# Unit Test Support for Java via Reflection and Annotations

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

Weston Jossey
College of Computer and Information Science
Northeastern University
Boston, MA
wjossey@ccs.neu.edu

## ABSTRACT

Novice Java programmers face great difficulties when learning to design unit tests for any nontrivial cases. Deciding whether the result of a method, or the effect the method produced represents the expected result one must understand the difference between equality based on the values an object represents versus the reference equality (identity) — and be able to define the correct equals method.

We describe the *tester* library that supports test design, evaluation, and reporting of the results in the manner that supports a novice programmer. The library uses Java reflection and annotation to compare any two data items (primitive types or objects) by the value they represent, produces report where the expected and actual values are pretty-printed, and a failed test report includes a link to the failed test.

The library has been used in classrooms and is used daily in our 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, Verification

## Keywords

CS1/2, Design, Testing

## 1. MOTIVATION

Most of the introductory textbooks that focus on program design in Java (with or without emphasis on data structures) contain a section on software testing. But unlike other topics, in all but two of the surveyed textbooks [2, 3, 4, 5, 6, 7, 8, 11, 12, 16, 21, 22, 23, 24, 25, 26, 31, 32, 33, 34, 35, 38, 41] testing is only described in general terms with no real explanation how the students should design and evaluate the test cases or report the test results. The two texts that ask the students to practice testing from the beginning do not provide examples of tests, nor do they provide a support for running the tests — or for reporting the test results. [1]

Of course, knowing what it takes to design a test using the JUnit [19], the industry de-facto standard, this is no surprise. In order to use JUnit, the programmer must have a good understanding of how to design methods that compare two objects for equality exactly in the sense that the programmer desires. It also involves extending the `Test` class, and so the programmer should know something about classes and subclasses. If the programmer wants to get meaningful error messages, she must also implement the `toString` method. And, finally, Java requires that when the `equals` method is overridden the `hashCode` method must also be overridden in a manner consistent with the behavior of the `equals` method. All of these concepts are far beyond what a beginner Java programmer can handle. The fact that implementing the `equals` method for more complex class hierarchies is a non-trivial and error-prone process itself exacerbates the situation.

### 1.1 Related work

Over the past several years a number of papers reported on testing in the introductory object-oriented programming instruction [9, 10, 14, 15, 17, 20, 27, 28, 29, 30, 36, 37, 39, 40]. A majority of papers investigates students' experiences with designing tests using JUnit, a testing environment designed for a professional programmer. Overall, the papers report some improvement in the test coverage, but no serious effect on the code quality or on acquiring a habit of writing tests. Most papers comment on the negative impact of the increased complexity of the programming tasks and the resulting code. Even the BlueJ Unit Test support suffers from increased complexity of the overall programming task. Furthermore, when the test cases must refer to already defined classes and methods, it is difficult to enforce *test-first* design practice.

Two papers [29, 36] define the test cases in a special *Test* class that emulates the client to the program — our chosen approach. But even here, there is no test harness that supports the novice programmer in defining the test cases, evaluating them, or reporting the results. The only paper that presents testing support for a novice programmer is

[1] BlueJ software supports unit testing in the JUnit style, but is not discussed in the textbook [5].


*PPPJ '09,* August 27–28, 2009, Calgary, Alberta, Canada.

presented by Thornton et.al. [37], but they focus only on testing of GUI programs.

A special case is the Java Bat web site [18] that provides practice problems for students to write programs that satisfy the given tests. All problems focus on the design of a single function with a predetermined purpose, practicing just the specific aspect of expression evaluation.

## 1.2 Project history

Seven years ago we launched a radically different curriculum (known as *ReachJava*) that introduces object-oriented program design in a systematic way. The draft of the textbook *How to Design Classes* is nearly complete. The curriculum asks the student to use the *Design Recipe* when designing any method. The six steps of the *Design Recipe* ask the student to analyze the problem and define all the relevant data as the first step; to write down the method header and a short purpose statement that identifies the method arguments and what is it expected to produce in the second step; and to make examples of the method use with the expected outcomes in step three. The fourth step asks the student to analyze all available data and list all accessible fields or any methods these fields can invoke (including `this` object and its fields). The design of the body is the fifth step; and finally, step six, requires that the examples from step three be turned into runnable tests, and that the tests are evaluated. Additional tests are defined as needed.

