Test-Driven Learning: Intrinsic Integration of Testing into the CS/SE Curriculum

David S. Janzen
University of Kansas
Electrical Engineering and Computer Science
Lawrence, Kansas USA
djanzen@eecs.ku.edu

Hossein Saiedian
University of Kansas
Electrical Engineering and Computer Science
Lawrence, Kansas USA
saiedian@eecs.ku.edu

ABSTRACT

Test-driven learning (TDL) is an approach to teaching computer programming that involves introducing and exploring new concepts through automated unit tests. TDL offers the potential of teaching testing for free, of improving programmer comprehension and ability, and of improving software quality both in terms of design quality and reduced defect density.

This paper introduces test-driven learning as a pedagogical tool. It will provide examples of how TDL can be incorporated at multiple levels in computer science and software engineering curriculum for beginning through professional programmers. In addition, the relationships between TDL and test-driven development will be explored.

Initial evidence indicates that TDL can improve student comprehension of new concepts while improving their testing skills with no additional instruction time. In addition, by learning to construct programs in a test-driven manner, students are expected to be more likely to develop their own code with a test-driven approach, likely resulting in improved software designs and quality.

Keywords

Test-driven learning, test-driven development, extreme programming, pedagogy, CS1

1. INTRODUCTION

Programmers often learn new programming concepts and technologies through examples. Instructors and textbooks use examples to present syntax and explore semantics. Tutorials and software documentation regularly present examples to explain behaviors and proper use of particular software elements. Examples, however, typically focus on the use or the interface of the particular software element, without adequately addressing the behavior of the element.

Consider the following example from the Java 1.5 API documentation:

```
void printClassName(Object obj)
{
  System.out.println("The class of " + obj +
          " is " + obj.getClass().getName());
}
```

While this is a reasonable example of how to access an object's class and corresponding class name, it only reveals the desired interface. It teaches nothing about the underlying behavior. To see behavior, one must compile and execute the code. While it is desirable to encourage students to try things out on their own, this can be time consuming if done for every possible example, plus it significantly delays the presentation/feedback loop.

As an alternative, we can introduce a simple automated unit test that demonstrates both the interface and the expected behavior. For instance, we could replace the above example with the following that uses the assert keyword[1]:

```
void testClassName1()
{
  ArrayList al = new ArrayList();
  assert al.toString().equals("[]");
  assert al.getClass().getName()
         .equals("java.util.ArrayList");
}
```

This example shows not only the same interface information as the original example in roughly the same amount of space, but it also shows the behavior by documenting the expected results.

[1]Although assert has existed in many languages for some time, the assert keyword was introduced in Java with version 1.4 and requires extra work when compiling and running:
javac -source 1.4 ClassTest.java
java -ea ClassTest

A second example below demonstrates the same interface using an `Integer`. Notice how these two examples also reveal the `toString()` results for an empty `ArrayList ("[]")` and an `Integer ("5")`.[2]

```
void testClassName2()
{
  Integer i = new Integer(5);
  assert i.toString().equals("5");
  assert i.getClass().getName()
         .equals("java.lang.Integer");
}
```

These examples demonstrate the basic idea of test-driven learning:

- Teach by example
- Present examples with automated tests
- Start with tests

Teaching by example has a double meaning in TDL. First TDL encourages instructors to teach by presenting examples with automated tests. Second, by holding tests in high regard and by writing good tests, instructors model good practices that contribute to a number of positive results. Students tend to emulate what they see modeled. So as testing becomes a habit formed by example and repetition, students may begin to see the benefits of developing software with tests and be motivated to write tests voluntarily.

The third aspect of TDL suggests a test-first approach. TDL could be applied in either a test-first or a test-last manner. With a test-last approach, a concept would be implemented, then a test would be written to demonstrate the concept's use and behavior. With a test-first approach, the test would be written prior to implementing a concept. By writing a test before implementing the item under test, attention is focused on the item's interface and observable behavior. This is an instance of the test-driven development (TDD) [4] approach that will be discussed in section three.

2. TDL OBJECTIVES

Teaching software design and testing skills can be particularly challenging. Undergraduate curriculums and industry training programs often relegate design and testing topics to separate, more advanced courses, leaving students perhaps to think that design and testing are either hard, less important, or optional.

This paper introduces TDL as a mechanism for teaching and motivating the use of testing as both a design and a verification activity, by way of example. TDL can be employed starting in the earliest programming courses and continuing through advanced courses, even those for professional developers. The lead author has integrated TDL into CS1 and a four-day C++ course for experienced professional programmers. Further, TDL can be applied in educational resources from textbooks to software documentation.

Test-driven learning has the following objectives:

- Teach testing for free
- Teach automated testing frameworks simply
- Encourage the use of test-driven development
- Improve student comprehension and programming abilities
- Improve software quality both in terms of design and defect density

[2]If the `toString()` information is deemed distracting, this first assert could simply be left out of the example.

