Test-Driven Design for Introductory OO Programming

Viera K. Proulx
College of Computer Science
Northeastern University
Boston, MA
vkp@ccs.neu.edu

ABSTRACT

Test-Driven Design (TDD) has been shown to increase the productivity of programming teams and improve the quality of the code they produce. However, most of the introductory curricula provide no introduction to test design, no support for defining the tests, and do not insist on a comprehensive test coverage that is the driving force of the TDD.

This paper presents a curriculum, pedagogy, and the software support for introductory object-oriented program design that uses the TDD consistently from the very beginning. The testing software does not increase the program complexity and is designed to work with the simplest programs. It has been used by hundreds of students at several colleges and is freely available on the web.

Our experiences show that besides improving the quality of code students produce, TDD combined with the novice-appropriate test libraries reinforces students' understanding of the object oriented program design.

Categories and Subject Descriptors

K.3.2[**Computer and Information Science Education**]; D.1.5 [Programming Techniques]: Object-oriented Programming]; D3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects*

General Terms

Design, Reliability, Languages

Keywords

CS1/2, Design, Pedagogy, Programming Education, Test Driven Design

1. INTRODUCTION

Test Driven Design: Industry vs Teaching

Nearly everyone agrees that program design includes the testing of the program to verify that it performs (to the extend that it can be tested) as expected. The professional programming community has realized that designing tests after the program has been written often leads to tailoring the tests to the code, rather than to the original problem statement. The Test Driven Design (TDD) program development technique has evolved to address this problem. It has been the key process used by the Extreme Programming teams and has been shown to improve the quality of the code teams produce. Jeff Langr [12] comments on the improvements in the code organization as the result of enforcing the *test first design (TfD)* strategy: *"One of the practices in XP is test-first design (TfD). Adopting TfD means that you write unit-level test for every piece of functionality that could possibly break. It also means that these tests are written prior to the code. ... The first (hopefully obvious) effect of TfD, is that the code ends up being testable... Third, the granularity of code chunks written by a developer via TfD is much smaller. This occurs because the easiest way to write a unit test is to concentrate on a small piece of functionality. By definition, the number of unit tests thus increases - having smaller chunks, each with its own unit test, implies more overall code chunks and thus more overall unit tests."*

One would think educators would jump at the opportunity to improve students' programming skills by incorporating TDD (or TfD) into the introductory curriculum. However, the problem in doing so lies in the tools one has available for defining the test cases and evaluating the test results. Some authors and curriculum designers resort to JUnit, a professional level testing framework. But, this introduces an extra burden on the novice programmer. The student not only has to learn how to use this new framework, but also has to understand program design concepts that are much more advanced than the novice's programming expertise.

A survey of over 20 current introductory textbooks that use Java programming language found only two in which testing is a required activity from the beginning (see Appendix A). All other texts in this survey relegate testing to a brief narrative of no more than a few pages with no instructions for the student on how to design and use tests. The two texts that use testing throughout do not provide examples of tests, nor do they provide a support for running the tests — or for reporting the test results [1].

A survey of recent papers on computer science education shows a number of attempts to include testing in introductory object-oriented programming instruction [4, 5, 8, 9, 10, 11, 13, 15, 16, 17, 18]. Most of them rely on JUnit, a testing

[1]BlueJ software supports unit testing in the JUnit style, but is not discussed in the textbook [1].

environment designed for a professional programmer. While in many cases students do write more test cases and follow the more stringent design protocol, the writers do not see students acquiring a habit of writing tests. All of them comment on the added complexity of the overall code — typically there is one test class for each class a student defines, making the final product even more complex. Some attempts in simplifying this process for students follow the same path — the BlueJ Unit Test support increases the difficulty, as students clutter the screen with yet more test classes.

Our experience has led us to believe that associating test cases with specific classes (as is often done when using JUnit) detracts from good object-oriented design. For example, the tests for methods defined for classes that recursively implement the binary search tree data type do not belong to the *Leaf* class or to the *Node* class, but should in one place cover both variants of the *BinarySearchTree* union type. [2]

Consequently, our students define all examples of data and the test cases that may use these instances in a separate `Examples` class, that represents the client for the student's code. Only two papers from the ones listed above [13, 16] defined the test cases in a special *Test* class that emulated the client to the program — our chosen approach. But even in those cases the authors failed to provide a novice-appropriate test harness for defining the tests, evaluating the tests, and reporting the results. The approach that comes the closest to ours is presented by Thornton et.al. in [17]. However the focus of their tool is the testing of GUI programs, not programs from the general domain.

