Inheritance and Testing

The trouble with superclasses

João Pereira ©

Inheritance and Testing

- Given a well-tested superclass and a new subclass:
  - Should you retest inherited methods?
  - Should you retest overridden 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?

- How to test abstract classes?

Inheritance

- In the early years people thought that inheritance would 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 good for a subclass”
- However, **both** claims are **wrong**
- Inherited methods cannot always be trusted:
  - Separately reliable methods can fail when used in a subclass due to interactions that occur only in that context
  - Questionable uses of inheritance can lead to subclass failures
  - **Inheritance can be abused in many ways**

Class contract

- It is a *formal* specification
- It specifies obligations
  - for the class itself
  - for the clients of the class
- A contract specifies
  1. Constraints that the caller (client) must meet before using the class
  2. Constraints that are ensured by the callee (class) when used
- The contract makes the obligations explicit instead of implicit
  - This way class and client share the same assumptions
  - It is the responsibility of the class developer
- Some languages support contracts natively, others use third-party support

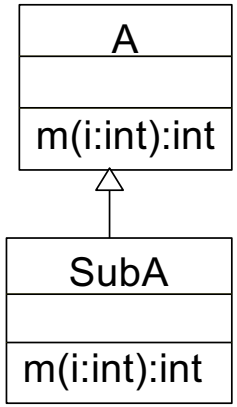
Class contract - 2

- Contracts should be specified at the Analysis & design stages
  - In UML, a language has been defined for that purpose: The Object Constraint Language (OCL)
  - JML is available to define contracts within Java programs that can be checked at run time
    - http://en.wikipedia.org/ wiki/Java_Modeling_Language)
- Three types of constraints involved in contracts:
  - *Invariant* (class)
  - *Pre-condition* (method)
  - *Post-condition* (method)

Class contract specification

- Class Invariant: Condition that must always be met by all instances of a class
  - Invariants must be true all the time, except during the execution of a method
  - A violated invariant suggests an illegal system state
- Pre-condition (method)
  - What must be true before executing an operation
- Post-condition (method)
  - Assuming the pre-condition is true, what should be true about the system state and the changes that occurred after the execution of the operation
- If the pre- and post-conditions are satisfied, then the class invariant must be preserved
  - Pre-conditions – client's responsibility
  - Class invariant and post-conditions – class creator's responsibility
- All described using a boolean expression that evaluates to true if the condition is met

Polymorphism - Liskov substitution principle interpretation



Contract of A
- Pre *m()*: i >= 0
- Post *m()*: res > 0

```
client code
public void f(A a) {
   …
   int j;
   …
   int res = a.m(j);
   …
}
```

- **Is contract of SubA independent from contract of A?**
- **No**, if we want to be able to develop flexible code!
- LSP is a requirement that subclass objects can be substituted for superclass objects without causing failures or requiring special case code in clients

Liskov substitution principle interpretation - 2

- If the client satisfies the pre-condition of the method of the superclass then the corresponding superclass postcondition and superclass invariant will hold after a message is accepted by a subclass
- Contract of subclass depends on contract of superclass
  - Otherwise, subtype relationship does not make sense