It is clear, that the *Design Recipe* teaches the student to practice *test-first design*. We start with mutation-free methods — every method produces a new value and has no side-effects. This makes the test design conceptually easier: one only needs to compare the value of the new object (or primitive type data) with the expected value. However, there is no support in Java for this type of equality comparison (extensional equality).

We have already seen the clearly positive impact of enforcing the use of the *Design Recipe* in our introductory course that focuses on functional programming in a Scheme-like languages and were hoping to achieve the same results in Java-like environment. It became clear from the first semester we taught the course that the problem of defining tests in a novice-appropriate manner needs to be resolved before students truly believe in the power of *test-first* design and before we can seriously enforce the use of the *Design Recipe* throughout the semester.

In the early weeks of the course we have used a special environment, *ProfessorJ* within the *DrScheme* IDE. *ProfessorJ* provides a beginner Java-like language that has no visibility modifiers, no assignment statement (just field initialization statements or field assignment within the constructor) and no loop statements. It allows for a class to implement an interface, but does not allow for an interface to extend another interface, or for a class to extend another class. By enforcing this simple functional style of programming students concentrate on designing classes that represent the given information, and learn to design methods that dispatch over the union of classes that implement a common interface (for example, a binary search tree interface with the `Leaf` and `Node` classes within the union). We have worked with Kathy Gray, the implementor of *ProfessorJ* in defining the desired behavior for the testing library. Once the *ProfessorJ* testing support was reasonably stable, we have seen a noticeable improvement in students' ability to define tests and their understanding of the need for systematic testing.

However, the move to the standard Java after four or five weeks of the course quickly erased any gains. Faced with the added burden of defining equality for every class, and especially for unions of classes, together with the desired `toString` method overshadowed anything students were trying to learn. As the course introduced stateful methods with the need for testing the effects, and adding to the tests the *setup* and *tear-down* code, testing was no longer a benefit, but has become a major hurdle.

For the past three years we have worked on designing a testing library for standard Java that allows the programmer to compare the values represented by any two objects, reports the results in a novice-appropriate manner, pretty-prints the values of the expected value and the actual result, taking over the task of defining and evaluating the extensional equality of two pieces of data. The *tester* library has been used by students at several colleges and high school not only in the context of our curriculum but also in programming classes that follow a more traditional route.

In the next section we describe the initial design of the library that relied on Java support for reflection. Third section describes the additional functionality of the *tester* enabled by the use of *annotations*. We then present our experiences with using the *tester* both in the classes and in our own work, follow with a list of challenges and planned extensions, and conclude with acknowledgments.

# 2. REFLECTION

## 2.1 *Tester* library: User's view

A novice programmer needs to see how the program that consists of a collection of class definitions is actually used by the intended client. This includes seeing the code that defines instances of these classes as well as the actual code where methods in the class are invoked by the appropriate instances of this class. This has motivated our decision to define all needed data as well as all tests within the *Examples* class. The *Examples* class represents the client to the code student has designed. Our *Design Recipe for Data Definition* requires that students construct examples of data for every class they design. When the student designs the first method, instances of the class that can be used to invoke the method are already available in the *Examples* class.

The tests are grouped into a method whose name starts with `test` and consumes an instance of the class `Tester`. The method either returns a `boolean` value (to support our mutation-free style at the beginning) or it is `void`.

