Unit Test Support for Java
via Reflection and Annotations

Viera Králová Proulx
Weston Jossey
Northeastern University, Boston MA

vkp@ccs.neu.edu
wjossey@ccs.neu.edu

PPPJ 2009, Calgary, Alberta, Canada
Outline:

Anatomy of Unit Testing

Why the tester Library

The Design of Tests and Test Evaluation

Reporting of the Results

Our Experiences

Future Work
Anatomy of Unit Testing

Concerns to address when testing a single method

• Define data to use to invoke the method
• Define the expected outcomes/effects of the method
• Define how to compare the actual and expected values
• Run the tests
• Report the results to the programmer
Anatomy of Unit Testing

Define the expected outcomes/effects of the method

Example:

class Balloon{
    int x; int y; int rad;
    // full constructor ...

    // produce a balloon moved up by dy
    Balloon moveUpFun(int dy) {
        return new Balloon(this.x, this.y - dy, this.rad);
    }

    Balloon bOrig = new Balloon(10, 40, 20);
    Balloon bFun = new Balloon(10, 37, 20);

test: bOrig.moveUpFun(3) same as bFun
Anatomy of Unit Testing

Define the expected outcomes/effects of the method

Example:

class Balloon{
    int x; int y; int rad;
    // full constructor ...

    // produce a balloon moved up by dy
    Balloon moveUpImp(int dy){
        this.y = this.y + dy; return this;
    }
}

Balloon bOrig = new Balloon(10, 40, 20);
Balloon bImp = new Balloon(10, 37, 20);
Balloon bResult = bOrig.moveUpImp(3);

test: bResult same as bImp
Anatomy of Unit Testing

Define the expected outcomes/effects of the method

Example:

class Balloon{
    int x; int y; int rad;
    // full constructor ...

    // produce a balloon moved up by dy
    Balloon moveUpBad(int dy){
        this.x = this.x + dy; this.y = this.y - dy;
        return this;
    }
}

Balloon bOrig = new Balloon(10, 40, 20);
Balloon bBad = new Balloon(10, 37, 20);
Balloon bResult = bOrig.moveUpBad(3);

test: bResult same as bBad
Anatomy of Unit Testing

Define the expected outcomes/effects of the method

Example:

class Balloon{
    int x; int y; int rad;
    // full constructor ...

    // produce a balloon moved up by dy
    Balloon moveUpBad(int dy){
        this.x = this.x + dy; this.y = this.y - dy;
        return this;
    }

    Balloon bOrig = new Balloon(10, 40, 20);
    Balloon bBad = new Balloon(10, 37, 20);
    Balloon bResult = bOrig.moveUpBad(3);

    test: bResult.y same as bBad.y
Anatomy of Unit Testing

Define the expected outcomes/effects of the method

Problems:

• Java does not support comparison by values needed for the first example

• Test for the second example does not guarantee that the same object is returned

• The test cannot be run twice - it would fail the second time

• Undetected additional effect in the third example if only the field values are compared

But most students do not see even this much
Anatomy of Unit Testing

Support for the following is needed:

• Understand clearly what needs to be tested
  ○ how the actual values are generated
  ○ how the expected values are defined

• Define the appropriate equality evaluation

• Define test cases that represent the actual and expected values and the relevant equality evaluation

• Run the tests, and collect the results

• Report the results to the programmer in an appropriate form
Anatomy of Unit Testing

Support for the following is needed:

• Understand clearly what needs to be tested
  ◦ how the actual values are generated
  ◦ how the expected values are defined

• Define the appropriate equality evaluation

• Define test cases that represent the actual and expected values and the relevant equality evaluation

• Run the tests, and collect the results

• Report the results to the programmer in an appropriate form

JUnit fails here...
Why the tester Library

The DESIGN RECIPE for every method:

• 1: Problem analysis and data definition
• 2: Purpose statement and the header
• 3: Examples with expected outcomes
• 4: Inventory/Template of available data fields and methods
• 5: Method body
• 6: Tests

Pedagogical advantages:

Each step is well defined
-- with a tangible result
-- with a guidance on what questions to ask
Why the tester Library: Test-First Advantages

Design tests first

• understand what data is needed for the method
• understand what are the expected outcomes
• gain insight into how the method behaves

Evaluating the tests

• define additional tests based on method design
• know that the expected behavior works
• for failed tests see what went wrong

Benefits

• simple methods -- simple tests
Why not JUnit?