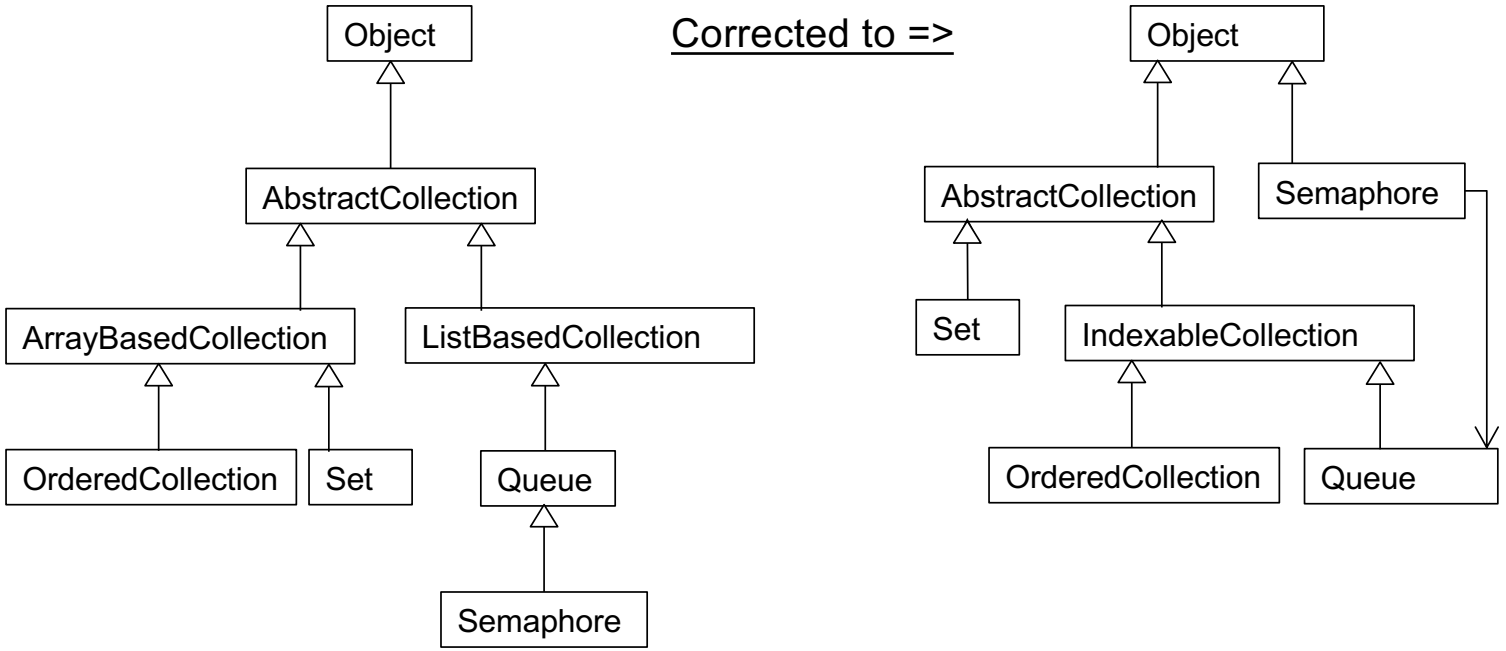
- How can the contract of subclass be **incompatible** with contract of superclass?

Inheritance rules

- Subclass may keep or weaken the precondition in the overridden methods
- Subclass may keep or strengthen the postcondition in the overridden methods
- Subclass may keep or strengthen the class invariant
- Subclass may keep or strengthen state invariants
  (subclass can accept superclass event in greater number of states than superclass – state of the superclass can be divided into substates)

Inheritance-related bugs – Convenience inheritance

- Can serve as a programming convenience
- From the StepStone Objective-C class library



- Usually have the Fat Interface bug:
  - A subclass inherits methods that are inappropriate or irrelevant

Inheritance related bugs - Square Peg in a Round Hole

- A subclass is incorrectly located in a hierarchy
- Consider *SquareWindow* as a subclass of *RectangularWindow* - a square is a special case of a rectangle.
  - Problem:
    - Suppose *RectangularWindow.resize(x,y)* is inherited by *SquareWindow*. It allows different lengths for adjacent sides, which causes *SquareWindow* to fail after it has been resized.
  - Possible Solution:

```
class Square extends Rectangle{
  public void resize(int x, int y){
    super.resize(x, x);
  }
 // …
}
```

- **Any problem?**

Not a valid solution

- Consider this client code:

```
Rectangle r;
…
r.resize(5, 4);

assert(r.getWidth() * r.getHeight() == 20);
```

- Where is the problem?

- The problem is definitely not with the client code

What went wrong?