The test case syntax is straightforward and illustrates the actual method invocation in any client code. So, if the method `discount` reduces the price of the book by the given amount, the data example and the test case will be:

```
Book b = new Book("Orwell", "1984", 20);

// test the method discount in the class Book
boolean testDiscount(Tester t){
  return t.checkExpect(b.discount(3),
                       new Book("Orwell", "1984", 17),
                       "test book discount");
}
```

The `String` argument that explains the purpose of the test is optional. If the above test fails the report will be:

```
Found 1 test method
.....................................................
..............
Ran 1 test.
1 test failed.

Test results:
--------------

Error in test number 1
test book discount
tester.ErrorReport: Error trace:
at Examples.testDiscount(Examples.java:19)
at Examples.main(Examples.java:29)

actual:                      expected:
  Book: Orwell, by 1984        Book: Orwell, by 1984
   new Book:1(                  new Book:1(
    this.title =  "Orwell"       this.title =  "Orwell"
    this.author =  "1984"        this.author =  "1984"
    this.price = 16).............this.price = 17)

--- END OF TEST RESULTS ---
```

It pretty-prints both the actual and the expected values, marks the first place where the two values differ, and includes a link to the failed test case. (Both `Examples.java:19` and `Examples.java:29` are live links).

It is clear, that the design of tests poses no additional burden. Quite the contrary, the student can see clearly how the classes he defines will be used to define the needed data, and how the methods will be invoked.

The library is distributed as a single *jar* file and the class that includes the tests must include the `import tester.*;` statement.

The name of the *Examples* class is not mandatory — it is the default class name when using the simplest options for running the tests. We provide a number of choices for the user for invoking the test evaluation. The simplest is by launching the `main` method in the `tester.Main` class. If the user changed the name of the *Examples* class, the new name can be supplied to the `tester.Main` as a run time argument. There are several static methods within the `Tester` class that also launch the test evaluation and test reporting. These give the user the choices whether or not all data defined in the *Examples* class should be displayed, and whether to report only the failed tests or all test results.

## 2.2 Core features

The following tests are invoked by a `checkExpect` method with the actual and expected values and the (optional) test description `String`.

We include in the test comparison all relevant fields: we exclude `static` fields, and fields declared as `volatile` or `transient`. Before considering the general case the *tester* considers the following special cases:

- Verify the neither of the two objects is `null`. If both values are `null` the test succeeds, otherwise it fails.
- If the two objects are identical instances the test succeeds.
- If the two objects are instances of the *String*, use the `equals` method for `String`s to evaluate the test.
- For data of the primitive type use the `equals` method for the wrapper classes to evaluate the test, unless these are inexact numbers.
- The comparison of two values of the types `double` or `float` requires special handling (described in the next section). Unless the two values are identical, the desired comparison requires a specification of allowable tolerance. Furthermore, the equality within a given tolerance (whether absolute or relative) is not transitive, violating the definition of the equality relation.
- Two instances of the wrapper classes and handled in a similar manner. Comparison between a primitive type and a data in the corresponding wrapper class is permitted.
- Two `Array`s are first compared by their length, then item-by-item.
- Two instances of *Java Collections* classes which implement the `Iterable` interface are traversed and compared item-by-item.
- Two instances of a class that implements the `Map` interface are compared by their `EntrySet`s.
- Finally, two instances of any other class are compared field-by-field, invoking the same comparison strategy for the values of every field.
- At the start of each test we clear a hash map that is used to store links to all data items already compared. It is consulted before each additional comparison to detect circularity in data definitions.

## 2.3 Special handling

Using the *tester* library with our students identified several test scenarios that should be represented by a single test statement. This motivated us to extend the *tester* library to handle these special cases. For each of them we provide motivation and describe the behavior of the test evaluation.

- `checkInexact`

  The comparison of two objects that contain a field with the values of the types `double` or `float` (or their wrapper classes) must invoke the appropriate `Inexact` variant of the test method, and must provide the desired relative `TOLERANCE` to determine the desired accuracy of the comparison.

  Inexact numbers (`double` and `Double` or `float` and `Float`) are considered the same if their values `val1` and `val2` satisfy the formula:

  ```
  (Math.abs(val1 - val2) /
      (Math.max (Math.abs(val1), Math.abs(val2))))
    < TOLERANCE;
  ```

  If a comparison between two objects involves inexact comparison of any two fields, the test case report, whether successful or not, includes a warning that an inexact comparison has been involved in evaluating this test case.

Note: It is important to note that a comparison of two numbers of the types `double` or `float` could succeed through direct comparison. For example when evaluating `new Double(5.0) == 5.0` the comparison succeeds and no warning of inexact comparison is issued.

