TOOLS AND TECHNIQUES FOR
TEST-DRIVEN LEARNING IN CS1

ABSTRACT
Test-Driven Development is a design strategy where a set of tests over a class is defined prior to the implementation of that class. The goal is to use the tests toexercise the class being developed, to provide immediate feedback of the overall quality of the code, and to identify and correct bugs as they are written. Testing is usually performed with automated testing tools, such as JUnit, which give clear feedback about the status of the tests1. The test-first approach requires students to think about the problem, identify corner cases, analyze ways their code may fail, and evaluate the goodness of their design. This paper presents a tool for teaching CS1 using a “Test-First” approach that will allow students to construct their tests using a simple GUI interface. The goal is to give CS1 students the ability to do test-driven development independently.

INTRODUCTION
Students in introductory programming classes typically learn experientially. The traditional process is one where students enter an existing piece of code into an editor, and after getting the code to compile and run, they frequently are satisfied that the code is correct. Test-Driven Learning (TDL) [2] is a learning approach designed to encourage students to evaluate the correctness of their code by first thinking about what kinds of errors can happen, and then developing a set of tests to ensure that those errors are not present in their code. TDL encourages students to first think about the behavior of their program and then develop a set of tests to ensure their implementation is consistent with the given problem.

TDL applies the tenets of Test-Driven Development (TDD), a topic generally discussed in more advanced software engineering or agile development classes. To get a feel for how students in an introductory course would take to test-first development, the concept was introduced to students as part of a lab in a CS1 class taught in the fall 2005. From that lab experience it was determined that students we able to adapt to the test-first approach for small problems, and that developing tests caused students to think about what tests would be useful before coding, and in general, the quality of their submissions for this lab improved from past and successive labs. The other conclusion from this lab is that students could use existing JUnit code, but they had great difficulty in writing their own JUnit code.

In this paper, we present the organization for a CS1 course which uses a Test-Driven Learning approach. The course uses the Eclipse IDE 2, which is a powerful editor for Java and other languages. A unit-testing framework, JUnit 3 is integrated into the Eclipse framework. To mitigate the amount of material students must master to start using tests, we developed a set of plug-ins to the Eclipse environment which implement Wizards to guide students through class and test creation.

The course presented in this paper is offered as part of the Computer Science program. Our institution is a mid-sized, regional, masters and liberal arts institution. The course is a four credit course, and the first of three core courses in the programming and fundamentals tract of our program. The department uses an objects-first approach, and uses Java as a programming language. Our goal is to introduce TDL into the existing course to improve student understanding, within the resource constraints of our institution.

1For more information on Test-Driven Development see: en.wikipedia.org/wiki/Test_driven_development
2www.eclipse.org
3junit.sourceforge.net
This paper will discuss how students have used tests to validate their code at our institution. Additionally, the paper will introduce our new plugin for Eclipse that will allow students to easily construct JUnit tests for their code. The paper will address our TDL approach to CS1 in four sections. First, a short background section on TDL, TDD, past experience, and the tools. Following that a section covering how to get the students to think test-first as they write their first program and details on the plugin. After which it will be discussed how JUnit can be used within the later stages of a CS1 course so that students can start writing their own tests. Lastly, the paper will conclude with a discussion on the hopes of the course and how an evaluation will be done as to the effectiveness of test-first as a learning tool for CS1.

BACKGROUND

Test Driven Development represents an evolutionary approach to code development. TDD is a key component of extreme programming and agile development, and is necessary to facilitate the refactoring stages of that approach. TDD and its related methods have been shown to improve code quality in production environments [1]. In a broad sense, TDD is a tool that encourages designers to think about the utility of their design before they implement the final code.