Some have suggested that if objects are the goal, then we should start by teaching objects as early as the first day of the first class [2]. TDL takes a similar approach. If writing good tests is the goal, then start by teaching with tests. If it is always a good idea to write tests, then write tests throughout the curriculum. If quality software design is the goal, then start by focusing on habits that lead to good designs. Test-first thinking focuses on an object's interface, rather than its implementation. Test-first thinking encourages smaller, more cohesive and more loosely coupled modules [4], all characteristics of good design.

Examples with tests take roughly the same effort to present as examples with input/output statements or explanations. As a result, TDL adds no extra strain on a course schedule, while having the benefit of introducing testing and good testing practices. In other words TDL enables one to teach testing for free. It is possible that the instructor will expend extra effort moving to a test-driven approach, but once mastered, the instructor may find the new approach simpler and more reusable because the examples contain the answers.

By introducing the use of testing frameworks gradually in courses, students will gain familiarity with them. As will be seen in sections four and five, tests can use simple mechanisms such as assert statements, or they can utilize powerful frameworks that scale and enjoy widespread professional support. Depending on the language and environment, instructors may introduce testing frameworks early or gradually.

When students observe both the interface and behavior in an example with tests, they are likely to understand a concept more quickly than if they only see the interface in a traditional example. Further, if students get into the habit of thinking about and writing tests, they are expected to become better programmers.

3. RELATED WORK

Test-driven learning is not a radical new approach to teaching computer programming. It is a subtle, but potentially powerful way to improve teaching, both in terms of efficiency and quality of student learning, while accomplishing several important goals.

TDL builds on the ideas in Meyer's work on Design by Contract [12]. Automated unit tests instantiate the assertions of invariants and pre- and post-conditions. While contracts provide important and rigorous information, they fail to communicate and implement the use of an interface in the efficient manner of automated unit tests. Contracts have been suggested as an important complement to TDD [9]. The same could be said regarding TDL and contracts.

TDL is expected to encourage adoption of TDD. Although its name implies that TDD is a testing mechanism, TDD is as much or more about analysis and design as it is about testing, and the combination of emphasis on all three stands to improve software quality. Early research reports mixed results [10] regarding quality and productivity improvements from TDD particularly on small software projects, however recent research [7] suggests that a test-first approach increases the number of tests written and improves productivity, increasing the likelihood of higher quality software with similar or lower effort.

TDL was inspired by the Explanation Test [4] and Learning Test [4] testing patterns proposed by Kent Beck, Jim Newkirk, and Laurent Bossavit. These patterns were suggested as mechanisms to coerce professional programmers to adopt test-driven development.

The Explanation Test pattern encourages developers to ask for and provide explanations in terms of tests. The pattern even suggests that rather than explaining a sequence diagram, the explanation could be provided by "a test case that contains all of the externally visible objects and messages in the diagram." [4]

The Learning Test pattern suggests that the best way to learn about a new facility in an externally produced package of software is by writing tests. If you want to use a new method, class, or API, first write tests to learn how it works and ensure it works as you expect.

TDL expands significantly on the Explanation and Learning Test ideas both in its approach and its audience. Novice programmers will be presented with unit tests as examples to demonstrate how programming concepts are implemented. Further, programmers will be taught to utilize automated unit tests to explore new concepts.

While the idea of using automated tests as a primary teaching mechanism is believed to be a new idea, the approach of requiring students to write tests in lab and project exercises has a number of predecessors. Barriocanal [3] documented an experiment in which students were asked to develop automated unit tests in programming assignments. Christensen [5] proposes that software testing should be incorporated into all programming assignments in a course, but reports only on experiences in an upper-level course. Patterson [13] presents mechanisms incorporated into the BlueJ [11] environment to support automated unit testing in introductory programming courses.

Edwards [6] has suggested an approach to motivate students to apply TDD that incorporates testing into project grades, and he provides an example of an automated grading system that provides useful feedback. TDL pushes automated testing even earlier, to the very beginning in fact.

4. TDL IN INTRODUCTORY COURSES

Test-driven learning can be applied from the very first day of the very first programming course. Textbooks often begin with a typical "Hello, World!" example or the declaration of a variable, some computation and an output statement. The following is a possible first program in C++:

```
#include <iostream>
using namespace std;

int main()
{
  int age;
  cout << "What is your age in years?" << endl;
  cin  >> age;
  cout << "You are at least "
       << age * 12
       << " months old!" << endl;
}
```

This approach requires the immediate explanation of the language's input/output facilities. While this is a reasonable first step, a TDL approach to the same first program might be the following:

```
#include <cassert>

int main()
{
  int age = 18;
  int ageInMonths;
  ageInMonths = age * 12;
  assert(ageInMonths == 216);
}
```

Notice how use of the `assert()` macro from the standard C library is used, rather than a full-featured testing framework. Many languages contain a standard mechanism for executing assertions. Assertions require very little explanation and provide all the semantics needed for implementing simple tests. The assert approach minimizes the barriers to introducing unit testing, although it does bring some disadvantages. For instance, if there are multiple assert statements and one fails, no further tests are executed. Also, there is no support for independent tests or test suites. However, because the programs at this level are so small, the simplicity of assert statements seems to be a reasonable choice.