If the comparison of any two objects involves inexact comparison, and the test method required exact comparison, (the programmer failed to use the `Inexact` variant) the test case fails and the warning is displayed as well.

The objects involved in the *inexact* comparison may contain several *inexact* fields, where the magnitude of the *inexact* values is significantly different. Using the *relative* `TOLERANCE` allows us to define meaningful tests for these scenarios.

- `checkFail`

  At times we may want to include a test that we expect to fail. This variant evaluates the test case in the usual manner, and reports success if the comparison failed.

- `checkIterable`

  While the *Java Collections Framework* classes that implement the `Iterable` interface are compared item-by-item, we do not impose this limitation on user-defined classes of this type. If the programmer defines class hierarchy (or a single class) that represents a binary search tree and implements the `Iterable` interface, he may want to compare the data structurally at one time, and by the sorted collection of data the tree represents at other times.

- `checkTraversal`

  The *tester* library defines a functional iterator interface `Traversal` as follows:

```
// a functional iterator interface
public interface Traversal<T>{

  // is this dataset empty?
  boolean isEmpty();

  // produce the next element in this dataset
  T getFirst();

  // produce a traversal over the rest of this dataset
  Traversal<T> getRest();
}
```

  We find this useful, especially while programming in the functional style at the beginning of our course. The test evaluation traverses over all data in the actual and expected datasets for which the `Traversal` objects have been provided.

- `checkException`

  The `checkException` method tests whether the program throws an `Exception` at the specified time. The arguments to the `checkException` method include:

  - the object that should invoke the method that throws an `Exception`
  - the name of the method that should be invoked
  - A list of arguments to the method that should be invoked, supplied at the end of the argument list as an `Array` or as *ellipsis* arguments
  - an instance of the expected `Exception` that includes the expected message.

  The test not only verifies that the method invocation as specified throws the `Exception` of the desired class, but also compares the actual and expected error messages.

- Testing the desired method invocation

  It is impossible to invoke a `private` method within the *Examples* class. To enable tests of such methods, the programmer can supply the `checkExpect` method with the object that should invoke the method, the method name, the expected result the method produces, and a list of arguments to be used in the method invocation.

  This problem has been eliminated in the newer version of the *tester* library that uses *Annotation*s to aid in selection of the test methods.

- `checkOneOf`

  Our students design simple games where part of the action is the movement of an object by a short random distance (for example in the range from -3 to 3, horizontally). This test allows the student to verify that the actual value is one of the several expected values.

  The expected values can be specified either as an `Array` or as a collection of *ellipsis* arguments at the end of the argument list.

- `checkNoneOf`

  This is just the reverse of the previous scenario and is handled in the same manner.

- `checkRange`

  At times we may want to know that the given value (numeric or not) is within the given range. We may want to check that the daily temperature is in the range from -120 to 150, or that after we filter a list of names every name starts with the letters between `A` and `G`. The `checkRange` method has several variants. It covers all numerical types, as well as instances of any class that implements the `Comparable` interface. Another variant allows the user to provide the `Comparator` object to be used in evaluating the test. User supplies the lower and the upper bound. By default the lower bound is inclusive and the upper bound is exclusive, but the user may override this default. Finally, the method `checkNumRange` allows the user to mix the primitive (or the wrapper) types used to specify numeric ranges. So, one can check whether `7.5` is in the range between `3` and `9`.

## 2.4 Extensibility

### 2.4.1 *User defined* `toString` *method*

At times we would like to ask students to design `toString` method for their class hierarchies. For example, we may want the `toString` method for a binary search tree produce

all data in the tree following the *inorder* traversal. Or when working with classes that deal with expression evaluation, we may want to see the expressions displayed as mathematical formulas.

To make this possible, but still allow for the informed error reporting, the *tester* library checks whether the user has overridden the `toString` method. If that is the case the relevant data is displayed in two ways — first using the user-defined `toString` method, then using the standard pretty-printing process.

### 2.4.2 User defined equality comparison

While it is desirable to provide automatic test evaluation for our students at the beginning, they need to understand what is involved in designing equality comparison at various levels.

