Software Testing

Making the Right Software, and Making Software Right

QUIZ

- Join YACR

Software Failure

There are three ways to describe problems with the software system. A *failu*

A: an unacceptable behaviour by the system.

B: a flaw in any aspect of the system that contributes to a larger problem with the system.

C: a slip up or inappropriate decision by a software developer that leads the to introduction of a defect.

D: none of the above

Software Failure

There are three ways to describe problems with the software system. A *failu*

A: **an unacceptable behaviour of the system.**

B: a flaw in any aspect of the system that contributes to a larger problem with the system.

C: a slip up or inappropriate decision by a software developer that leads the to introduction of a defect.

D: none of the above

Software Testing

A software engineer, working in a hurry, accidentally added a duplicated

This is an example of:

A: Defect

B: Failure

C: Fault

D: Error

Software Testing

A software engineer, working in a hurry, accidentally added a duplicated

This is an example of:

A: Defect

B: Failure

C: Fault

D: **Error**

Why is Testing Important?

- All software development models include testing
- Small or large projects include better code
- More sophisticated techniques require more efficiency

Software Testing

- Testing can only show the presence of errors, not their absence.

How Much Testing?

Validation versus Verification

- Validation – Build the Right Product
- Verificatio

Types of Testing

- Black/Opaque Box Testing
  - When yo urce code
  - Test Input nctionality
- White/Clear/Glass Box Testi
  - When you have access to the source code, and design test cases to cover paths

Testing Coverage

- Black Box – you can only test based on inputs and expected outputs
- Glass Box – Coverage can't ing all possible paths, and covering all possible edges, and cover all nodes
- In both cases, you should be able to design a set of tests that put a system through it's paces. Check baseline, valid, and invalid input as a standard approach to testing.

Equivalence Classes: Basic Approach

- Valid, invalid

Equivalence Classes

- Another way to think about validity. Divide all possible inputs into groups that should be treated the same way.

- For Example: Valid entation.

Equivalence Class	Range of Values
Invalid – Larger	13 to 2**31 (Max int)
Valid	1 to 12
Invalid – Smaller	0 to -2**31 (Min int)

Equivalence Classes

- Telephone Number Validation

- Invalid – Null Inp
- Invalid – Too Few Digits
- Invalid – Correct Number of Digits without leading 0
- Valid – Correct Number of Digits with leading 0
- Invalid – Too Many Digits

JUnit

- JUnit is a testing tool, it is already incorporated into Eclipse
  - Create a JUnit test case – JUnit Test Case
- Run Test Cases – Right Click on Project -> Run As -> JUnit Test Case

Test Assertions

- fail
- assertTrue
- assertFalse
- assertEquals
- assertNull
- assertNotNull
- assertSame
- assertNotSame

Test Annotations

- @Test
- @Test(expec
- @Test(timeout=)
- @Before
- @After
- @BeforeClass
- @AfterClass
- @Ignore

Break

A JUnit test class


```
import org.junit.*;
import static org.junit.Assert.*;

public class MyFirstSimpleTest {
    ...

    @Test
    public void myFirstSimpleTest() {
        ...
    }
}
```

- A method with `@Test` is flagged as a JUnit test case.
  - All `@Test` methods run when JUnit runs your test class.

JUnit assertion methods

`assertTrue(`**test**`)`	fails if the boolean test is `false`
`assertFalse(`**test**`)`	fails if the boolean test is `true`
`assertEquals(`**expect**	s are not equal
`assertSame(`**expected**	s are not the same (by ==)
`assertNotSame(`**expected**`,` **actual**`)`	f *are* the same (by ==)
`assertNull(`**value**`)`	f lue is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

Each method can also be passed a string to display if it fails:

e.g. `assertEquals(`**"message", expected, actual**`)`

JUnit exercise

Given a `Date` class with the following methods:

```
public Date(int year, int month, int day)
public Date()                          // today
public int ge                       Year       ()
public void a                    dvanceDay    days
public int da
public String dayOfWeek()                  g. "Sunday"
public boolean equals(Objec
public boolean isLeapYear()
public void nextDay()                  // advances by 1 day
public String toString()
```


- Come up with unit tests to check the following:
  - That no `Date` object can ever get into an invalid state.
  - That the `addDays` method works properly.
    - It should be efficient enough to add 1,000,000 days in a call.