The TDD process starts with a specification and the class structure is defined (e.g. using UML). A set of tests is written to test various aspects of the class structure. During this stage, the programmer is focused on using the class to be developed, and less on implementing it. From this perspective, deficiencies in the initial design may be identified, and the class can be revised with relative ease. After the tests are implemented, the code for the class to be tested is implemented. During implementation, the programmer makes a small set of changes, and re-runs the tests to check for new bugs. The full measure of these tools and approaches is found in large and complex systems, and is seemingly inappropriate for an introductory CS1 course.

In prior work, Janzen and Saiedian implemented their own test driven learning approach for CS1 [2]. They discovered that TDL showed the potential to improve student understanding of the material and make the test-first approach more natural to students. Their approach eased students into the test-first aproach, where by the end of the course they were able to code tests for unwritten code on their own.

The approach at Kansas used the assert function defined in Java 1.5 to evaluate the tests. The JUnit framework expands on the assert function to provide more powerful testing capabilities. Unfortunately, this gain comes at the cost of more information for the individual or student to learn.

Test-Driven Learning (TDL) uses the testing portion of the TDD approach (and typically skips refactorings and other agile methodologies). However, there are two key issues in TDL, namely students develop their own tests, and use an automated testing framework to execute those tests as they are writing code.

0.1 Blind Testing

In one of our previous CS1 courses, we used a blind testing framework to evaluate students code. There were some key differences between this course and the TDL approach, which based on student assessment, contributed to a negative outcome. 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 optimum number of coins. The specifications were rigid, in that students had to implement a set of methods such as `int getDimes()`.

The tests were developed by the course instructor. There were two types of tests, value based and introspection based. The first set of tests are intuitive. For the coin changing example, given a value, did they compute and return the correct number of coins. The second type of test used Java’s introspection capabilities to allow runtime binding and inspection of the students code. These tests ensured that students provided the specified program methods, and that there were no unnecessary public methods exposed.

In this scenario, students submitted their source code through a web page. Their code was compiled and the set of tests were applied. Students would receive a status indicating whether
their code passed or failed, and indicated the result of each test. A diagnostic message such as
\textit{test failed: getDimes() was incorrect} would be displayed.

This \textit{blind} approach did not allow students to develop their own test cases. Additionally,
when their program failed they only had limited information as to the cause(s). So while the
students had the tests first, they didn’t understand how the tests were created or why their
programs failed and therefore didn’t really understand what changes to make. The result was
students making changes without any reasoning behind them, rendering the tests useless and in
some cases detrimental. In the most extreme case, one student submitted nearly identical code
over one-hundred times, each time, modifying the return value of \textit{getDimes()} by one, in hopes of
passing the tests. By allowing students a true test-first approach these mistakes can be avoided.
We will now look tools for allowing students to run a set of tests in which they are given useful
feedback as to what needs to be fixed.

0.2 Tools

Eclipse is an open source project that provides the user with a powerful IDE for developing code.
The Eclipse framework is really an environment to implement an IDE, with Java being the most
common implementation. The Java perspective for Eclipse includes many features that make it
valuable in a CS1 class, such as \textit{quick fix} code help, dynamic compilation, syntax highlighting,
easy to use debugging tools, and dynamic code insertion during debugging. For example, as the
user is entering code, Eclipse compiles the code in the background. If it finds an error, the code
is decorated with a red underline to alert the programmer to syntax errors in his or her code.
In many cases, Eclipse will suggest ways to fix the error, a technique that is especially useful for
new programmers.

The Eclipse environment is divided into several subsystems, namely: core (platform,
Equinox, and SWT), the Java Development Tools (JDT), and the Plug-in Development Envi-
ronment (PDE). The API is designed to allow developers to make new plug-ins that can simply
add some small feature or support an new perspective for different programming languages. In
section 0.3 we use this feature to develop our own set of wizards to help students implement
classes and tests.

