Adding Software Testing to Programming Assignments

Stephen Edwards
Virginia Tech

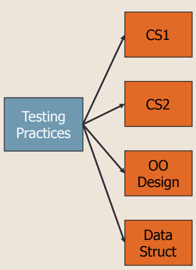
http://web-cat.org/cseet2013

My goals today are to . . .

Answer these questions:

- How can I **add** software testing to my assignments?
- How can I **assess** student testing efforts?
- How can I write "**testable**" assignments?
- And show you some **live examples**!

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

---

More educators are adding software testing to their program-ming 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
- Here are **my experiences** in teaching test-driven development with the help of an automated grader over the past 10 years

---

What kinds of assign-ments?

- Regular CS1 and CS2 assignments (of course!)
- Text adventure games
- Greenfoot-style micro-worlds
- **Asteroids**, MineSweeper
- AI computer players for Battleship!, Tetris, and more
- Random maze explorers
- Swing **GUI applications** (even 2D drawing editors)
- **Android apps** (even 2D and physics-based games, and map-based geotagged photo apps)
- Parsers and interpreters for PL courses

Just this semester

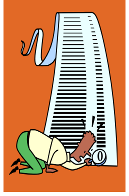
- CS1
  - Jeroo maze explorer
  - "Invasion of the Greeps" contest
  - Battleship!
  - Asteroids
  - "Design your own" game
- CS2
  - Adventure Time! (Android text adventure)
  - Maze solver app
  - Yelp restaurant guide
  - "Design your own" Android app

Software testing helps students frame and carry out experiments

- The **problem**: too much focus on synthesis and analysis too early in teaching CS
- Need to be able to read and comprehend source code
- Envision how a change in the code will result in a change in the behavior
- Need explicit, continually reinforced practice in **hypothesizing** about program behavior and then **experimentally verifying** their hypotheses

Expect students to apply testing skills all the time

- Expect students to **test their own work**
- **Empower** students by engaging them in the process of assessing their own programs
- **Require** students to demonstrate the correctness of their own work through testing
- Do this consistently **across many courses**



What tools and tech-niques should we teach?

- We want to start with skills that are **directly applicable** to authentic student-oriented tasks
- Don't want to add bureaucratic **busywork** to assignments
- Without tool support, this is a lost cause!
- It is imperative to give students skills they **value**
- ... But most textbooks only give a "conceptual" intro to idealized industrial practices, not techniques students can use in their own assignments

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

TDD tools are widely, freely available

- Lots of open-source tools, particularly for OO languages
- JUnit (for Java): http://junit.org/
- XUnit links (for other languages): http://xprogramming.com/software/
- We use tools like this for Java, C++, Scheme, Prolog, Haskell, and even Pascal in our courses

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 (often unneeded in Java)

Suppose we have a class for DVR record-ings

```
public class DvrRecording
{
  private String title;
  private int duration;

  public DvrRecording(
    String title, int duration)
  {
    ...
  }

  public String getTitle() { ... }
  public int getDuration() { ... }
  public String toString() { ... }
}
```

```
public void testToString()
{
  // 1. Initial conditions
  DvrRecording recording =
    new DvrRecording("Lost", 60);

  // 2. Action to test
  String output =
    recording.toString();

  // 3. Check expected results
  assertEquals(
    "Lost [60 min.]", output);
}
```

A test might look like this

Naming/signature convention

```
public void testToString()
{
  DvrRecording recording =
    new DvrRecording("Lost", 60);
  assertEquals(
    "Lost [60 min.]",
    recording.toString());
}
```

Assertions compare expected and actual outcomes

The same, but shorter

```
private DvrRecording recording;

// Initial conditions for all tests
public void setUp()
{
  recording =
    new DvrRecording("Lost", 60);
}

public void testToString()
{
  assertEquals(
    "Lost [60 min.]",
    recording.toString());
}
```

Always starts in a clean starting state

With common setup factored out

```
private DvrRecording recording;

@Before
public void setUp()
{
  recording =
    new DvrRecording("Lost", 60);
}
@Test
public void testToString()
{
  assertEquals(
    "Lost [60 min.]",
    recording.toString());
}
```

Annotations instead of inheritance

No more naming conventions

The same, but in JUnit 4

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)

My goals today are to . . .

Answer these questions:

- How can I **add** software testing to my assignments?
- How can I **assess** student testing efforts?
- How can I write "**testable**" assignments?
- And show you some **live examples**!