- What is the post-condition of Rectangle.resize?
- What is the post-condition of Square.resize?
- The Liskov substitution principle was violated
  - The overridden version of *resize* broke the post-condition of its superclass version
  - If you are expecting a rectangle, you can not accept a square
- Isn't a square a rectangle?
  - Yes, but not when it pertains to its behavior

Inheritance Related Bugs

- Incorrect initialization
  - Superclass initialization is omitted or incorrect
  - Deep hierarchies may lead to initialization bugs
- Inadvertent binding
  - Incorrect bindings due to misunderstood name-scoping rules.
    - e.g., same instance variable name used in superclass and subclass
  - Worst in multiple inheritance
- Missing override
  - A subclass-specific implementation is omitted
  - e.g., **equals** and **hashCode** methods
- Naked Access
  - Direct access to superclass fields from the subclass code
  - Related faults
    - Subclass bugs or side effects can cause failure in superclass methods
    - Changes to the superclass implementation can create subclass bugs
- …

Example 1 – No modification of inherited method

- Need to retest inherited method?

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

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

- If **m2()** has a bug and breaks invariant, **m** is incorrect in the context of **B** even though it is correct in **A**
  - Therefore, **m()** should be tested in **B**
    - **Test method sequence m2()m()**

Example 2 – Indirectly modification of inherited method

- Subclass overrides a method of superclass
  - May affect other inherited methods

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

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

- Do I need to test *m()* in the context of the subclass?
  - Yes
- Is the test suite of *m()* made for A enough to test it in B?
  - Maybe. Probably not.

Testing of inheritance

- Inherited methods should be retested in the context of a subclass
  - At least at class scope
- Test cases developed for a method **m** defined in class **X** are not necessarily sufficient for retesting **m** in subclasses of **X. Why (When) ?**
  - m calls other methods that are overridden in subclass
  - m is overridden in subclass
- 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)

Impact 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
- However, can reuse superclass test cases

Flattening class scope

- Flattened class makes all inherited features explicit
  - Necessary to find inheritance bugs
- Example:
  - Class A: base class
    - Class scope: a1, a2, a3
    - Flattened scope: same
  - Subclass B
    - Class scope: B.a2, b4, b5, b6
    - Flattened scope:
      - **a1**, B.a2, b4, b5 and b6
  - Subclass C
    - C.a2/b5 – overrides B.a2/b5
    - c7 – specialization
    - Class scope: C.a2, C.b5, c7
    - Flattened scope:
      - **a1**, **b4**, C.a2, C.b5, c7

```
public class A {
    public void a1() {…}
    public void a2() {…}
    private void a3() {…}
}

public class B extends A {
    public void a2() {…}
    public void b4() {…}
    public void b5() {…}
    private void b6() {…}
}

public class C extends B {
    public void a2() {…}
    public void b5() {…}
    public void c7() {…}
}
```

Goals for testing a flattened class

1. Reuse superclass test suites as much as possible

2. At method-scope testing:
    1. Retest superclass methods only to the extent necessary to gain confidence that inherited features work correctly in subclass
    2. Extend class test suites with test cases that are effective for the kind of bugs that accompany inheritance

Decision rules for flattened class testing

	Type of Subclass Method		
	Extension	**Overriding**	**Specialization**
Should this method be included in the flattened class scope testing?	Yes[1]	Yes[1]	Yes[1]
Prudent extent of retesting of this method in subclass	Minimal[2,3]	Full	Full
Reuse superclass method tests	Yes	Probably[4]	N/A
Develop new subclass method tests	Maybe[3]	Yes	Yes

1. Individual method tests are not sufficient to test method interactions
2. Method scope tests do not need to be repeated.
3. New tests are needed when inherited features use locally defined features
4. Superclass tests are reusable to the extent that the superclass and subclasses comply with the LSP.