JUnit is a toolkit to support automated unit testing for Java source code. Eclipse includes
a tight integration with the JUnit framework, and enables the JUnit tests to be run within the
Eclipse editor itself. The result is that JUnit is very easy to use under Eclipse. JUnit allows the
programmer a simple means to code tests and evaluate the results of those tests by a bar that
is color coded red for failed and green for passed. For example, if a programmer wants to check
to see if a variable is equal to a specified value takes but one line of code (See \textit{Figure 3}). By
running the test the programmer gets two possible results: a green bar if passed or a red bar and
the message \textit{“HelloWorldStr does not equal Hello World.”}. Thus, if the test fails the students
what failed and therefore what to try and fix.

INTRODUCING TEST-FIRST TO STUDENTS

\begin{figure}[h]
\centering
\begin{verbatim}
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}\end{verbatim}
\caption{HelloWorld.java}
\end{figure}

Two key goals of our CS1 course is to teach students the basics of object oriented
programming (using Java) and teach them effective problem solving strategies. Students are
initially assumed to have no prior programming experience. As such, the first program the
students will tend to write is \textit{HelloWorld} or one very similar to it. This is also the first chance
to get the students thinking test-first, rather than code-first. Instead of having the students code
the \textit{HelloWorld} program and run it, the goal will be getting them to think: “Assuming the code
will run, how would you test the code to see if it ran correctly? This is where the first challenge presents itself.

```java
public class HelloWorld {
    public static String HelloWorldStr = "Hello World";
    public static void main(String[] args) {
        System.out.println(HelloWorldStr);
    }
}
```

Figure 2: HelloWorld.java

If we code HelloWorld.java as it is normally done in Figure 1, then as can be seen it is not possible to do any automated testing. This is because we have no variables or return values of methods to test their value of. The best that can be done is for the student to run the program and see if the output is correct. This, however necessary at times, is not a true test-first approach. However, if we code HelloWorld.java a little differently, as seen in Figure 2. Then the test for HelloWorld would look something along the lines of the code shown in Figure 3.

```java
import junit.testcase.*;
public class TestHelloWorld extends TestCase {
    public void testHelloWorldStr() {
        assertEquals("HelloWorldStr does not equal Hello World.",
                HelloWorld.HelloWorldStr, "Hello World");
    }
}
```

Figure 3: Test code for HelloWorld

All of this creates a pretty hefty addition of information to be piled onto the CS1 students. We have the introduction of variables, methods, more than one class, statements, subclasses, and import. Since we are planning test-first we must have the use of variables or a method that returns a value. Otherwise, we dont have anything to test. For the very early examples, the goal is to have the students think of the test and then have the instructor provide them the JUnit test class. Then by using Eclipse the instructor can have the students run the test and evaluate the results using the red/green bar as well as the output string from the test. This allows the students to have “constructed” the test before they write the code, and for the moment limit the amount of knowledge needed to create the tests.

### 0.3 TDL Plugin

One novel contribution to the Test-Driven Learning approach is the development of a set of plug-ins for the Eclipse environment which implement wizards to guide the students through the development and testing project. The goal of the wizards is to automate the code generation, not the problem solving aspects of early CS courses.

The first wizard supplants the existing new class wizard that comes with Eclipse. This wizard asks students to layout the class, including any variables and any methods. The wizard generates an empty framework class, including the defined variables, each given a default variable, an accessor and mutator for each of these variables, and then any additional methods defined by the user. The code is filled in with defaults so as to compile and be suitable for use by the test cases.

The novel aspect of this plug-in is that the code it creates is simple (e.g. everything is kept as public), and it provides an initial framework for students, who are new to programming, to get started with; while at the same time demanding that students provide the content for the generated class.

The TDL Wizard guides students through creating a test case. 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. Each of these tasks would likely be confusing to the new CS1 student.

The first panel of the wizard asks the user to select the class to test, and to select a new or existing class to hold the test case. The second panel of 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. The final panel (see figure 4) prompts the user to enter values for the selected constructor and method, and finally if the test should pass or fail. The wizard then generates the JUnit test code, constructing an object with the given parameters, calling the method, and comparing the results with the given expected value.