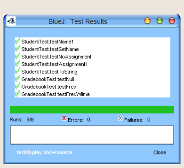
There are five main strategies for adding testing to assign-ments

1. Use test cases as **specifications**
2. Write "**acceptance tests**" for grading
3. Require **student-written tests** as part of the assignment
4. Use a **reference model** to assess student tests
5. Write assignments that focus on **testing and/or debugging** instead of writing code

A simple example will ground the discussion: our first live demo!

- Let's switch to an IDE for this example
- I'll use Eclipse and Java for this tutorial, but similar techniques apply in other IDEs or OO languages
- If you visit the workshop web site after the tutorial, you can find this under "Example 1: Building a Gradebook"

Tools make running tests easy



- Most XUnit frameworks include **test runners** that allow you to directly execute test cases from one class or many
- Often, either textual or graphical output is available
- Many IDEs include direct support for running such test cases (BlueJ, Eclipse, JGRASP, ...)

You can use test cases in assignment specifications

- 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

Assessing test cases in assignment specifications

Pros

- Students learn about test cases and how to use them
- Greater **clarity and precision** in assignments
- Easy to combine with "acceptance tests"

Cons

- Students don't learn to write tests
- **Least impact** on student behavior

Let's discuss …

- Questions about this example?
- Questions about how to apply this technique in an assignment?
- Questions about the costs or benefits?

1. Use test cases as **specifications**
2. Write "**acceptance tests**" for grading
3. Require **student-written tests** as part of the assignment
4. Use a **reference model** to assess student tests
5. Write assignments that focus on **testing and/or debugging** instead of writing code

There are five main strategies for adding testing to assign-ments

You can write "acceptance tests" to use for grading

- Write **your own** test suite(s) for grading
- Instructor or TA can run student code against your tests as part of the assessment process
- Can even be used in automated grading systems, if available
- Helps **standardize** the assessment of correctness

Using acceptance tests in grading

- An example:
- If the assignment is to write Student and Gradebook classes …
- Give instructor-written StudentTest and GradebookTest classes to the grader
- Run these tests against student submissions, and use the percentage of passed tests as a measure of program correctness

Assessing the use of acceptance tests

Pros

- Increased **consistency** and **thoroughness** of correctness assessment
- Easy to **automate**
- With appropriate automation, more grading time available for assessing design and giving feedback

Assessing the use of acceptance tests

Cons

- Students don't learn to **write tests**
- Students focus more on getting correct behavior, less on quality
- Acceptance tests only work for **tightly specified** assignments
- Difficult to accommodate **individual differences** in the design of student solutions (using an **adaptor** can help)
- Limited ability to assess solutions to **open-ended** problems

A test adaptor can decouple test cases from student designs

- **Don't** specify student design
- Instead, **do** specify the interface for a **test adaptor** that allows all the behavior to be explored
  - Students **create their own design**
  - Students also write their own **adaptor implementation**

- Careful adaptor interface design can ensure students don't use the adaptor as *their* design

Example 2: Appointments

- Let's switch to the IDE for this example
- If you visit the workshop web site after the tutorial, you can find this under:

"Example 2: Appointments"

Let's discuss …

- Questions about this example?
- Questions about how to apply this technique in an assignment?
- Questions about the costs or benefits?

---

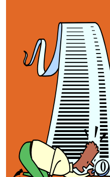
1. Use test cases as **specifications**
2. Write "**acceptance tests**" for grading
3. Require **student-written tests** as part of the assignment
4. Use a **reference model** to assess student tests
5. Write assignments that focus on **testing and/or debugging** instead of writing code

There are five main strategies for adding testing to assign-ments

---

- Expect students to **test their own work**
- **Empower** students by engaging them in the process of assessing their own programs
- **Require** students to demonstrate the correctness of their own work through testing
- Do this consistently **across many courses**

Expect students to apply testing skills all the time

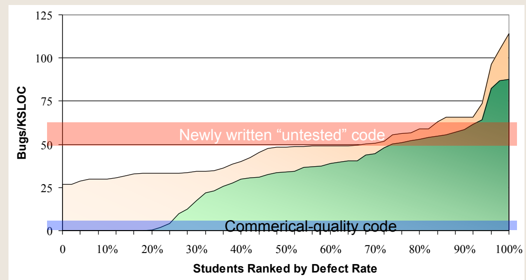


Requiring student-written tests

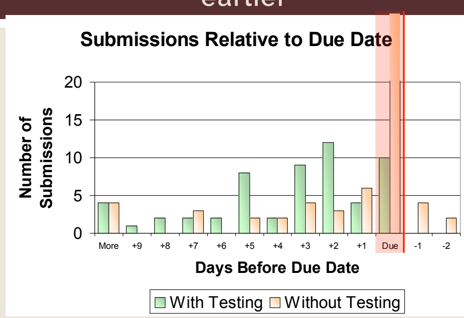
An example:

- If the assignment is to write Student and Gradebook classes …
- Require students to write their own StudentTest and GradebookTest classes as part of the assignment
- Require students to run these tests themselves, and then to include their test classes as part of their program submissions

Students improve their code quality when required to test



Students start earlier and finish earlier



Assessing student-written tests as part of programming assignments

Pros

- Students **learn to test**
- Increased **student confidence** in work
- **Increased understanding** of behavioral requirements
- More likely to turn assignments in **on time**
- **Reduce bug density** by one third or more

Assessing student-written tests as part of programming assignments

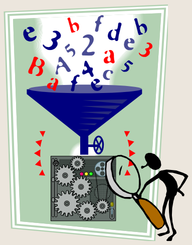
Cons

- How do you assess student performance at testing?
- How do you provide feedback so students improve their testing skills?
- More effort goes into writing assignments so they are testable
- Hard to do well without appropriate tool support

1. Use test cases as **specifications**
2. Write **"acceptance tests"** for grading
3. Require **student-written tests** as part of the assignment
4. Use a **reference model** to assess student tests
5. Write assignments that focus on **testing and/or debugging** instead of writing code

There are five main strategies for adding testing to assign-ments

You can use a "reference model" to assess student-written tests



- Some form of executable model of the problem can be used to gauge coverage of student tests
- A **reference solution** is typical
- Run student's tests against the reference solution to assess correctness of student tests
- **Instrument** reference solution to assess the "thoroughness" or completeness of student tests

Using a reference model

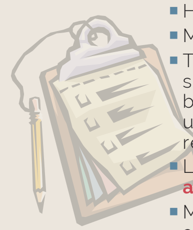
- An example:
- If the assignment is to write Student and Gradebook classes …
- Write your own solution to the problem
- Run student-written tests against this reference model to confirm their correctness
- Optionally, use data about which portions of the reference model were executed to assess completeness of testing

Assessing the use of a reference model in assessing tests

Pros

- Provides consistency and detail in assessing student testing skills
- Can automate assessment of student-written tests
- May lead to more detailed feedback

Assessing the use of a reference model in assessing tests



Cons

- Have to write the model
- May have to instrument it by hand
- Ties down student-written tests so they can only examine behavior in the reference model, using signatures present in the reference model
- Limits use of **open-ended assignments**
- May require **overspecification** of assignment

---

1. Use test cases as **specifications**
2. Write "**acceptance tests**" for grading
3. Require **student-written tests** as part of the assignment
4. Use a **reference model** to assess student tests
5. Write assignments that focus on **testing and/or debugging** instead of writing code

There are five main strategies for adding testing to assign-ments

Write assignments on testing or debugging, rather than coding

- To promote comprehension and analysis skills …
- Give students some existing (buggy) code
- Require them to write tests, find the bugs, and repair them
- You can combine this with instructor-written acceptance tests, assessment of student-written tests, or both

Using testing/debugging assignments

- An example:
- See the Bricks example on the web site
- It includes a small project containing two buggy classes
- Students are instructed to write tests, find the bugs, and repair them
- Students submit their tests and their repaired code, and are given feedback on how many of the hidden bugs they have found

Assessing assignments that focus on testing and debugging

Pros

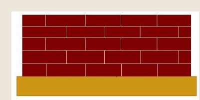
- Directs student attention **solely** at testing and/or debugging skills
- Gives **specific feedback** on these skills, rather than on code writing
- Really promotes **comprehension** and **analysis**
- Students work with code from other authors, rather than writing from scratch

Assessing assignments that focus on testing and debugging

Cons

- Requires a different assessment strategy than conventional programming assignments
- Giving concrete, directed feedback is important for students to improve, but may take more time to provide
- Can take time to write, compared to code writing assignments

Example 3: Debugging Bricks



Let's discuss …

- Questions about this example?
- Questions about how to apply this technique in an assignment?
- Questions about the costs or benefits?

My goals today are to . . .

