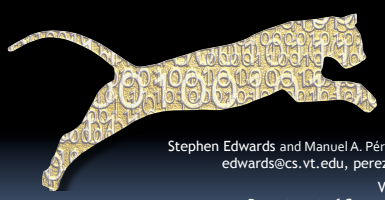
The Absolute Beginner's Guide to JUnit in the Classroom



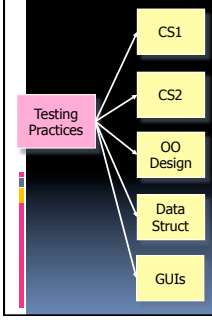
Stephen Edwards and Manuel A. Pérez-Quiñones
edwards@cs.vt.edu, perez@cs.vt.edu
Virginia Tech
Department of Computer Science
http://web-cat.org/

NSF DUE-0633594 and DUE-0618663

More educators are adding software testing to their programming courses

- Now it's almost routine
- Tools like **JUnit**, and XUnit frameworks for other languages, make it much easier
- Built-in support by many mainstream and educational IDEs makes it much easier
- Many instructors have also experimented with automated grading based on such testing frameworks

Why have we added software testing across our programming core?



- Students **cannot test** their own code
- Want a **culture shift** in student behavior
- A single upper-division course would have **little impact** on practices in other classes
- So: Systematically incorporate testing practices across many courses

What is JUnit?

- A **unit testing** framework
- To understand this, we need to know …

Basic testing terms

- Unit testing (one programmer's work)
- Integration testing (many units together)
- End-to-end testing ("whole program")
- Acceptance testing (does the customer like it?)
- Regression testing (re-running the same tests)

What is a software test?

- A specific plan for how to **execute** a piece of software
- Together with a method for deciding whether it **behaves as intended**
- A **test case** is a single software test, usually focused on checking just one situation or behavior
- A **test suite** is a collection of test cases, usually run as a group

What is the goal of software testing?

- Testing **cannot prove** software is correct
- … But testing **can prove** software still contains bugs
- The goal of testing is to **find bugs**
- … So a successful test run is one that **reveals one or more bugs**

JUnit is about *automating* tests

- Tests are written in the form of **program code**
- They are **executable**
- They can be repeated **any time**, for **free**

JUnit was created to support TDD

- Test-driven development
- "**Write a little code, write a little test**"
- TDD involves writing new tests for each small addition you make to your code
- TDD involves constantly re-running tests you have written so far each time you make a change
- Regression testing improves confidence that changes work exactly as intended

Test-driven development is very accessible for students

- Also called "test-first coding"
- Focuses on thorough unit testing at the level of individual methods/functions
- "Write a little test, write a little code"
- Tests come first, and describe what is expected, then followed by code, which must be revised until all tests pass
- Encourages lots of small (even tiny) iterations

Students can apply TDD and get immediate, useful benefits

- Conceptually, easy for students to understand and relate to
- **Increases confidence** in code
- **Increases understanding** of requirements
- Preempts "big bang" integration

The basic steps involved in a test

1. Set up the "initial conditions" for the test
2. Carry out the action(s) you want to test
3. Check that the desired result(s) were achieved
4. Clean up

The JUnit version of the basic steps

1. Create a test class
2. Set up the "initial conditions" in **setUp()**
3. Write individual tests as **test methods**:
   a. Carry out the action(s) you want to test
   b. Check that the desired result(s) were achieved
4. Clean up using **tearDown()** (rarely needed)

Let's learn about JUnit with live examples

Assertion methods in JUnit tests

- assertEquals(x, y);
- assertEquals(x, y, delta);
- assertSame(x, y);
- assertTrue(x);
- assertFalse(x);
- assertNull(x);
- assertNotNull(x);

Lets look at examples using JUnit 4

Any example situations you would like to discuss?

How can you use testing in the classroom?

Five common ways of using testing in the classroom …

- As part of an assignment specification
- Acceptance testing
- Automated grading
- Students write their own tests for their own code
- Students write tests to learn testing and debugging

As part of an assignment specification

- Provide downloadable test cases in the assignment
- Students run the tests as a sanity check, compliance to assignment specification
- Details of method names, signatures, interfaces are checked at compilation time
- Gives student direct evidence that program runs as expected

Acceptance testing

- Instructor uses unit test in grading process
- Professor gets compile-time compliance to specification
- Professor gets behavioral checks when tests are run
- Consistency in grading assignments
- Ability to run all students as batch process (e.g. JAM*Testor)

Automated Grading

- Similar to acceptance test, but with fully automated workflow
- Students can get immediate feedback
- Supports multiple submission, tight feedback cycle

Students write their own testing code

- Better quality, fewer bugs
- Students are required to articulate understanding of the behavior of their code
- Testing is experimentally verifying that code behaves as the student expects (or intends)
- Grade students on how well they test their code, not just whether it works or not

Students write test to learning testing and debugging

- Give students buggy code and ask them to write tests to expose the bug
- Fix bug and retest to confirm their fix works