The *tester* library provides two ways in which the user can define equality comparison. The user may define the equality comparison for a specific class, leaving the rest of the equality comparison to the *tester* library. The *tester* library defines the `ISame` interface as follows:

```
// an equality comparison interface
public interface ISame<T>{

  // is this item the same as the given one?
  boolean same(T t);
}
```

The test evaluation for any pair of objects checks first whether the class in which the objects were defined implements the `ISame` interface. If that is the case, the appropriate `same` method is used to determine the equality of these two objects only, and the rest of the test evaluation proceeds in the normal manner.

To provide a complete flexibility and extensibility, the *tester* library provides the `checkEquivalence` method, where the user needs to supply a function object that is an instance of the class that implements the `Equivalence` interface.

The `Equivalence` interface is defined as follows:

```
// an equivalence comparison interface
public interface Equivalence<T>{

  // is t1 item the equivalent to t2?
  boolean equivalent(T t1, T t2);
}
```

We have decided not to rely on the Java `equal` method for two reasons. First, there is no need here to worry about implementing the `hashCode` method as well. Second, this technique allows the user to define several different measures of equality between the same two objects.

## 3. ANNOTATIONS

After a student has designed twenty classes named *Examples* it would be nice to have the name mean something with respect to the program whose examples and tests it contains. Also, it may be tedious to use for every test method a name that starts with `test`. Java `Annotation` support allows us to solve this problem.

There are two uses of `Annotation`s in the *tester* library. User may annotate the class definition with the `@Example` annotation. Doing so alerts the tester to search for test methods defined within this class. So, if the current project contains several classes with this annotations, all test methods within all the annotated classes will be performed when the test suite runs. The only restriction is that the annotated class must contain a default constructor. However, there is no requirement that the default constructor be visible outside the class, and so, by declaring it `private` the client APIs remain unchanged. It is perfectly fine if the default constructor does not initialize any fields.

User may also annotate individual methods as test methods by adding the `@TestMethod` annotation.

The advantage of using annotations is manifold. Not only do we gain the flexibility of naming the test classes and test methods, but we now can include tests for `private` methods and tests that access `private` fields within that class.

The following example illustrates the use of `Annotation`s. To run the tests one must invoke the `main` method in the class `tester.Main`, providing no run time arguments.

```
import tester.*;

// a sample class --- it is not called 'Examples'
@Example public class AnnotateExample {

  // private field
  private int n;

  // the required default constructor
  private AnnotateExample(){}

  // publicly available constructor
  public AnnotateExample(int n){
    this.n = n;
  }

  // a sample private method
  // is this number bigger than the given one?
  private boolean biggerThan(int m){
    return n > m;
  }

  // test method: the name does not start with 'test'
  @TestMethod public boolean doIt(Tester t){

    AnnotateExample foo = new AnnotateExample(3);
    AnnotateExample bar = new AnnotateExample(8);

      return
        t.checkExpect(foo.biggerThan(5), false) &&
        t.checkExpect(foo.biggerThan(2), true) &&
        t.checkExpect(bar.biggerThan(5), true) &&
        t.checkExpect(bar.biggerThan(10), true,
            "test will fail");
  }
}
```

The test results are as expected:

```
Tester Results
Found 1 test methods

Ran 4 tests.
1 test failed.

Test results:
--------------

Error in test number 4
test will fail
tester.ErrorReport: Error trace:
```

```
at AnnotateExample.doIt(AnnotateExample.java:34)

actual:              expected:
true ............... false


--- END OF TEST RESULTS ---
AnnotateExample
```

# 4. EXPERIENCES

We have been developing our test-first curriculum for the past seven years. In the Spring 2007 we had a testing framework that evaluated all test cases defined with the *Examples* class, using student-implemented `same` method (i.e. requiring that every class implements the `ISame` interface. The test coverage on the final projects was minimal and mostly very disorganized. In the Spring 2008 we had a prototype of the *tester* library in place. While about half of the final projects still had only a minimal test coverage, in the other half there were several projects with hundreds of test cases. This semester, in the Spring 2009, it was very rare that a project did not have a substantial well-designed test suite.