Test Design: A Pedagogical Imperative

We argue that the students need to develop their testing skills hand-in-hand with their programming skills, supported at each stage by a test harness that allows them to express their tests at their current level of mastery of the programming language. This approach not only teaches students to practice good programming style, but it also reinforces their understanding of the programming concepts they are learning.

Let us illustrate this on an example. Suppose the first class the student sees is the class `Item` that contains two fields, `String name` and `int price`, and the first method we define is the method `Item reduce(int d)` that produces a new item with the price reduced by the given amount. Student's test of this method should focus on the expected result: a new `Item` with the same name where `price` equals the original `price` minus `d`. (We ignore in this example the discount value which would lead to a negative price.) The appropriate test would be:

```
Item bread = new Item("Bread", 200); // price in cents
checkExpect(bread.reduce(20), new Item("Bread", 180));
```

This is the first object and the first method invocation the student sees. The syntax overload and the complexity of getting even the simplest program to run is great. This is no time to learn the intricacies of equality, overriding the `equals` method, using yet another environment for defining tests — especially if overriding the `equals` method requires that the student understands the distinction between the two kinds of equality. (And, of course, the student does not know anything about overriding methods.) Instead, the `checkExpect` test case compares two object by their values, which is what the student would expect.

Furthermore, instead of being a burden, writing the test case in this simple form teaches the student how to define an instance of the class `Bread` and how to invoke the method `reduce` in this class, as the code would appear in the client class.

As students master more advanced concepts the discussion of the equality comparison can be presented in the context of well understood concepts. Students can then start gradually defining their own equality tests, so that by the end of the semester they can use the professional environment of JUnit with confidence, having developed a deep understanding of the problems they may encounter.

2. TEST DESIGN FOR NOVICES

Test design in the functional style

Our curriculum starts by enforcing the use of *Value Objects*, i.e. making sure that every method produces a new instance with the values modified as desired. Beck in Effective Java [2] writes: *"We can use objects as values... One of the constraints on Value Objects is that the values of the instance variables of the object never change once they have been set in the constructor. ... When you have Value Objects, you needn't worry about aliasing. ... One implication of Value Objects is that all operations must return a new object ... Another implication is that Value Objects should implement* `equals`*, because ...".* The benefit of using *Value Objects* approach is that the test for every method only needs to compare the value of the object it produced against the value of the expected object. [3]

To enforce the use of unit tests with every method a student designs, students are taught to design methods by following the DESIGN RECIPE that consists of six steps:

Step 1: Problem analysis and data definition; **Step 2:** Purpose statement and the method header; **Step 3:** Examples with expected outcomes; **Step 4:** Inventory of available data elements and methods that can be invoked on them; **Step 5:** Design of the method body; **Step 6:** Converting examples into test cases and running the tests.

[2]Robert V. Binder [3] writes:
The work of testing must be orchestrated to complement the technical constraints and opportunities of the object-oriented paradigm and the structure of the system under test.
1. Although a class is the smallest natural unit for testing, class clusters are the practical unit for testing. Testing a class in total isolation is typically impractical. Methods are meaningless apart from their class, but must be used individually to exercise class responsibilities. Method testing must consider the class as a whole. As a result, the test design must be method-specific but based on cluster-scope responsibilities. ... At this scope, testing must combine aspects of unit and integration testing.
6. Class testing must be closely tied to class programming. Object-oriented unit testing proceeds in a shorter cycle than the corresponding activities in procedural development. The development process must facilitate short code/test cycles. ...

[3]The Java `equals` method does not by default compare objects by the values they represent (extensional equality). The programmer must override the `equals` method to implement the desired measure of equality. Many other languages (especially functional languages such as Scheme) provide two different equality comparison functions.

The grading of the problems reinforces the emphasis on the test design by assigning equal weight to the design of the method, the design of its header and purpose (contract and documentation), and the design of the tests.

The programs students write consist of interface definitions, class definitions, and a class `Examples` that contains sample data definitions for each class defined earlier as well as test cases for every method defined in any of the classes defined earlier. At the beginning of the semester students use a special programming environment designed for a novice programmer (*ProfessorJ* within the *DrScheme* [6, 7, 14]). The test cases are grouped into boolean expressions that have the following format:

```
Item bd = new Item("Bread", 200);
boolean testReduce =
 (check bd.reduce(20) expect new Item("Bread", 180)) &&
 (check bd.reduce(40)  expect new Item("Bread", 160));
```

Every `check - expect` expression produces a boolean value, and a test report is generated for all boolean expressions with identifiers that start with `test`. The code within the `Examples` class uses the same syntax and behaves the same way as all other class definitions - with the single exception that properly formulated test cases are evaluated when the program runs. The test report not only lists the tests that failed, but also displays both the actual and the expected values in a humanly-readable form (as one would get from a well-formed `toString` method), and provides for each failed test a link to the test source code.