Decision rules for flattened class testing - 2

- Additional test cases in subclass context for:
  - New and overridden methods (method scope)
    - Maybe for inherited methods
  - To exercise interactions between subclass methods (class scope)
  - To exercise interactions between superclass and subclass methods (flattened class scope)
- If the class hierarchy under test is designed to conform with the LSP then re-running all superclass test suites is an effective means to check LSP conformance
- If a subclass is not intended to be LSP-conforming, then the value of its superclass test suite is less clear.

Flattened class scope test design patterns

- How to run superclass test suites for subclasses?
  - Important only if the class hierarchy under test is designed to conform with the LSP

- Solution: Inherit class test suites
  - Ease rerunning a superclass test suite on a subclass object
  - Partial solution for the additional testing load imposed by inheritance

Inheritance-based Patterns

- All test patterns can be applied to subclasses
  - Apply best model for each method and class
  - Reuse superclass test cases for inherited and overridden methods
    - May need to add test cases
- Is this enough?
- For testing at flattened class scope:
  - Polymorphic Server Test – Design a reusable test suite for a LSP-compliant polymorphic server hierarchy
  - Modal Hierarchy Test – Design a test suite for a hierarchy of modal classes
  - Non-modal classes – Apply non-modal class test at flattened class scope

Polymorphic Server Test

- Intent
  - Design a test suite at flattened class scope that verifies LSP compliance for a polymorphic server hierarchy
- Context
  - The hierarchy under test contains overridden methods and is non-modal.
    - For modal classes apply **Modal Hierarchy test**
  - An overriding method must respect the constraints of the overridden method to provide a consistent and correct response
  - An attempt to verify a server in the context of every client is an *impossible* task.
  - **Goal:** How can we exercise the overridden methods in a polymorphic hierarchy with a driver to verify that a correct and consistent response is produced for all bindings for all clients?

Fault model

- Any of the fault classes described earlier
- Faults related to an overridden method
- Faults in clients are not the target of this pattern

Strategy: Test model

- Model:
  - Contract of polymorphic methods

- Strategy:
  - Build a test suite for each class in the hierarchy that checks for conformance with **LSP**
  - Each polymorphic method should be exercised in its defining class and all subclasses that inherit it
  - Test suite focus on contract of polymorphic methods
  - Must be possible to reuse it with instances of subclasses
    - Parametrize it with a reference to an object of the CUT
  - If subclass does not conform to LSP, the base class test suite may reveal this fault

Example

Contract of Account

deposit(val) -> pre: 0 < val <= 100

…

```
Class TestAccount {
  protected Account a;
  @BeforeMethod protected void setUp() {
    a = new Account();
  }

  @Test
  public final void testMaxDeposit() {
    a.deposit(100);
    assertEquals(a.getBalance(), 100);
  }
  ...
}
```

Contract of SavingsAccount

setInterest(interest) pre: 0 < interest <= 0.1

applyInterest() -> post: bal = bal*(1 + interest)

…

```
Class TestSavingAccount extends
                                TestAccount {
  protected SavingsAccount sa;
  @BeforeMethod protected void setUp() {
    sa = new SavingsAccount();
    a = sa;
  }

  @Test public final void testMaxInterest() {
    sa.deposit(100);
    sa.setInterest(0.1);
    sa.applyInterest();
    assertEquals(sa.getBalance(), 110);
  }
  ...
}
```

Entry and Exit Criteria

- Entry Criteria
  - An alpha-omega cycle is developed from the subclass up.
  - All superclasses pass their test suites.
- Exit Criteria
  - Every superclass method is exercised in the context of every subclass once

Consequences

- Establishes high confidence in the reusability of polymorphic servers
- Reveal faults that prevent LSP conformance and may reveal other inheritance-related faults
- It requires the design and implementation of test drivers that parallel the polymorphic structure of the CUT
- Test suites for method, class, and flattened class scope responsibilities should be implemented in separate test suites
- Input/output and sequential behavior faults are not the target of this pattern
- Clients should use Polymorphic Message Test