Defining the tests

- **extends** `TestCase` before you see inheritance
- no access to private methods, fields
- new syntax, language

Evaluation of the test cases

- define your own `equals` method

Reporting of the results

- JUnit bar: red or green
- links to the line where the test failed
- `expected:<BTNode@c66e698d> but was:<BTNode@52d68153>`
Why not JUnit?

The key problems for a novice:

• extra language, syntax
• the need to define `equals` method
• Test result reporting uninformative

of course, we can teach students to do this...
they should learn how to do this...

... but not in their first week of Java
Why the tester Library: Motivation

Testing is hard
Java does not support comparing data by value
Defining such equality is hard for a novice
It increases the program complexity
Detracts from the focus on the program design

Learning to design tests, equality comparison, test reporting
• is a topic on its own
• we need pedagogy for that too

But: testing should be integrated into program design early
Why the tester Library: Preview

The tester library:

Tests are written as a part of the program design

Test library suitable for the beginner

• Tests compare data by their values
  ○ handle collections of data
  ○ handle circularity
  ○ handle random choice
  ○ handle tests of Exceptions
  ○ ... and more

• Test evaluation is automatic - compares data by their values
Why the tester Library: Preview

Automate the extensional equality comparison

The design of tests is simple

Support several test scenarios besides actual - expected model

Provides hooks for extensibility and user-defined equality evaluation

Results include pretty-printed values of actual and expected with the differences highlighted

Custom options for reporting of the results
Why the tester Library: How?

Automate the extensional equality comparison
* using Java reflection and annotations

The design of tests is simple
* the Example class acts as a client to student's code

Support several test scenarios besides actual - expected model

Provides hooks for extensibility and user-defined equality evaluation

Results include pretty-printed values of actual and expected with the differences highlighted

Custom options for reporting of the results
The Design of Tests and Test Evaluation

A wide range of test scenarios

- compare any two objects, including circularly defined
- compare two *inexact* objects
- compare two *Iterable* objects
- compare two *Map* objects
- `checkOneOf` a random set of values
- `checkRange` value within the given range: *Comparator*
- `checkNumRange` mixed numeric ranges
- test if a method throws *Exception* with the given message
- `checkFail` for test we want to fail
User options

• user can define several classes with test methods

• user can implement own equality:

\[
\text{interface ISame<T>}{ \text{ boolean same(T t); } } \]

• checkEquivalence user implements Equivalence interface

\[
\text{interface Equivalence<T>}{ \text{ boolean equivalent(T t1, T t2); } } \]

• user can annotate any method to be a test method

• user can include test methods within class definition
  ◦ this provides access to \texttt{private} fields and methods

• \texttt{Printer.print(Object obj)} pretty-prints any object
Let's compare:

Binary Search Tree: ABST, Node, Leaf

- Test the add method - build a tree
- we want to make sure the tree is built correctly
- the test should compare two trees

Defining equals method

- Three classes: needs to use `getClass`
- Should override `hashCode`
- ... and test that both work correctly ...

Define toString method

- to make sure the results are meaningful
Defining the equals method for the Node class:

```java
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Node<T> other = (Node<T>) obj;
    if (data == null) {
        if (other.data != null)
            return false;
    } else if (!data.equals(other.data))
        return false;
    if (left == null) {
        if (other.left != null)
            return false;
    } else if (!left.equals(other.left))
        return false;
    if (right == null) {
        if (other.right != null)
            return false;
    } else if (!right.equals(other.right))
        return false;
    return true;
}
```
Test case definition:

```java
// test the method add that builds ABST
public void testInsert() {
    assertEquals(tree3, tree3a);
    assertEquals(tree4, tree4a);
    assertEquals(tree4, tree4b);
}

// test the method add that builds ABST
class Tester t) {
    t.checkExpect(tree3, tree3a, "same trees");
    t.checkExpect(tree4, tree4a, "same trees");
    t.checkExpect(tree4, tree4b, "different trees");
}
```
Test results:

```
Failure Trace

<table>
<thead>
<tr>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>junit.framework.AssertionFailedError: expected:<a href="mailto:Node@2206331a">Node@2206331a</a> but was:<a href="mailto:Node@f43a7d5c">Node@f43a7d5c</a></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>at TreeTests.testInsert(TreeTests.java:51)</td>
</tr>
</tbody>
</table>
```
Test results:

error in test number 3
different trees

test ErrorReport: Error trace:
at ExamplesTrees.testTreeAdd(ExamplesTrees.java:88)
at ExamplesTrees.main(ExamplesTrees.java:95)