A common mistake

```
public class DateTest {
    @Test
     public void test1() {
          Date d = new Date(2050, 2, 15);
          d.addDays(4);
          asse                                  (), 2050);
          asse                                  (), 2);
          asse                                  19);
     }

     @Test
     public void test2() {
          Date d = new Date(2050, 2, 15);
          d.addDays(14);
          assertEquals(d.getYear(), 2050);
          assertEquals(d.getMonth(), 3);
          assertEquals(d.getDay(), 1);
     }
}
```

Expectation on the left

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear());  // expected
        assertEqual                       // value should
        assertEqual                       // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    }   // test cases should usually have messages explaining
}       // what is being checked, for better failure output
```

Objects are good to simplify tests

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expecte
        assertEquals                    n expected answer
    }                                   t to minimize tests

                                                    ust have toString
    @Test                                           uals methods)
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }
}
```

Use informative name

```
public class DateTest {
    @Test
    public void test addDays_withinSameMonth_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(4);
        Date expected = new Date(2050, 2, 19);
cted,   assertEquals                                    actual);
    }
    // give test cas                          scriptive names

    @Test
    public void test addDays_wrapTo                 () {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
    // give descriptive names to expected/actual values
}
```

Variable messages? … not so good

```
public class DateTest {
    @Test
    public void testDay_1() {
        Date actual = new Date(2, 16, 15);
        actual.a
        Date expected = new D                2, 16);
        assertEquals(
            "should have gott                ected + "\n" +
            " but instead got " + actual\n",
            expected, actual);
    }
    ...
}
```

Tests with a timeout

```
@Test(timeout = 5000)
public void name() { ... }
```


- The above metho … it does n't finish running within 5000 ms


```
private static final int UI = 2000;
...

@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Times out / fails after 2000 ms

Pervasive timeouts

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_withinSameMonth_1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
 s", expected, d);
    }


    @Test(timeout = DEFAULT_TIM
    public void test_addDays_wr                nth_2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }

    // almost every test should have a timeout so it can't
    // lead to an infinite loop; good to set a default, too
    private static final int DEFAULT_TIMEOUT = 2000;
}
```

Testing for exceptions

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```


- Will pass if it *does*
  - If the excepti
  - Use this to test for expected erro


```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);   // should fail
}
```

Setup and teardown

```
@Before
public void name() { ... }
@After
public void name() { ... }
```


- methods to run before/after each test method is called

```
@BeforeClass
public static void name() { ... }
@AfterClass
public static void name() { ... }
```

- methods to run once before/after the entire test class runs

PI Questions

- Join YACRS Session 1262

PI 7.1 @Before example

```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class SimpleTest {
    private Collection<Object> collection;

    @Before
    public void setUp() {
        collection = n
    }

    @Test
    public void testEmptyCollection(){ \\ f2
        assertTrue(collection.isEmpty());
    }

    @Test
    public void testOneItemCollection() { \\ f3
        collection.add("itemA");
        assertEquals(1, collection.size());
    }
}
```

What is the execution order?

- A: f1 > f2 > f3
- : f1 > f2 > f1 > f3
- : f3 > f1 > f2 > f1
- D: f2 > f1 > f3 > f1

PI 7.1 @Before example

```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class SimpleTest {
    private Collection<Object> collection;

    @Before
    public void setUp() {
        collection = n
    }

    @Test
    public void testEmptyCollection(){ \\ f2
        assertTrue(collection.isEmpty());
    }

    @Test
    public void testOneItemCollection() { \\ f3
        collection.add("itemA");
        assertEquals(1, collection.size());
    }
}
```

What is the execution order?

- A: f1 > f2 > f3
- **: f1 > f2 > f1 > f3**
- f3 > f1 > f2 > f1
- D: f2 > f1 > f3 > f1

@Before and @After

```
import org.junit.*;
import static org.junit.Assert.*;
import java.io.*;
public class OutputTest {
  private File output;

  @Before
  public void creat
    output = new File(...);
  }

  @After
  public void deleteOutputFile() {
    output.delete();
  }

  @Test
  public void testSomethingWithFile() {
    ...
  }
```

Execution order:

- createOutputFile()
- testSomethingWithFile()
- deleteOutputFile()

Execution order

```
public class SimpleTest {
 private Collection collection;
  @BeforeClass
  public static void oneTimeSetUp() { //f1
    // one-time initialization code
  }

  @AfterClass
  public static void on
    // one-time cleanup
  }

  @Before
  public void setUp() { //f3
    collection = new ArrayList();
  }
  @After
  public void tearDown() { //f4
    collection.clear();
  }
 ..
```

oneTimeSetUp()
setUp()
testEmptyCollection()
tearDown()
setUp()
testOneItemCollection()
tearDown()
oneTimeTearDown()


```
@Test
public void testEmptyCollection(){//f5
assertTrue(collection.isEmpty());
}
@Test
public void testOneItemCollection(){//f6
collection.add("itemA");
assertEquals(1, collection.size());
}
}
```