But, as we well know, that is not the only benefit of the test-first design strategy. Programs where the tests are designed before the method bodies tend to be more granular, the methods tend to be shorter, and the tasks are delegated to other classes and methods more readily.

Our final project reviews confirm this as well. Our final project review involve a 15 minute demonstration and code walk for every one of the 50 pairs of students. Only a few contained convoluted code, the majority of programs was readable, consisted of clearly defined classes and methods and had a substantial test suite.

We still need to work on a better organization of imperative tests. The following example shows how the *tester* library can be used to design imperative tests:

```
// to represent a bank account
public class Acct{
  int acctNo;
  int balance;

  Acct(int acctNo, int balance){
    this.acctNo = acctNo;
    this.balance = balance;
  }

  // EFFECT: update the balance by the deposit amount
  void deposit(int amt){
    this.balance = this.balance + amt;
  }
}

// examples of accounts
Acct acct1 = new Acct(34, 100);
Acct acct2 = new Acct(77, 1000);

// reset the accounts to the original values
void resetAccts(){
  acct1 = new Acct(34, 100);
  acct2 = new Acct(77, 1000);
}

// test the method deposit in the class Acct
void testDeposit(Tester t){
  this.acct1.deposit(20);
  t.checkExpect(this.acct1, new Acct(34, 120));

  this.acct2.deposit(1000);
  t.checkExpect(this.acct2, new Acct(77, 2000));
  resetAccts();  // TEST FAILS IF MISSING

  this.acct1.deposit(20);
  t.checkExpect(this.acct1, new Acct(34, 120));
  this.acct2.deposit(1000);
  t.checkExpect(this.acct2, new Acct(77, 2000));
  resetAccts();
}
```

The important concept that the students need to learn here is that every test case should be designed to run independently of the other tests. Without resetting the account states to the original values the second series of tests would fail, even though the test case code is identical.

# 5. CHALLENGES

This project is a work in progress. With each new addition we also ask new questions and see new problems with unit testing that need to be addressed systematically.

The first is a formal definition of each of the test scenarios we have already implemented as well as any new scenarios we may add in the future. Next is a careful study of tests for imperative (state-changing) methods. Suppose the expected effect of the method is the change of the value of two fields. Most of the test designers today verify in two separate test cases that each of the two values has been set to the desired value. Rarely does anyone bother to make sure that no other unintended changes have happened. Robert Glass cites an example where a C++ program dealing with sets of data passed the test for repeated add — if the item was in the set already, a correct message was displayed. However, the item has been added to the set as well. But this has been discovered only after the program has been running for a while.

Kathy Gray and Alan Mycroft [13] describe a solution based on taking a snapshot of the object value before the test and only updating the value once the test against the old value has been performed. Our experimental advanced version of the *tester* library (*Avanti*) accomplishes this task by implementing deep cloning of the object values and verifying that no unspecified changes have been made. While this is not going to be the ultimate solution, we plan to work further on this problem.

There are several other directions we wish to pursue. We have implemented the first version of a test coverage tool, adopting the cobertura [1] open software tool.

We plan to look carefully at the test scenarios that compare the changes in the contents and the structure of the typical classes and class hierarchies that represent collections of data.

We also plan to further investigate the use of `Annotation`s to provide the user with a way to define how the automated testing library should implement the equality comparison. This direction seems to lead to defining a language extension that supports unit test design — approaching from a different direction the path taken by Gray and Mycroft.

Michael Sperber and his team at the University of Tübingen adopted the *tester* library to run each test case as a JUnit test within their version of the *BlueJ* programming

environment. While we believe that a novice programmer benefits from the simple infrastructure of our library, we plan to add the support for JUnit style of tests in the future version of the library — both to support the transision to the less supportive environment found in the industry, and to gain access to the various tools and add-on features already available within the JUnit community.

# 6. CONCLUSION

## 6.1 Resources

The tester library has been used successfully by hundreds of students at several institutions and is used daily in all of our Java programming tasks. The *http://ccs.neu.edu/javalib* website contains the *tester* library tutorial, sources, documentation and download *jar* files.