actual:
new Node:1
  this.$comp = new Book$BookByTitle:2
  this.data = new Book:3
    this.title = "HDL"
    this.author = "MF"
    this.price = 0
  this.left = new Node:4
    this.$comp = Book$BookByTitle:2
    this.data = new Book:5
      this.title = "Es"
      this.author = "EBM"
      this.price = 0
    this.left = new Leaf:6
      this.$comp = Book$BookByTitle:2
      this.right = Leaf:6
    this.right = new Node:7
      this.$comp = Book$BookByTitle:2
      this.data = new Book:8
        this.title = "OMGS"
        this.author = "Hemingway"
        this.price = 0
      this.left = new Node:9
        this.$comp = Book$BookByTitle:2
        this.data = new Book:10
          this.title = "Little Lisper"
          this.author = "MF"
          this.price = 0
        this.left = Leaf:10
        this.right = Leaf:6
    this.right = Leaf:6

expected:
new Node:1
  this.$comp = new Book$BookByTitle:2
  this.data = new Book:3
    this.title = "HDL"
    this.author = "MF"
    this.price = 0
  this.left = new Node:4
    this.$comp = Book$BookByTitle:2
    this.data = new Book:5
      this.title = "Es"
      this.author = "EBM"
      this.price = 0
    this.left = new Leaf:6
      this.$comp = Book$BookByTitle:2
      this.right = Leaf:6
    this.right = new Node:7
      this.$comp = Book$BookByTitle:2
      this.data = new Book:8
        this.title = "OMGS"
        this.author = "Hemingway"
        this.price = 0
      this.left = new Node:9
        this.$comp = Book$BookByTitle:2
        this.data = new Book:10
          this.title = "Little Lisper"
          this.author = "MF"
          this.price = 0
        this.left = Leaf:10
        this.right = Leaf:6
    this.right = Leaf:6
Reporting of the Results: Formatting

Automatic pretty-printing of any object
Optional comment allowed for every test
Print the actual and expected values juxtaposed
Highlight the first place where the values differ
Provide a link to the failed test
Allow user-defined `toString` method
Print individual items for arrays, `Iterable`
Reporting of the Results: Style

Selects all methods in the **Examples** class that start with `test...`

Selects all methods in the **AnyName** class when supplied as runtime argument to `tester.Main`

Selects all methods in the class annotated as `@Example` that are annotated with `@Test`

Test methods return type is either `boolean` or `void`

Test case value is either `boolean`
Reporting of the Results: Style

Automatically reports warnings for all inexact comparisons

Optionally displays all fields in the Examples class

User may select to view all test results (successes and failures)
Our Experiences

Tester library

Classroom trials:

Spring 2008 -- the first prototype

Fall 2008 -- beta version used at five institutions

- Northeastern University
- Worcester Polytechnic Institute, Worcester, MA
- Seton Hall University, South Orange, NJ
- duPont Manual High School, Louisville KY
- Millard Public Schools, Omaha, NE

Spring 2009 -- fully deployed, new users added

- Vassar College, Poughkeepsie, NY -- in a regular Java course
Our Experiences

Tester library

Used with hundreds of students throughout the semester

• Students get real feedback on validity of their programs
• Students believe testing matters
• Students understand why smaller methods are better
• Students explore the design of the tester
• New appreciation of the meaning of equality
• Some get excited about testing!

Colleague reports uniformly positive
Our Experiences -- curriculum overall

Yearly surveys done for over 10 years:

Coop employers report higher expectations of students
Students exceed even the higher expectations

Instructors in follow-up courses: students are better prepared
on pretest 30 percent failure reduced to 1 percent

Very low attrition rate (<5%)

Students are much more confident in their understanding of program design

Dissemination:

Workshops in summer 2007, 2008, 2009 at four US locations

A growing number of followers
Future Work

Goal: curriculum support for testing at all levels

• Better tutorials, examples, exercises
• Detailed guide for testing effects
• Refactor the test suite for tester
  ○ make it an example of a regression test suite
• Pedagogy for regression testing
• Integrated testing for advanced courses
Future Work

Goal: support for a seasoned programmer

• make the tester the learning ground for defining own tests, evaluation
• include a coverage tool
• include automatic test generation *QuickCheck* style
• define semantics of testing effects and provide support for it
• special support for effects on complex data structures
• refactor the tester internals
• explore compiler extension approach of Gray and Mycroft
THANK YOU

NSF support:

• Redesigning Introductory Computing: The Design Discipline
  ◦ DUE-0618543
• Integrating Test Design into Computing Curriculum from the Beginning
  ◦ DUE-0920182

http://www.ccs.neu.edu/javalib

Main site for the TeachScheme/ReachJava! project:

• http://www.teach-scheme.org