Test Code Patterns

How to design your test code

Testing and Inheritance

- Should you retest inherited methods?
- Can you reuse superclass tests for inherited and overridden methods?
- To what extent should you exercise interaction among methods of all superclasses and of the subclass under test?

Inheritance

- In the early years people thought that inheritance will reduce the need for testing
  - Claim 1: “If we have a well-tested superclass, we can reuse its code (in subclasses, through inheritance) with confidence and without retesting inherited code”
  - Claim 2: “A good-quality test suite used for a superclass will also be sufficient for a subclass”
- Both claims are wrong.

Inheritance-related bugs

- Missing Override
  - A subclass omits to provide a specialized version of a superclass method
  - Subclass objects will have to use the superclass version, which might not be appropriate
  - E.g. method **`equals`** in **`Object`** tests for reference equality. In a given class, it might be right to override this behaviour

Inheritance-related bugs

- Direct access to superclass fields from the subclass code
  - Changes to the superclass implementation can create subclass bugs
  - Subclass bugs or side effects can cause failure in superclass methods
  - If a superclass is changed, all subclasses need to be tested
  - If a subclass is changed, superclass features used in the subclass must be retested

Testing of Inheritance

- Principle: inherited methods should be retested in the context of a subclass

- Example 1: if we change some method **m** in a superclass, we need to retest **m** inside all subclasses that inherit it

Example 2

```
class A {
  int x; // invariant: x > 100
  void m() { // correctness depends on
             // the invariant } }
```

```
class B extends A {
   void m2() { x = 1; }  }
```

- If we add a new method **m2** that has a bug and breaks the invariant, method **m** is incorrect in the context of **B** even though it is correct in **A**
  - Therefore, **m** should be tested in **B**

Example 3

```
class A {
void m() { …; m2(); … }
void m2() { … }     }
```

```
class B extends A {
     void m2() { … } }
```

- If inside **B** we override a method from **A**, this indirectly affects other methods inherited from **A**
  - e.g., method **m** calls **B.m2**, not **A.m2**: so, we cannot be sure that **m** is correct anymore and we need to retest it inside **B**

Testing of Inheritance (cont)

- Test cases developed for a method **m** defined in class **A** are not necessarily sufficient for retesting **m** in subclasses of **A**
  - e.g., if **m** calls **m2** in **A** and then some subclass overrides **m2** we have a completely new interaction that may not be covered well by the old test cases for **m**
- Still it is essential to run all superclass tests on a subclass
  - Goal: check behavioral conformance of the subclass w.r.t. the superclass (LSP)

Inheritance-related bugs

- Square Peg in a Round Hole
  - Design Problem
  - A subclass is incorrectly located in a hierarchy
  - Liskov Substitution Principle: Functions that use references to base classes must be able to use objects of derived classes without knowing it.

An example

- Consider class Rectangle below

```
class Rectangle{
  public void setWidth(double w) {itsWidth=w;}
  public void setHeight(double h) {itsHeight=w;}
  public double getHeight() {return itsHeight;}
  public double getWidth() {return itsWidth;}

  private double itsWidth;
  private double itsHeight;
};
```

An example

- Assume that the system containing Rectangle needs to deal with squares as well
- Since a square is a rectangle, it seems to make sense to have a new class Square that extends Rectangle
- That very “reasonable” design can cause some significant problems

Problems with this design

- Do not need both itsHeight and itsWidth
- setWidth and setHeight can bring a Square object to a corrupt state (when height is not equal to width)

One solution

```
class Square{
  setWidth(double w){
    super.setWidth(w);
    super.setHeight(w);
}
  // Similar for setHeight
}
```

Not really a solution

- Consider this client code

```
Rectangle r;
…
r.setWidth(5);
r.setHeight(4);
assert(r.getWidth() * r.getHeight()) == 20);
```

- The problem is definitely not with the client code

What went wrong?

- The Liskov substitution principle was violated
  - If you are expecting a rectangle, you can not accept a square
- The overridden versions of setWidth and setHeight broke the postconditions of their superclass versions
- Isn’t a square a rectangle? Yes, but not when it pertains to its behaviour

Effect of Inheritance on Testing?

