roberts, "Article Title", *Journal*, Publisher, Location, Date, pp. 1-10.

[2] Jones, C.D., A.B. Smith, and E.F. Roberts, *Book Title*, Publisher, Location, Date.