Writing assignments so they can be tested easily

The most important step in writing testable assignments is …

- Learning to write tests yourself
- Writing an instructor's solution **with tests** that thoroughly cover all the expected behavior
- Practice what you are teaching/preaching
- Extra effort before assignment is "opened" (more prep time) but less effort after assignment is due (less grading time)

Areas to look out for in writing "testable" assignments

- How do you write tests for the following:
  - Testing exceptional conditions
  - Main programs
  - Code that reads/write to/from stdin/stdout or files
  - Assignments with lots of design freedom
  - Code with graphical output
  - Code with a graphical user interface

Testing exceptional conditions

- Unexpected exceptions are handled automatically by JUnit
- If you want to test explicitly thrown exception:
  - JUnit 3, use try/catch
  - JUnit 4, add 'expected' parameter @test annotation

Testing main programs

- The key: think in object-oriented terms
- There should be a principal class that does all the work, and a **really short** main program
- The problem is then simply how to test the principal class (i.e., test all of its methods)
- Make sure you specify your assignments so that such principal classes provide enough accessors to inspect or extract what you need to test

Testing input and output behavior

- The key: specify assignments so that input and output use streams given as parameters, and are **not hard-coded** to specific sources destinations
- Then use string-based streams to write test cases; show students how
- In Java, we use Scanners and PrintWriters for all I/O
- In C++, we use istreams and ostreams for all I/O

Assignments with lots of design freedom

- Allowing design freedom is good so students can learn design
- Two kinds of design freedom:
  - Students can make different design choices to implement the same required behavior
  - Students have latitude to add their own individual additions or flourishes or extras

When students implement same behavior in different ways

- Good for practicing design skills
- To test required behavior, use a fixed API that encapsulates the design freedom
- Write reference test against that API

When students add their own extras

- Good to encourage creativity and individual expression
- Limit instructor tests to only required features
- Write flexible tests that don't impose extra (hidden) assumptions
- Have students write their own test for their extensions

Mock objects can also help

- A mock object is a 'conveniently stubbed out' replacement for the real thing for use in testing
- Allows to decouple object being tested from other object dependencies
- Substitute behavior that is convenient for testing for real behavior
- Google 'JUnit mock objects' for more information

Testing programs with graphical output

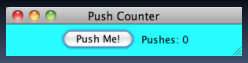
- The key: if graphics are only for output, you can ignore them in testing
- Ensure there are enough methods to extract the key data in test cases
- We used this approach for testing Karel the Robot programs, which use graphic animation so students can observe behavior

Testing programs with graphical UIs

- This is a harder problem—maybe too distracting for many students, depending on their level
- The key question: what is the goal in writing the tests? Is it the GUI you want to test, some internal behavior, or both?
- Three basic approaches:
  - Specify a well-defined boundary between the GUI and the core, and only test the core code
  - Switch in an alternative implementation of the UI classes during testing
  - Test the actual GUI (see our SIGCSE 08 paper)

Testing a GUI

- Button increments a counter
- Button is embedded in a panel that is self contained
- Main program creates a window, puts the panel in it and makes it visible



LIFT is our library for testing GUIs

- Student friendly
- Easy to write JUnit test for Swing, JTF, and objectdraw
- Android version called RoboLIFT
- See our SIGCSE 2011 and 2012 papers on LIFT and RoboLIFT

Lessons learned writing testable assignments

- Requires greater clarity and specificity
- Requires you to explicitly decide what you wish to test, and what you wish to leave open to student interpretation
- Requires you to unambiguously specify the behaviors you intend to test
- Requires preparing a reference solution before the project is due, more upfront work for professors or TAs
- Grading is much easier as many things are taken care by Web-CAT; course staff can focus on assessing design

If you give students tests instead of writing their own

- Students appreciate the feedback from tests, but will **avoid thinking** more deeply about the problem
- Seeing the results from a complete set of tests discourages student from thinking about how to check about their solution on their own
- This **limits the learning benefits**, which come in large part from students **writing their own** tests
- Lesson: balance providing suggestive feedback without "giving away" the answers: **lead the student** to think about the problem

Conclusion: including software testing promotes learning and performance

- If you require students to write their own tests …
- Our experience indicates students are more likely to complete assignments on time, produce one third less bugs, and achieve higher grades on assignments
- It is definitely more work for the instructor
- But it definitely improves the quality of programming assignment writeups **and** student submissions

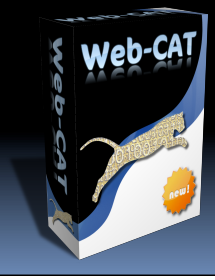
Visit our Community Site

- http://web-cat.org/
- Info about using our automated grader, getting trial accounts, etc.
- Movies of making submissions, setting up assignments, and more
- Custom Eclipse and Visual Studio plug-ins for C++-style TDD
- Links to our own Eclipse feature site

Thank you!

- Our community is our most valuable asset!

http://web-cat.org



It is time for any final questions …

- About anything covered …
- About how we've used these techniques in courses
- About how we start our freshmen out in the very first lab
- About the availability of Web-CAT
- … Or anything else you want to ask