Modal Hierarchy Test

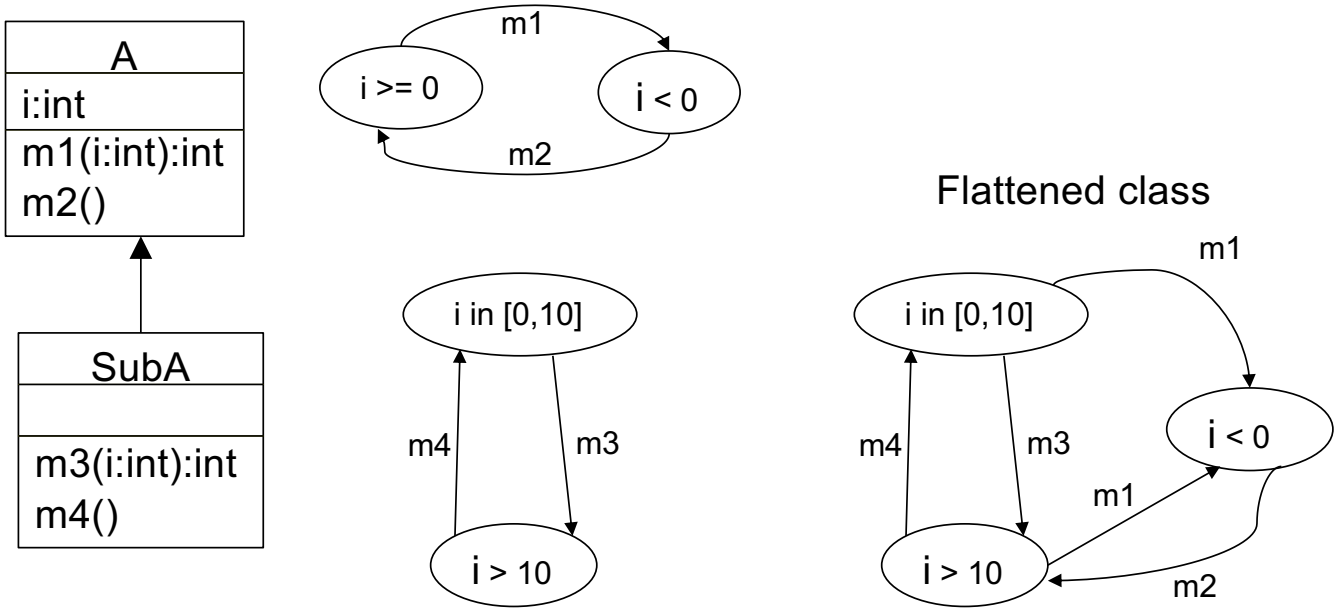
- Intent
  - Design a flattened class scope test suite for a hierarchy of modal classes
- Context
  - CUT is a subclass that
    - Inherits from a modal superclass
    - Implements its own modal behavior, and/or
    - Extends the superclass's modal behavior
  - It must conform not only to its own state model, but also to the superclass state model
    - How to check the flattened state model?

Fault Model

- The state-based faults and failures that occur at flattened scope include all those that can occur at class scope
- A subclass can introduce state-based failures if the superclass state control requirements are not observed
  - A valid superclass transition is rejected.
  - An illegal superclass message transition is accepted.
  - An incorrect action is produced on a superclass transition.
- In an LSP-compliant modal subclass, all behavior of the superclass must be respected
- Classes can be modal without being LSP-compliant