As a later example, a student learning to write *for* loops in C++ might be presented with the following program:

```
#include <iostream>
#include <cassert>
using namespace std;

int sum(int min, int max);

int main()
{
  assert(sum(3,7)==25);
  cout << "No errors encountered" << endl;
}

// This function sums the integers
//   from min to max inclusive.
// Pre: min < max
// Post: return-value = min + (min+1) + ...
//   + (max-1) + max
int sum(int min, int max)
{
  int sum = 0;
  for(int i=min;i<=max;i++)
  {
    sum += i;
  }
  return sum;
}
```

In a lab setting, the student might then be asked to write additional unit tests to understand the concept. For instance, they might add the following assert statements:

```
assert(sum(-2,2) == 0);
assert(sum(-4,-2) == -9);
```

Later they might be asked to write unit tests for a new, unwritten function. In doing so, they will have to design the function signature and perhaps implement a function stub. This makes them think about what they are going to do before they actually do it.

Once the programmer ventures beyond the lab into larger projects, tests can be separated into a `run_tests()` function and tests can be partially isolated from each other by placing them in independent scopes as in the following example:

```
#include <cassert>

class Exams
{
  public:
    Exams();
    int getMin();
    void addExam(int);
  private:
    int scores[50];
    int numScores;
};

void run_tests();

int main()
{
  run_tests();
}

void run_tests()
{
  { //test 1 Minimum of empty list is 0
    Exams exam1;
    assert(exam1.getMin() == 0);
  } //test 1

  { //test 2
    Exams exam1;
    exam1.addExam(90);
    assert(exam1.getMin() == 90);
  } //test 2
}
```

TDL should not compete with other approaches in introductory courses. Rather TDL should complement and integrate well with various programming-first [1] approaches such as imperative-first, objects-first, functional-first, and event-driven programming among others.

5. TDL IN LATER COURSES

TDL is applicable at all levels of learning. Advanced students and even professional programmers in training courses can benefit from the use of tests in explanations.

As students gain maturity, they will need more sophisticated testing frameworks. Fortunately a wonderful set of testing frameworks that go by the name xUnit have emerged following the lead of JUnit [8]. The frameworks generally support independent execution of tests (i.e. execution or failure of one test has no effect on other tests), test fixtures (common test set up and tear down), and mechanisms to organize large numbers of tests into test suites.

The final example below demonstrates the use of TDL when exploring Java's `DefaultMutableTreeNode` class. Such an example might surface when first introducing tree structures in a data structures course, or perhaps when a programmer is learning to construct trees for use with Java's `JTree` class. Notice the use of the `breadthFirstEnumeration()` method and how the assert statements demonstrate not just the interface to an enumeration, but also the behavior of a breadth first search. A complementary test could be written to explore and explain depth first searches. In addition, notice that this example utilizes the JUnit framework.

```
import javax.swing.tree.DefaultMutableTreeNode;

import junit.framework.TestCase;

public class TreeExploreTest extends TestCase {
 public void testNodeCreation() {
   DefaultMutableTreeNode node1 =
        new DefaultMutableTreeNode("Node1");
   DefaultMutableTreeNode node2 =
        new DefaultMutableTreeNode("Node2");
   DefaultMutableTreeNode node3 =
        new DefaultMutableTreeNode("Node3");
   DefaultMutableTreeNode node4 =
        new DefaultMutableTreeNode("Node4");
   node1.add(node2);
   node2.add(node3);
   node1.add(node4);
   Enumeration e = node1.breadthFirstEnumeration();
   assertEquals(e.nextElement(),node1);
   assertEquals(e.nextElement(),node2);
   assertEquals(e.nextElement(),node4);
   assertEquals(e.nextElement(),node3);
 }
}
```

6. ASSESSMENT AND PERCEPTIONS

A short experiment was conducted in two CS1 sections at the University of Kansas in Spring 2005. The two sections were taught by the same instructor using a popular C++ textbook. The experiment was conducted in three fifty-minute lectures and one fifty-minute lab that covered the introduction of classes and arrays. While both sections had been introduced previously to the `assert()` macro, during this experiment the first section was instructed using TDL and the second section was presented examples in a traditional manner using standard output with the instructor explaining the expected results.