Answer these questions:

- How can I **add** software testing to my assignments?
- How can I **assess** student testing efforts?
- How can I write "**testable**" assignments?
- And show you some **live examples**!

First, realize that it *is* more work!

- If a student turns in both code and tests, then you need to consider assessing (and giving feedback on!):
  - Code correctness
  - Code quality (design, style, etc.)
  - Test suite correctness
  - Test suite thoroughness (coverage)
  - Test suite quality

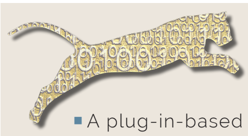
Second, work within the limits of your manpower

- You may not be able to do it all
- Prioritize which items you wish to devote your resources toward
- Plan an assessment budget:
  - For example, you can break down the expected time for grading one assignment into the expected time to spend on each of the separate subcategories

Use auto-mation where you can

- Best targets of opportunity:
  - Code correctness
  - Test suite correctness
  - Test suite thoroughness (coverage)
- Available tools:
  - Acceptance tests, student-written tests, a reference solution (may be instrumented), …

What is Web-CAT ?



- A plug-in-based web application
- Supports **electronic submission** and **automated grading** of programming assignments
- Fully customizable, scriptable grading actions and feedback generation
- Lots of support for grading students based on **how well they test their own code**

The Java plug-in grades assignments that include student tests

- **ANT**-based build of arbitrary Java projects
- **PMD** and **Checkstyle** static analysis
- ANT-based execution of student-written JUnit tests
- Carefully designed Java **security policy**
- **Clover** test coverage instrumentation
- ANT-based execution of **optional instructor reference tests**
- Unified HTML web printout
- **Highly configurable** (PMD rules, Checkstyle rules, supplemental jar files, supplemental data files, java security policy, point deductions, and lots more)

Our strategy is a hybrid of techniques

- We require student-written tests for **everything**
- We use automation to:
  - Run student tests on student code
  - Instrument student code to measure test coverage
  - Run acceptance tests on student code
  - Run static analysis tools on student code (style, commenting, etc.)
  - Combine measures into an appropriate score

Assessing student tests is tricky, so we use complementary methods

- First, we measure how many of the student's own tests pass
- Second, we instrument student code and **measure code coverage** while the student's tests are running
- Third, we use instructor-provided **reference tests** to cross-check the student's tests
- We **multiply the percentages** together, so students must excel at all three to increase their score

My goals today are to . . .

Answer these questions:

- How can I **add** software testing to my assignments?
- How can I **assess** student testing efforts?
- How can I write "**testable**" assignments?
- And show you some **live examples**!

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

How do you write tests for:

- 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 to the @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 BufferedReaders and PrintWriters for all I/O
- In C++, we use istreams and ostreams for all I/O

Or use JUnit add-ons (Ex 4)

```
public static void main(String[] args)
{
    System.out.println("Hello world!");
}


public void testMain()
{
    HelloWorld.main(null);
    assertEquals("Hello world!\n",
      systemOut().getHistory());
}
```

Or use JUnit add-ons (Ex 4)

```
public static void main(String[] args)
{
    System.out.println("Hello world!");
    System.exit(0);
}

public void testMain()
{
    try
    {
        HelloWorld.main(null);
    }
    catch (ExitCalledException e)
    {
        assertEquals(0, e.getStatus());
    }
    assertEquals("Hello world!\n",
        systemOut().getHistory());
}
```

Our testing library provides …

In our student.jar library:

- Set **stdin** in test cases
- Get history of **stdout** (cleanly reset for each test)
- Newline normalization for output
- System.**exit()** throws exception
- Better error messages for student assertion mistakes
- "**Fuzzy**" string matching (ignore caps, punctuation, spacing, etc.)
- **Regular expression** and fragment matching
- Adaptive **infinite loop** protection during grading
- Swing GUI testing through **LIFT**

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
- **Or**, just test common/required elements, and let **students be responsible** for testing the rest

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 decoupling 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

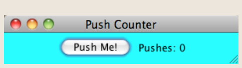
- The key: if graphics are only for output, you can ignore them in testing
- Ensure there are enough methods to extract the key data in test cases
- We use this approach for testing Karel the Robot programs, which use graphic animation so students can observe behavior

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 by simulating GUI events

Example 5: 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

It is time for any final questions …

- About anything covered …
- About how I' 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

Thank You!

- Our community is our most valuable asset!

http://web-cat.org