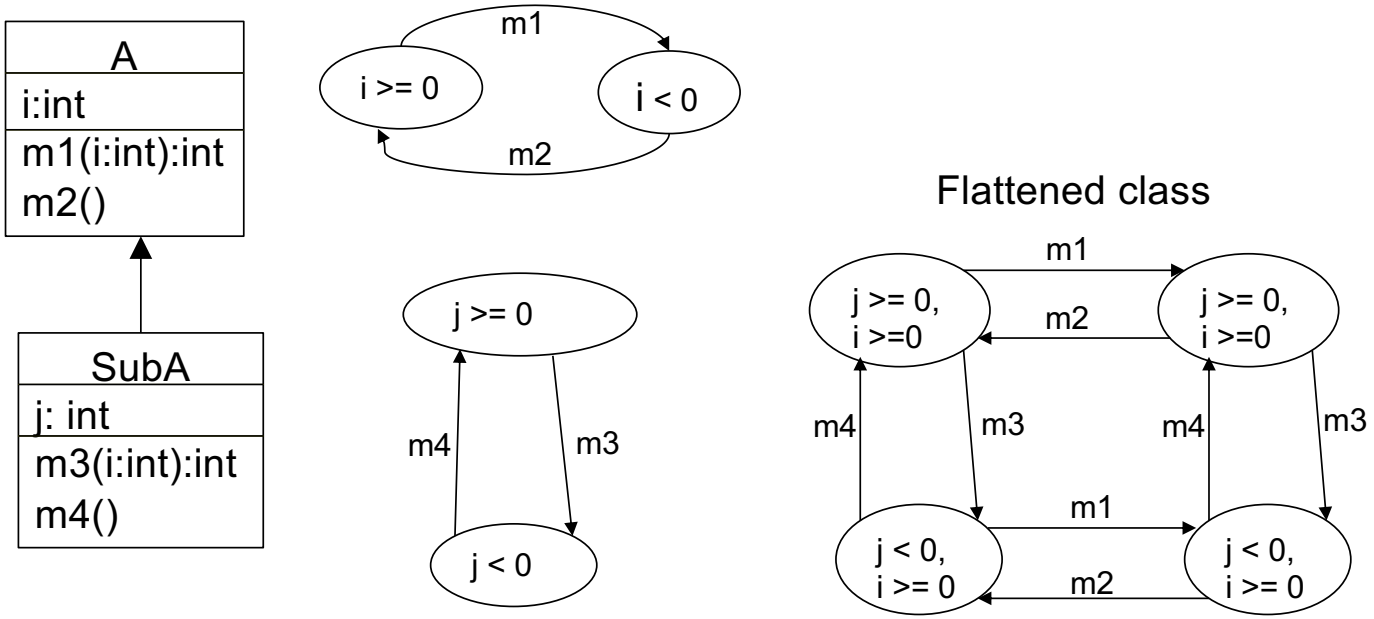
Partition the state of superclass

- The subclass can partition one or more states of superclass
  - Some states of subclass are part of a state of the superclass



Extend state model

- The subclass defines its own state behavior orthogonal to the state behavior of the superclass



Two more cases

- No modal behavior defined in subclass

- The subclass defines its own modal behavior
  - **But does not preserve** the modal behavior of the superclass
  - Can change transitions by changing the behavior of overridden methods
  - Can define sub-states that break invariant of superclass states

Testing strategy - 1

- Test model is the same as that employed for **Modal Class Test**
- **However**, the state machine must reflect the states and events of the flattened class
- Develop test suite for exercise the behavior specified by the state machine of the *flattened* class
- If subclass is LSP-compliant with superclass then run superclass test suite as regression test
- If not LSP-compliant, then

Entry and Exit criteria

- Entry Criteria
  - An alpha-omega cycle should be developed from the subclass up.
  - All superclasses pass their test suites.
- Exit Criteria
  - The same as those for **Modal Class Test**, but at flattened scope.
- Consequences
  - Requires development of a flattened scope state model
  - More those shown for **Modal Class Test**

Extra - Useful simple test patterns

- There are several situations testers face often for which simple established solutions exist
- We will discuss three of these **test patterns**
  - Log String
  - Crash Test Dummy
  - Mock objects
- And how to access private members

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
  - **Drawback:**
    - **Requires** changes to the implementation

Mock Objects

- How do you test a class that uses an expensive or complicated resource?
  - Databases take a long time to start
  - Or server methods with a long execution time
- Solution: Have a **mock object** that acts like a database

```
Database db = new MockDatabase();
db.query("select ....");
db.result("Fake Result");
```