Transition to the real world

We have used this ideal setting for a couple of years. Students learned early to practice test-driven design and the supporting testing library just reinforced the key course concepts — with no syntactic or logistic overhead. However, after a couple of weeks using the learning environment, our course moves on to standard Java (using the Eclipse IDE). Without the support of a testing library the task of designing and evaluating tests became a nightmare.

During the first year we taught this curriculum we required that students override the `toString` method for every class and compare visually the actual and the expected values. Besides the extra overhead, this technique did not scale to large data sets or programs with a number of methods to test. Comparing visually two binary search trees with six nodes, seven leaves, and where the data in each node consists of three fields is an impossible task, especially if one has to evaluate several such tests.

In the following years we required that students implement their own equality comparison by defining the interface

```
interface ISame<T>{
  boolean same(T that); }
```

and requiring that every class students design implements this interface. The test cases then used this method to compare the two instances. Here the task of implementing the `same` method for more complex class hierarchies, such as binary search trees, exceeded the difficulty of the original programming assignment and forced us to introduce prematurely the concepts that the students were not ready to handle.

To automate the test evaluation and reporting we designed a testing library that automatically evaluated all test cases and reported the results (using the user-defined `same` methods). But students' frustration with designing the `same` methods remained and the test coverage of their programs after the transition to standard Java remained marginal.

Testing library for novices

In 2007 we designed and implemented a new testing library that automates the comparison of the values of arbitrary instances of all classes (whether user-defined or a part of the Java Standard Libraries), produces the test results with display of the actual and expected values for each failed test with a link to the source for the test.

We have used a very crude prototype of the library in the Fall of 2007 and a reasonably stable version in the Spring of 2008. The difference in the student's experience has been noticeable throughout the semester. Undergraduate peer tutors who have taken the course in the previous years commented on how much easier it is to enforce the design and use of testing — and wished they would have had access to our *tester library* when they took the course. We also had fewer students who failed the course (2 out of 80 in the Spring 2008, compared to 8—10 out of 80 in the previous years).

But the most convincing evidence of success emerged with the final projects. Students worked on similar final projects at the end of both the Spring 2007 and the Spring 2008. The goal of the project was to let the students implement their own design of a reasonably complex program with some user interactions (typically a game), properly documented and tested (or so we hoped).

In the Spring semester 2007 the test coverage of the submitted final projects was dismal. In many cases there was no test suite, in most cases the test cases tested only the top level methods regardless of whether the students used JUnit or our version of testing library that relied on the user-defined `same` methods.

In the Spring 2008 there were only a handful of projects from among 40 that did not include a substantial test suite. There were several projects designed by average students who followed the design recipe faithfully, tested every method before designing the body, and ended up with a beautiful working project with readable code and documentation. Sadly, several of the projects with almost no tests (and which barely worked) were written by students who had prior Java experience and who were trying to create a project with fancy user interactions, getting ensnared by their own trap of poor design habits.

3. TESTER DESIGN AND USE

Understanding equality

We illustrate why the design of equality comparison is much harder than the design of programs when dealing with complex data. Suppose we want to verify that the binary search tree after several insertions has the desired shape.

To insert into a binary search tree all we need to do is:

```
// in the Leaf class that implements or extends BST:
BST insert(Data d){
  return new Node(d, this, this); }

// in the Node class that implements or extends BST:
BST insert (Data d){
  if d.lessThan(this.data) return
```

```
    new Node(this.d, this.left.insert(d), this.right);
  else return
    new Node(this.d, this.left, this.right.insert(d));}
```

However, to determine whether we constructed a correctly shaped tree we need to implement the method
`boolean same(BST that)`
The method header in defined in the interface (or abstract class) `BST` and the method has to be implemented in both variants. However, we see that in the class `Node` we first need to check whether `that` is an instance of the `Node` class. For this we either need to use the Java operator `instanceof` (not really in object-oriented style); or we need to add two new methods: `boolean sameLeaf(Leaf that)` and `boolean sameNode(Node that)` and use double dispatch to complete the implementation; or we need to use the *Visitor pattern*. Every alternative is much harder than the original problem and adds a new layer of complexity to the problem. The design of equality tests provides a rich context for discussing object-oriented program design, but should be introduced only when the student is ready to handle the concepts needed to solve the problem.

