Testing and Bags

1/8/16
Announcements

• **Reading**
  – For Monday: Java Interlude 1 & Chapter 2 thru “Testing the Core Methods” (pages 53-72)
  – For Wednesday: Rest of Chapter 2 and Java Interlude 2

• HW hopefully out today or tomorrow (by email)

• If you borrow a clicker, send me an email to let me know
Talk about lab

I thought we did that last time...
Testing
Automated testing

• Nice feature of codingbat: it tells if your code is correct without a human looking at it

• Want same thing for large coding projects so you can tell if changes break part of the code

• Relies on thorough test cases
  – each is a call to your code an expected answer
Black-box Testing

- You don’t know (or you pretend you don’t know) how something is implemented
- You test only based on inputs and outputs

Clear-box Testing

- (Also known as “white-box testing”)
- If you can look inside the black box and see how a method is implemented, you can do more detailed testing

Green slides taken (w/ minor modifications) from Cynthia Lee’s CS 2 slides on http://www.peerinstruction4cs.org, licensed under Creative Commons Attribution-NonCommercial 4.0 International License.
Designing clear-box tests

How many test cases would you need to do thorough clear-box testing on this code?

```java
public void mysteryMethod(int x){
    someMethod();
    if (x < 5){
        methodCall();
    } else {
        otherMethod();
    }
    anotherOne();
}
```

A. 1 test case  
B. 2 test cases  
C. 3 test cases  
D. > 3 test cases
Designing clear-box tests

How many test cases would you need to do thorough clear-box testing on this code?

```java
public void mysteryMethod(int x){
    someMethod();
    if (x < 5){
        methodCall();
    } else {
        otherMethod();
    }
    anotherOne();
}
```

A. 1 test case  
B. 2 test cases  
C. 3 test cases  
D. > 3 test cases
Discussion

- What do you think are advantages and disadvantages to each?
- When might you do black-box testing?
- When might you do clear-box testing?
Unit testing

• Whether you are doing black-box or clear-box testing, you should test every important unit of a software system

• What is a unit? In object-oriented programming, usually a software unit is taken to be a single method

• So: we should test every method of every class in the software

• JUnit is a widely used framework for unit testing of Java software...
JUnit basics

- To do unit testing in the JUnit framework:
  - Define a subclass of `junit.framework.TestCase`
  - Define public void no-argument methods with names that start with `test`. Each “testXXX” method should test a particular thing
- Optional:
  - Define instance variables that store the state of the “test fixture”, i.e. the objects that will be tested
  - Initialize the fixture state by overriding the `setUp()` method
  - Clean-up after a test by overriding `tearDown()` method
**Test Case and Test Fixtures**

- Define a subclass of `junit.framework.TestCase`
- Define instance variables that store the state of the “test fixture”, i.e. the objects that will be tested

```java
import junit.framework.*;

public class RectangleTester extends TestCase {

    private Rectangle r1, r2; // test fixtures
}
```

To inherit the testing and test run methods we will need.
setUp() and tearDown()

/* Called AUTOMATICALLY before each testXXX() method is run */
protected void setUp() {
    r1 = new Rectangle();
    r2 = new Rectangle(2.0, 3.0);
}

/* Called AUTOMATICALLY after each testXXX() method is run */
protected void tearDown() {
    r1 = null;
    r2 = null;
}

Make sure each test method starts with a clean copy of the test fixture.

setup();
testXXX();
tearDown();

This is the sequence of calls the JUnit framework does for you automatically for each test method testXXX() that is invoked.
Coding a test case

/** Test case 2.1: verify that default constructor sets default instance variable values correctly */
public void testDefaultInstance() {
    assertEquals(1.0, r1.getLength());
    assertEquals(1.0, r1.getHeight());
}

/** Test case 2.6: verify that mutator for length throws exception on illegal input. */
public void testMutatorIllegalInput() {
    try {
        r1.setLength(0); // 0 is illegal, should throw
        fail();
    } catch (IllegalArgumentException e) {
        // test passes
    }
}
More JUnit basics

- Test fixture instance variables are optional: you can use test method local variables instead.
- When you run your TestCase class as a program, each “testXXX” method is called automatically...
- If a Junit assertion fails when a testXXX method runs, that test fails.
- If testXXX method throws an exception, that is considered an error, not a test failure!
- If a testXXX method returns normally, that test passes.
JUnit 3.8.1 assertion methods

- The JUnit framework provides many useful assertion methods to use in your testXXX() methods

  ```java
  assertEquals(x, y) // fail if x is not equal to y
  assertTrue(b) // fail if b has boolean value false
  assertFalse(b) // fail if b has boolean value true
  assertSame(x, y) // fail if x and y point to different objects
  assertNotSame(x, y) // fail if x and y point to the same object
  assertNull(x) // fail if x is not null
  assertNotNull(x) // fail if x is null
  fail() // always fails
  ```

- All assertion methods are overloaded with a version that takes an additional first argument of type String, for a message
Short version for our class

• Select New -> JUnit test case in Eclipse
• Be sure to use JUnit 3
• Declare object references as attributes, create the objects (and set them up) in setUp
• Write testing methods with assertions
  – I almost exclusively use assertEquals
Specifying a data type or class

A class specification should generally include these:

- **ADT name:** The name of the data type.
- **ADT description:** A brief summary description of the type.
- **ADT invariants:** Assertions that must always be true of any instance of this type.
- **ADT attributes:** Aspects of the state of an instance of the type, as observable by a client.
- **ADT operations:** The behavior of an instance of the type, as observable by a client.
Specifying operations

Specifying class operations is important: these operations are the primary way that users interact with instances.

For each operation, you should specify:

- **responsibilities**: A brief summary of what the operation does.

- **pre-conditions**: What must be true on entry to the operation. This may include assumptions about the state of the object, assumptions about the parameters passed in, etc. The pre-conditions must be consistent with the invariants.

- **post-conditions**: What the operation guarantees to be true when it returns (if the pre-conditions were met). Must be consistent with the invariants.

- **returns**: The value, if any, returned by the operation.

- **exceptions**: Description of exceptions that can be thrown.
Class descriptions can be viewed as a contract between the implementer and the user; if the user meets the preconditions, then the method works as advertised

- Can test this independent of the rest of the code
- Why not make unit tests for each class before it is implemented?
Checkpoint

- What is the relationship between ADT or class invariants and an operation’s pre- and post-conditions?
- How do pre- and post-conditions relate to writing a test for the operation?
- Can you write a test for what the operation does when its pre-conditions are not met?
Bag ADT
Bag

- Unordered collection of objects (type T) that allows duplicates

size() //book calls it getCurrentSize
add(item) //returns whether it fits
contains(item)
clear() //empties Bag
remove(item) //removes item; returns if was there
isEmpty()
remove() //removes arbitrary item; returns it
getFrequencyOf(item)
toArray() //returns array with contents