Mock objects

- If the MockDatabase does not get the query it expects, it throws an exception
- If the query is correct, it returns the provided result
- **Caveat**: the mock object needs to behave exactly like the real one

Accessing private fields

- Object-oriented design guidelines often designate that most 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;
  }
}
```

```
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[] = {"Richard"};
   methods[i].setAccessible(true);
   Object ret = methods[i].invoke(test, params);
   System.out.println(ret);
  }
 }
}
```

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 writing to a file because the file system is full
- Solution: Fake it!
  - Not always possible. Depends on design

How to fake system full?

Very hard

```
class Example {
  private FileOutputSystem fos;
  // ...
  public Example(String path) {
    this.fos = new FileOutputStream(path);
  }

  public boolean writeData()
              throws IOException {
    try {
      this.fos.write(_data);
    } catch (IOException ioe) { ... }
  }
}
```

Very easy

```
class Example {
  private FileOutputSystem fos;
  // ...
  public Example(FileOutputSystem fos) {
    this.fos = fos;
  }

  public boolean writeData()
              throws IOException {
    try {
      this.fos.write(_data);
    } catch (IOException ioe) { ... }
  }
}
```

Solution

- Define an *OutputFileSystem* that simulates the file system is full

```
class FullSystem extends FileOutputSystem {

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

  public boolean write(String str)
                    throws IOException {
     throw new IOException();
  }
  // similar for remaining methods
}
```

```
@Test public void testFileSystemFull() {
   Example f = new Example(new FullSystem("foo"));
   try {
      // ...
      f.writeData(str);
      …
   } catch (IOException e) {
      fail();
   }
}
```

Abstract Class Test

- Intent
  - Develop a test implementation of a test suite for an abstract class interface.
- Context
  - Abstract class cannot be instantiated
  - Abstract class may have abstract methods
  - Cannot be tested as written

Fault Model

- Design errors:
  - Incomplete and/or inconsistent identification of common responsibilities
    - Concrete methods
  - Invalid and/or incomplete representation of responsibilities in the interface
    - Abstract methods
  - Definition of fields, which should be delayed (defined later)

Strategy

1. Develop a subclass, which implements all of the superclass's abstract methods
   - Implementing a subclass is part of the test
2. Apply method and class scope test pattern for abstract class
   - Develop test cases for abstract methods
   - Develop test cases for concrete methods
3. Test subclass as flattened
   - May need to develop additional test cases for implemented abstract methods in subclass

- Separated test suites for abstract and subclass

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();
}
```

Test suite for abstract class

- Tests defining the functionality of the interface belong in the abstract test class

```
public abstract TestStatPak extends TestCase {
  private StatPak statPak;
  // final to prevent overriding
  public final setUp() throws Exception {
    statPak = createStatPak();
    assertNotNull(statPak);
  }
  // Factory Method.
  public abstract StatPak createStatPak();

  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 should be 3.0", 3.0,statPak.getMean());
  }
  public final void testStdDev() { ...}
  ...
}
```

Test suite for concrete class

- Write a simple concrete subclass
- Test this class at flattened scope
- Tests specific to an implementation belong in a concrete test class
  - We can add more test cases to **TestSuperSlowStatPak** that are specific to the subclass implementation
  - Or to improve coverage of superclass's test suite

```
public class TestSuperSlowStatPak extends TestStatPak {
   public StatPak createStatPak() {
      return new SuperSlowStatPak();
   }
   …
}
```

Entry and Exit Criteria

- Entry Criteria
  - Abstract class can be tested after successfull compilation
- Exit Criteria
  - Concrete test should achieve a coverage recommended for a pattern being used, minimally branch coverage within methods

Consequences

- Test subclass creation and its test suite can lead to detection of design mistakes or reusability problems
- Test suite is useful as a documentation for users of the abstract class
- A test suite that can be reused for future as-yet-unidentified descendants
  - Especially useful for writers of APIs.