Tester Design

The testing library (*tester*) leverages the Java reflection mechanism to analyze the program source. The user specifies the name of the class that contains the data definitions and the test cases (typically the `Examples` class). Each test case is specified as an invocation of a method in the `Tester` class and several test cases are typically grouped together in a method with a name that starts with `test`. For example, the following method may define tests for the method `reduce` mentioned in the earlier section:

```
Item bread = new Item("Bread", 100);
Item milk = new Item("Milk", 200);

void testReduce(Tester t){
 t.checkExpect(bread.reduce(20), new Item("Bread", 80));
 t.checkExpect(milk.reduce(30), new Item("Milk", 170)); }
```

The *tester* produces a report that consists of the display of the values (class name followed by a list of fields and their values) for all fields initialized in the `Examples` class, the number of tests performed, the number of failed tests, followed by a list of all failed tests. For each failed test the `tester` displays the actual and the expected values, and a link to the location of the test in the source code.

```
// a class to represent a book
class Book{
  String title;
  int price;  // in cents

  Book(String title, int price){
    this.title = title;
    this.price = price; }

  // is this book cheaper that the given amount?
  boolean lessThan(int amt){
    return this.price < amt; }
}

import tester.*;
class Examples{
  Book hamlet = new Book("Hamlet", 2000);
  Book abc = new Book("ABC", 1000);

  // test the method lessThan in the class Book
  void testLessThan(Tester t){
    t.checkExpect(hamlet.lessThan(1500), false);
    t.checkExpect(abc.lessThan(1500), true);
    t.checkExpect(abc.cheaperThan(1500), false,
        "This test will fail");
  }
}
```

```
Examples:
---------------

   new Examples(
    this.hamlet =
     new Book:1(
      this.title = "Hamlet"
      this.price = 2000)
    this.abc =
     new Book:2(
      this.title = "ABC"
      this.price = 1000))
---------------
Found 1 test method

Ran 3 tests.
1 test failed.

Test results:
--------------
Error in the test number 3
This test will fail
tester.ErrorReport: Error trace:
  at Examples.testLessThan(Examples.java:10)

actual:      true
expected:    false
--- END OF TEST RESULTS ---
```

Special tests:

- For **primitive types and wrapper classes** all values are converted to the corresponding primitive types and compared using Java `==` comparison.
- Two `double` or `float` values are compared for equality within a relative difference initialized to a default value before running the tests. It can be set by the programmer at any time within the test suite.
- The comparison of **circularly-referential** data (for example, a class `Book` with the field of the type `Author` while the class `Author` contains a field of the type `Book`) is handled correctly.
- **Collections - Iterable:** Comparison of two instances of a data collection that represents a sequential data structure (e.g. `Iterable`) traverses the structure and matches the corresponding pairs of data.
- **Collections - Map:** Comparison of two instances of a data collection that implements the `Map` interface matches the two collections for their key-value pairs.
- The `checkOneOf` method tests whether the actual value matches one of finite possible **random** values.
- The `checkRange` method tests whether the actual `Comparable` value falls within the range of the two given values.
- A `checkExpect` method that tests whether the invocation of a method with the given name and arguments by the given object throws the desired **exception** with the expected message.

Imperative Tests

We mentioned that we use the *tester* initially in testing the programs that handle *Value Objects*. While we believe greatly in the benefit of this approach, the *tester* can be used equally well for testing imperative programs.

For example, to test the method
`void reduce(int amount))`
in the class `Item` the test method would be:

```
void testReduce(Tester t){
  // setup:
  Item milk = (new Item("Milk", 200));
  milk.reduce(30);
  // test:
  t.checkExpect(milk, new Item("Milk", 170)); }
```

Alternatively, if we teach students to design methods that always return the object that invoked the method (`this` — a preferred Java style), the method header would become: `Item reduce(int amount));` and the test method would be the same as we have seen in the functional style.

User defined equality

To support students' gradual transition to self-defined equality comparison the *tester* compares two objects in a class that implements the `ISame` interface using its `same` method. This way the students can implement the equality comparison for only some of the classes they define — learning gradually to do all the work on their own and transition smoothly to using JUnit.

4. FINAL REMARKS

Enforcing the TDD for several years, even before we had the tools to fully support this approach, has had a significant impact on our students. The instructors of upper level courses uniformly comment on students' better preparation, a measurably better performance on the course pretests (1 or 2 failures compared with 30% failure prior to introducing the curriculum). The co-op coordinators that work with our students report and conduct extensive yearly evaluations of the co-op experiences based on surveys of both the students and the employers. The surveys show that the quality of the co-op jobs our students get has increased significantly since we introduced the TDD curriculum. They employers report that the programming skills of our undergraduate students surpass those of our Masters Degree students seeking co-op employment.

