What is unit testing?

- "Software test" --- a check on the behavior of some piece of code
- "Unit test" --- a test on a single unit (usually a single class, or a single method)
- Key ideas:
  - Write test in the form of program code
  - So it can be automatically repeated

Why?

- For you:
  - Repeatedly check behaviors of many student programs

- For students (when they do it):
  - Repeatedly check code so far works, as they make changes

Why?

- More work up front in writing an assignment (you have to be more careful)
- You have to write a solid solution, too!
- But this work buys you advantages in the long run
  - Better, more carefully thought out assignment writeups
  - Ability to automatically check behavior of student solutions

Basic Steps to Create a Test Class

- A test case is an individual test for a specific behavior in a unit
- A claim or assertion is a statement expressing the behavior or outcome we expect in a test case
- A test fixture is the name for the initial conditions used in one or more test cases
- A test suite is a collection of test cases
We write our tests in the form of code.

An individual test case is written in the form of a single method.

Test case methods are collected together into a test class.

Each test class focuses on testing the features of one class we have written.

Each test class embodies one test fixture (one set of initial conditions for all the test cases it contains).

### Organizing tests

#### In Java (using JUnit):

- We write our tests in the form of code.
- An individual test case is written in the form of a single method.
- Test case methods are collected together into a test class.
- Each test class focuses on testing the features of one class we have written.
- Each test class embodies one test fixture (one set of initial conditions for all the test cases it contains).

### 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).

### Public class DvrRecording

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

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

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

### Suppose we have a class for DVR recordings

Let's suppose we have a class for DVR recordings:

```java
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);
}
```

### Wrapped inside a basic class

```java
public class DvrRecordingTest extends TestCase {
    public void testToString() {
        ...
    }
}
```

### The same, but shorter

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

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());
}

The same, but in JUnit 4

---

Beginner’s Strategy

- Keep tests narrowly focused
- Write a separate test class for class you need to test
- As a starting point, group all tests for one class into a single test class
- Example: class ArrayQueue has all its tests in test class ArrayQueueTest

Group tests into classes

- Focus on testing one method at a time
- For each method, write one or more tests
- Use a different test for each distinct situation/behavior you want to test
- One test for simple methods, multiple tests for complex methods
- Example: Method enqueue() might have separate tests for adding to an empty queue, or a non-empty queue

Test each method individually

- While each test should focus on one method...
- You might need to use other methods to set up the “initial conditions”
- This is perfectly OK
- Example: Using multiple enqueue() calls to set up the initial conditions for testing dequeue()
Write assertions to test all of the expectations you have about what a method does.

For a "function", just testing the return value is typical.

For more complex behaviors, use your object's accessors to make claims about any relevant object properties.

assertEquals("these don't match!", expected, actual);
fail("something broke");

The message is optional.
Provided as the first parameter.
Used as the exception message if an assertion fails.

Most common:
assertEquals(expected, actual);
assertTrue(expression);
assertFalse(expression);
assertEquals(d1, d2, tolerance);

Less common:
assertNull(expression);
assertSame(expected, actual);
assertNotSame(expected, actual);
fail();

All asserts can take a message.

JUnit Tips

public class StudentTest extends TestCase
{
    // fixture to be used for testing
    private Student aStudent;

    public void setUp()
    {
        // initialize it here
        aStudent = new Student("Joe", "888-2993");
    }

    // all tests can use fixture
}

import student.TestCase;
public class StudentTest extends TestCase
{
    // can access to extra features!
}

Use fixtures in your test cases.

Use our custom base class.
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

---

Our testing library provides...

---

import student.TestCase;

public class HelloWorldTest extends TestCase
{
    public void testMain()
    {
        // Call main()!
        HelloWorld.main(null);
    }
}

---

import java.util.Scanner;

public class HelloWorld
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.next();
        System.out.println("Hello, "+ name + "!");
    }
}

---

Consider this example

---

public void testMain()
{
    // Don't forget the newline!
    setSystemIn("Joe\n");
    HelloWorld.main(null);
    assertEquals("Enter your name: Hello, Joe!\n", systemOut().getHistory());
}

---

Set contents of standard input

---

public void testMain()
{
    // Don't forget the newline!
    setSystemIn("Joe\n");
    HelloWorld.main(null);
    assertEquals("Enter your name: Hello, Joe!\n", systemOut().getHistory());
}

---

Test just part of the output, as needed

---

public void testMain()
{
    setSystemIn("Joe\n");
    HelloWorld.main(null);
    assertTrue(systemOut().getHistory().contains("Hello, Joe!"));
}
public void testMain()
{
    HelloWorld.main(
        new String[] { "Joe" });
    ...
}

Providing command line args

What if main calls exit()?

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());
}

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

JUnit 3 example

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

JUnit 4 example

@Test(expected = Exception.class)
public void testWithException()
{
    try
    {
        someObject.blowsUp();
        // Shouldn't reach here
        fail("Didn't throw!");
    }
    catch (Exception e)
    {
        // If we reach here, it worked
        // so no action necessary
    }
}

Tools and test runners
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, ...)

Tools make running tests easy

Adding Tests to Assignments

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 assignments

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 their learning ...
Learn to write tests yourself first!
- Don’t expect to teach students to write tests if you’ve never done it before
- Add unit tests gradually
- Try it out for yourself first
- Build up some experience before you ask students to write their own

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

Areas to look out for

But ...

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

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

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

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 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

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
Our community is our most valuable asset!

http://web-cat.org

Thank You!