- Does not reduce the volume of test cases
- Rather, number of interactions to be verified goes up at each level of the hierarchy

Polymorphic Server Test

- Consider all test cases that exercise polymorphic methods
- According to LSP, these should apply at every level of the inheritance hierarchy
- Expand each test case into a set of test cases, one for each polymorphic variation

An example

```
class TestAccount {
  Account a;
  @Before
  public void setUp(){
    a = new Account();
  }
  @Test
  public final void testDeposit(){
    a.deposit(100);
    assertTrue(a.getBalance()==100);
  }
}
```

An example

```
class TestSavingsAccount
                  extends TestAccount{
  SavingsAccount sa;
  @Before
  public void setUp(){
    a = new SavingsAccount();
    sa = new SavingsAccount();}
  @Test
  public void testInterest(){
    sa.deposit(100);
    sa.applyInterest();
    assertTrue(sa.getBalance()==101);
  }}
```

Testing abstract classes

- Abstract classes cannot be instantiated
- However, they define an interface and behaviour (contracts) that implementing classes will have to adhere to
- We would like to test abstract classes for functional compliance
  - Functional Compliance is a module's compliance with some documented or published functional specification

Functional vs. syntactic compliance

- The compiler can easily test that a class is syntactically compliant to an interface
  - All methods in the interface have to be implemented with the correct signature
- Tougher to test functional compliance
  - A class implementing the interface `java.util.List` may be implementing `get(int index)` or `isEmpty()` incorrectly
- Think LSP…

Abstract Test Pattern

- This pattern provides the following
  - A way to build a test suite that can be reused across descendants
  - A test suite that can be reused for future as-yet-unidentified descendants
    - Especially useful for writers of APIs.

An example

- Consider a statistics application that uses the Strategy design pattern

```
public interface StatPak
  {
    public void reset();
    public void addValue(double x);
    public double getN();
    public double getMean();
    public double getStdDev();
  }
```

Abstract Test Rule 1

- Write an abstract test class for every interface and abstract class
- An abstract test should have test cases that cannot be overridden
- It should also have an abstract Factory Method for creating instances of the class to be tested.

Example abstract test class

```
public abstract TestStatPak {
  private StatPak statPak;
  @Before
  public final setUp() throws Exception {
    statPak = createStatPak();
    assertNotNull(statPak);
  }
  // Factory Method. Every test class of a
  // concrete subclass K must override this
  // to return an instance of K
  public abstract StatPak createStatPak();
  //Continued in next slide…
```

Example abstract test class

```
@Test
public final void testMean() {
  statPak.addValue(2.0);
  statPak.addValue(3.0);
  statPak.addValue(4.0);
  statPak.addValue(2.0);
  statPak.addValue(4.0);
  assertEquals("Mean value of test data
  should be 3.0", 3.0,statPak.getMean());
}
@Test
public final void testStdDev() { ... }}
```

Abstract Test Rule 2

- Write a concrete test class for every implementation of the interface (or abstract class)
- The concrete test class should extend the abstract test class and implement the factory method

Example concrete test class

```
public class TestSuperSlowStatPak
   extends TestStatPak {

   public StatPak createStatPak()
   {
     return new SuperSlowStatPak();
   }
}
```

Only a few lines of code and all the test cases for the interface have been reused

Guideline

- Tests defining the functionality of the interface belong in the abstract test class
- Tests specific to an implementation belong in a concrete test class
  - We can add more test cases to **`TestSuperSlowStatPak`** that are specific to its implementation

Crash Test Dummy

- Most software systems contain a large amount of error handling code
- Sometimes, it is quite hard to create the situation that will cause the error
  - Example: Error creating a file because the file system is full
- Solution: Fake it!

```java
import java.io.File;
import java.io.IOException;

class FullFile extends File {

  public FullFile(String path) {
      super(path);
  }

  public boolean createNewFile()
                                          throws IOException {
      throw new IOException();
  }
}
```

```
public void testFileSystemFull() {
    File f = new FullFile("foo");
    try {
        saveAs(f);
        fail();
    }
    catch (IOException e)
    {}
}
```

```java
public void testFileSystemFull() {
    File f = new FullFile("foo") {
        public boolean createNewFile()
                                throws IOException {
            throw new IOException();
        }
    };
    try {
        saveAs(f);
        fail();
    }
    catch (IOException e)
    {}
}
```

Log String

- Often one needs to test that the sequence in which methods are called is correct
- Solution: Have each method append to a log string when it is called
  - Then, assert that the log string is the correct one
  - Requires changes to the implementation

Accessing private fields

- Object-oriented design guidelines often designate that certain fields should be private / protected
- This can be a problem for testing since a tester may need to assert certain conditions about private fields
- Making these fields public defeats the purpose

A solution

- Using reflection, one can actually call private methods and access private attributes!
- An example

```
class A {
  private String sayHello(String name) {
    return "Hello, " + name;
  }
}
```

```java
import java.lang.reflect.Method;

public void testPrivateMethod {
 A test = new A();
 Method[] methods =
          test.getClass().getDeclaredMethods();
 for (int i = 0; i < methods.length; ++i) {
  if (methods[i].getName().equals("sayHello")) {
   Object params[] = {"Ross"};
   methods[i].setAccessible(true);
   Object ret = methods[i].invoke(test, params);
   System.out.println(ret);
  }
 }
}
```