## 6.2 Acknowledgments

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, to Kathy Gray for her work on *ProfessorJ* and its testing framework, and to the first adopters of the *tester* library, especially Todd O'Bryan, Marc Smith, and Glynis Hamel for their support and feedback.

# 7. REFERENCES

[1] Cobertura. http://cobertura.sourceforge.net/.

[2] D. A. Bailey. *Java Structures*. McGraw Hill, 2 edition, 2003.

[3] D. A. Bailey and D. W. Bailey. *Java Elements: Principles of Programming*. Mc Graw Hill, 2000.

[4] D. Baldwin and G. W. Scragg. *Algorithms and Data Structures: The Science of Computing*. Charles River Media, 2004.

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

[6] J. Cohoon and J. Davidson. *Java 1.5 Program Design*. Mc Graw Hill, 2004.

[7] N. Dale, D. Joyce, and C. Weems. *Object-Oriented Data Structures Using Java*. Jones and Bartlett, 2 edition, 2006.

[8] P. J. Deitel and H. M. Deitel. *Java How to Program*. Pearson Prentice Hall, 7 edition, 2007.

[9] 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.

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

[11] J. Farrell. *Java Programming*. Thomson Course Technlogy, 2008.

[12] W. H. Ford and W. R. Topp. *Data Structures with Java*. Pearson Prentice Hall, 2005.

[13] K. E. Gray and A. Mycroft. Logical testing: Hoare-style specification meets executable validation. *FASE 2009*, 2009.

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

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

[16] C. Horstman. *Java Concepts*. John Wiley & Sons, 5 edition, 2008.

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

[18] JavaBat. http://www.javabat.coms.

[19] JUnit. http://www.junit.org.

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

[21] E. B. Koffman and U. Wolz. *Problem Solving with Java*. Addison Wesley, 2 edition, 2002.

[22] K. Lambert and M. Osborne. *Java A Framework for Program Design and Data Structures*. Brooks-Cole, 2004.

[23] J. Lewis and W. Loftus. *Java Software Solutions: Foundations of Program Design*. Addison Wesley, 6 edition, 2008.

[24] Y. D. Liang. *Introduction to Java Programming*. Prentice Hall, 3 edition, 2001.

[25] M. Main. *Data Structures and Other Objects Using Java*. Addison Wesley, 1999.

[26] D. S. Malik. *Java Programming From Problem Analysis to Program Design*. Thomson Course Technology, 3 edition, 2008.

[27] G. Melnik and F. Maurer. The practice of specifying requirements using executable acceptance tests in computer science courses. In *Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, San Diego, CA*, pages 365–370, Oct, 2005.

[28] L. Murphy, G. Lewandowski, R. McAuley, B. Simon, L. Thomas, and C. Zander. Debugging: The good, the bad, and the quirky - a quantitative analysis of novice's strategies. *SIGCSE Bulletin*, 40(1), 2008.

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

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

[31] D. D. Reily. *The Object of Data Abstraction and Structures Using Java*. Pearson Education/Addison Wesley, 2003.

[32] D. D. Riley. *The Object of Java: Introduction to Programming Using Software Engineering Principles*. Addison Wesley, 2002.

[33] K. E. Sanders and A. van Dam. *Object-Oriented Programming in Java A Graphical Approach*. Addison Wesley, 2006.

[34] W. Savich. *An Introduction to Computer Science and Programming*. Prentice Hall, 5 edition, 2008.

[35] R. Sedgewick and K. Wayne. *Introduction to Programming in Java: An Interdisciplinary Approach*. Addison Wesley, 5 edition, 2008.

[36] 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.

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

[38] P. T. Tymann and G. M. Schneider. *Modern Software Development Using Java*. Brooks/Cole, 2004.

[39] 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.

[40] M. Wick, D. Stevenson, and P. Wagner. Using testing and JUnit across the curriculum. *SIGCSE Bulletin*, 37(1), 2005.

[41] C. T. Wu. *A Comprehensive Introduction to Object-Oriented Programming with Java*. Mc Graw Hill, 2008.