The TDL Wizard helps students implement tests by generating the code for them. Using this wizard, students are free to generate tests without worrying about generating code. This is important for those students who are having difficulty, as it enables them to think about the behavior, and not worry about the implementation.

MORE ADVANCED TESTING

As the semester progresses students in CS1 learn about how to use a number of programming constructs, in order, variables, math operators, methods, classes, conditional statements, looping, arrays, and inheritance and polymorphism. Recently, some objects first texts introduce methods and/or classes earlier, being taught before the math operators. Therefore once students understand what a method and class is and how to make calls on a method, it is possible to have the students start writing their own code for the JUnit tests.

```java
public class ComputeArea {
    public double computeCircleArea(double radius) {
    }
    public double computeSquareArea(double side) {
    }
}
```

For example, assume we have the program shown in Figure 5. This is an example that the students will create later in the semester. As before with the HelloWorld example, the instructor asks the students to think of what values it would be good to use as test values and what results to expect. In addition to getting the students to think about the results before coding, it is also an exercise to get the students thinking about boundary tests. Thus, if the students don’t suggest it, the instructor could ask the question: “What would happen if we passed in a negative radius?” These types of questions help get students thinking not just about valid input, but invalid input as well. This questioning also forces the students to compute the answers on their own before they code it. Hopefully giving them a better idea how to code the solution.

Once the students have determined what tests they want to run the instructor shows
Figure 6: TestComputeArea.java

test ComputeArea extends TestCase {

  public void testComputeCircleArea() {
    ComputeArea ca = new ComputeArea();
    assertEquals("Area Incorrect",
                ca.computeCircleArea(3), 28.27, 0.05);
    assertEquals("Area Incorrect",
                ca.computeCircleArea(-3), 28-.27, 0.05);
  }

  public void testComputeSquareArea() {
    ComputeArea ca = new ComputeArea();
    assertEquals("Area Incorrect",
                ca.computeSquareArea(3), 9, 0.05);
    assertEquals("Area Incorrect",
                ca.computeSquareArea(-3), 9, 0.05);
  }
}

them how to create the tests using the plugin. As the plugin assumes clear understanding of
variables and methods it is not immediately applicable to CS1 students. Additionally, by this
point in the course students should also be familiar with Eclipse making the learning of the use
of the plugin easier for them. Thus, fairly quickly students can generate the needed JUnit test
class, like that shown in Figure 6. Hopefully by first walking the students through writing
simple JUnit test they will understand the basics of how to write a JUnit test on their own.

DISCUSSION

The goal of this change in teaching CS1 is to improve on getting students to think more about the
program they are going to write and how it will function. By doing this it will help the students
improve their problem solving skills and better understand programming. By having the students
use JUnit more as a tool than writing additional coding through the use of our plugin will help
keep the new material at a minimum and give the students a useful way to see if their program
is working correctly. Also, the students would slowly develop the basic methodology that goes
with testing from a hands on approach. One downside to note of this approach is it doesnt lend
itself to having the students write simple GUIs.

Evaluating the success of this approach is in two parts. The first part is based on
the amount of material covered. If the new class fails to covered the required material then the
approach is not a success, since it took too much time away from the core topics. The second part
of the evaluation is determining if it improved students’ understanding of the material and ability
to problem solve. To evaluate this an existing evaluation tool will be utilized. At the end of
each CS1 course all students wanting to take CS2 must pass a programming competency exam.
If the approach is successful then at the minimum we would expect that the same percentage of
majors in CS1 take the exam as in the past and pass. We hope that this approach improves the
results of those taking the exam and the percentage taking it. We plan to apply this approach
in the fall of 2006 in two sections of CS1.

References

NCSU, September 06 2002.

into the csse curriculum. In SIGCSE ’06: Proceedings of the 37th SIGCSE technical symposium