At the end of the experiment, all students were given the same short quiz. The quiz covered concepts and syntax from the experiment topics. In order to make the two sections homogeneous, two outliers (36 and 48 out of 100 on the first exam prior to the TDL experiment) were removed from the sample, leaving all students with first exam scores above 73. The results given in Table 1 indicate that the TDL students scored about ten percent higher on the quiz than the non-TDL students. While a larger study is needed before drawing any conclusions, the results indicate that TDL can be integrated without negative consequences and support further investigation into potential benefits.

To gauge programmer perceptions of Test-First and Test-Last programming, a survey was conducted at the beginning of a range of courses at the University of Kansas including CS2, an undergraduate software engineering course, and a

	Students	**Exam 1** 100 total	**Quiz 1** 10 total
TDL	13	86.15	7.84
Non-TDL	14	86.71	7.14

Table 1: TDL vs. Non-TDL Mean Scores

graduate software engineering course. Additionally, the survey was conducted at the end of a four-day training course for professional software developers in a large corporation after exposure to TDL. Students were briefly introduced to the differences between Test-First and Test-Last programming, then asked their opinions of the two approaches and asked which approach they would use given the choice. Results are summarized by course in Table 2 and by years of programming experience in Table 3. The Test-First (TF) and Test-Last (TL) opinions were recorded on a five-point scale with 0 being the most negative and 4 the most positive.

As the data shows, while the groups all had similar opinions of the Test-First and Test-Last approaches, the more experienced programmers were much less likely to choose a Test-First approach. Comments recorded on the surveys indicated that the predominant reason was a tendency to stick with what you know (Test-Last). Perhaps it is no surprise that younger students are more open to trying new ideas, but this points to the fact that early introduction of good ideas and practices may minimize resistance.

Course	**No. of Students**	**Avg. TF Opinion**	**Avg. TL Opinion**	**Choose TF**
CS2	28	2.71	2.75	54%
SE	10	2.63	3.70	50%
SE(grad)	12	2.91	2.83	67%
Industry	14	2.85	3.14	29%

Table 2: TDD Survey Responses by Course

Exp. (Yrs)	**No. of Students**	**Avg. TF Opinion**	**Avg. TL Opinion**	**Choose TF**
<=10	55	2.75	3.00	55%
>10	10	2.75	3.00	22%

Table 3: TDD Survey Responses by Experience

7. CONCLUSIONS

This paper has proposed a novel method of teaching computer programming by example using automated unit tests. Examples of using this approach in a range of courses have been provided, and the approach has been initially assessed. Connections between this approach and test-driven development were also explored.

This research has shown that less experienced students are more open to adopting a Test-First approach, and that students who were taught for a short time with the TDL approach had slightly better comprehension with no additional cost in terms of instruction time or student effort. In addition, the benefits of modeling testing techniques and introducing automated unit testing frameworks have been noted.

Additional empirical research and experience is needed to confirm the positive benefits of TDL without negative side-effects, but the approach appears to have merit. It seems reasonable that textbooks, lab books, and on-line references could be developed with the TDL approach. Some materials are already available at http://www.simexusa.com/tdl/.

8. REFERENCES

[1] Computing curricula 2001. *Journal on Educational Resources in Computing*, 1(3es):1, 2001.

[2] S. K. Andrianoff and D. B. Levine. Role playing in an object-oriented world. In *SIGCSE '02: Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pages 121–125. ACM Press, 2002.

[3] E. Barriocanal, M. Urb'an, I. Cuevas, and P. P'erez. An experience in integrating automated unit testing practices in an introductory programming course. *ACM SIGCSE Bulletin*, 34(4):125–128, December 2002.

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

[5] H. B. Christensen. Systematic testing should not be a topic in the computer science curriculum! In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, pages 7–10. ACM Press, 2003.

[6] S. Edwards. Rethinking computer science education from a test-first perspective. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications: Educators' Symposium*, pages 148–155, 2003.

[7] H. Erdogmus. On the effectiveness of test-first approach to programming. *IEEE Transactions on Software Engineering*, 31(1):1–12, January 2005.

[8] E. Gamma and K. Beck. http://www.junit.org.

[9] H. Heinecke and C. Noack. *Integrating Extreme Programming and Contracts.* Addison-Wesley Professional, 2002.

[10] D. Janzen and H. Saiedian. Test-driven development: concepts, taxonomy and future directions. *IEEE Computer*, 38(9):43–50, Sept 2005.

[11] M. Kölling and J. Rosenberg. Guidelines for teaching object orientation with java. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, pages 33–36. ACM Press, 2001.

[12] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[13] A. Patterson, M. Kölling, and J. Rosenberg. Introducing unit testing with BlueJ. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, pages 11–15. ACM Press, 2003.