The course instructors observed that after comparing instances for value equality for several weeks, students understand better the Java reference model, the effects of changing the value of a field vs. changing the structure of a complex piece of data and they have a much better understanding of the different measures of equality one can define.

The http://ccs.neu.edu/javalib website contains the *tester* library tutorial, sources, documentation and download jar files.

The design of the library is based on the testing support for *ProfessorJ*. The author wishes to thank her colleagues on the TeachScheme/ReachJava team, especially Matthias Felleisen for his support and Kathy Gray for her work on *ProfessorJ*.

5. REFERENCES

[1] D. J. Barnes and M. Koelling. *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall, 2003.

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

[3] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, 2000.

[4] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA*, pages 148–155, Oct, 2003.

[5] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE Bulletin*, 36(1):26–30, 2004.

[6] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 369–388, Southampton, UK, September 1997. Springer.

[7] K. E. Gray and M. Flatt. ProfessorJ: a gradual introduction to Java through language levels. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA*, pages 170–177, Oct, 2003.

[8] D. Gries. A principled approach to teaching OO first. *SIGCSE Bulletin*, 40(1), 2008.

[9] B. Hanks, T. Reichlmayr, C. Wellington, and C. Coupal. Integrating agility in the CS curriculum: Practices through Values. *SIGCSE Bulletin*, 40(1), 2008.

[10] D. S. Janzen and H. Saiedian. Test-driven learning in early programming courses. *SIGCSE Bulletin*, 40(1), 2008.

[11] M. Koelling. Unit testing in BlueJ. http://www.bluej.org/tutorial/testing-tutorial.pdf.

[12] J. Langr. Evolution of test and code via test-first design. www.objectmentor.com/resources/articles/tfd.pdfl.

[13] R. Pecinovský, J. Pavlíčkova, and L. Pavliček. Let's modify the objects-first approach into design-patterns-first. *SIGCSE Bulletin*, 40(1), 2008.

[14] V. K. Proulx and K. E. Gray. An introduction to OO program design. *SIGCSE Bulletin*, 38(1), 2006.

[15] V. K. Proulx and R. Rasala. Java IO and testing made simple. *SIGCSE Bulletin*, 36(1), 2004.

[16] J. Spacco and W. Pugh. Helping students appreciate test-driven development (TDD). In *Companion of the 21st annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Portland OR*, pages 907–913, Oct, 2006.

[17] M. Thornton, S. H. Edwards, R. P. Tan, and M. Peréz-Quinones. Supporting student-written tests of GUI porgrams. *SIGCSE Bulletin*, 40(1), 2008.

[18] D. West, P. Rostal, and R. P. Gabriel. Apprenticeship agility in academia. In *Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, San Diego, CA*, pages 371–373, Oct, 2005.

APPENDIX

A. TEXTBOOKS REVIEWED

D. A. Bailey. *Java Structures*; D. A. Bailey and D. W. Bailey. *Java Elements: Principles of Programming*; D. Baldwin and G. W. Scragg. *Algorithms and Data Structures: The Science of Computing*; D. J. Barnes and M. Koelling. *Objects First with Java: A Practical Introduction Using BlueJ*; J. Cohoon and J. Davidson. *Java 1.5 Program Design*; N. Dale, D. Joyce, and C. Weems. *Object-Oriented Data Structures Using Java*; P. J. Deitel and H. M. Deitel. *Java How to Program*; J. Farrell. *Java Programming*; W. H. Ford and W. R. Topp. *Data Structures with Java*; C. Horstman. Java Concepts; E. B. Koff man and U. Wolz. *Problem Solving with Java*; K. Lambert and M. Osborne. *Java A Framework for Program Design and Data Structures*; J. Lewis and W. Loftus. *Java Software Solutions: Foundations of Program Design*; Y. D. Liang. *Introduction to Java Programming*; M. Main. *Data Structures and Other Objects Using Java*; D. S. Malik. *Java Programming From Problem Analysis to Program Design*; D. D. Reily. *The Object of Data Abstraction and Structures Using Java*; D. D. Riley. *The Object of Java: Introduction to Programming Using Software Engineering Principles*; K. E. Sanders and A. van Dam. *Object-Oriented Programming in Java A Graphical Approach*; W. Savich. *An Introduction to Computer Science and Programming*; P. T. Tymann and G. M. Schneider. *Modern Software Development Using Java*; C. T. Wu. *A Comprehensive Introduction to Object-Oriented Programming with Java*;