JUnit exercise

Given our `Date` class seen previously:

- `public Date(int year, int month, int day)`
- `public Date()  // today`
- `public int getDay(), getMonth(), getYear()`
- `public void addDays(int days)  // advances by days`
- `public int day`
- `public String  e.g. "3rd May"`
- `public boolean equals(Obje`
- `public boolean isLeapYear(`
- `public void nextDay()  dvances by 1 day`
- `public String toString()`

- Come up with unit tests to check the following:
  - That no `Date` object can ever get into an invalid state.
  - That the `addDays` method works properly.
    - It should be efficient enough to add 1,000,000 days in a call.

A bit cluttered code

```
public class DateTest {
    @Test
    public void test addDays_withinSameMonth_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(4);
        Date expected = new Date(2050, 2, 19);
cted,   assertEquals                                    actual);
    }
    // give test cas                          scriptive names

    @Test
    public void test addDays_wrapTo                 () {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
    // give descriptive names to expected/actual values
}
```

Squashing redundancy

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_withinSameMonth_1() {
        addHelper(2050, 2, 15, +4, 2050, 2, 19);
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addD
;       addHelper(20
    }

    // use lots of helpers to make                    s extremely short
    private void addHelper(int y1,                    d1, int add,
                           int y2,                    d2) {
        Date act = new Date(y, m, d);
        actual.addDays(add);
        Date exp = new Date(y2, m2, d2);
        assertEquals("after +" + add + " days", exp, act);
    }

    // can also use "parameterized tests" in some frameworks
    ...
```

Organizing Tests

Option 1:

```
src
    com
        xyz
            SomeClass.ja
            SomeClassTes
```


Option2:

```
src
    com
        xyz
            SomeClass.java
test
    com
        xyz
            SomeClassTest.java
```

Test-driven development

- Unit tests can be written after, during, or even *before* coding.
  - **test-driven development**: Write tests, *then* write code to pass them.

- Imagine that we'd li `ractWeeks` to our `Date` class, that shifts this the given number of weeks.

- Write code to test this method *before* it has been written.
  - Then once we do implement the method, we'll know if it works.

P7. 2 Test First Development

How many of the following statements are true about test first development?

Test first development reduces testing by only using tests are the beginning of developme

Test cases can be used t table form.

Test first development prevents the occurrenc

A: 0

B: 1

C: 2

D: 3

Test First Development

How many of the following statements are true about test first development?

Test first development reduces testing by only using tests are the beginning of developme

Test cases can be used to table form.

Test first development prevents the occurrenc

A: 0

B: 1

C: 2

D: 3

Testing Credit Cards

You must develop a set of equivalence classes for a credit card processing system.

The only credit cards accepted are MasterCard and Visa. Card numbers must be 16 digits long. MasterCards must start with a 5, and Visa must start with a 4.

How many equivalenc t for valid credit card numb

A: 5

B: 6

C: 7

D: 8

Equivalence Classes

- Too Few Digits – Null Input
- Too Few Digits –
- Correct Digits wi
- Correct Digits with Valid Master
- Correct Digits with Valid Visa Format
- Too Many Digits - >16 Digits

Equivalence Classes: Testing Boundaries

Errors often happen at the boundaries of equivalence classes, so choosing a meaningful boundary and testing it thoroughly is important.

In the credit card example, how would you test the boundary cases?

A: By testing the di digit length for invalid input.

B: By testing the Max Int and Max Int -1 digit lengths for invalid input.

C: By testing invalid leading digits 3 and 6

D: Something else

Equivalence Classes: Testing Boundaries

Errors often happen at the boundaries of equivalence classes, so choosing a meaningful boundary and testing it thoroughly is important.

In the credit card example, how would you test the boundary cases?

A: By testing the differ ngth for invalid input. *There mi here* **These boundaries are in general we would focus on boundaries b nd invalid.**

B: By testing the Max Int and Max Int -1 digit lengths for invalid input. **We would expect that these values could reasonably behave the same and would probably be a waste of time to test both**

C: By testing invalid leading digits 3 and 6 **These are a good boundary case for valid integers since we might uncover an arithmetic and off by one error in our validation.**

D: Something else **Another boundary case is checking that 15 and 17 digits both come back as invalid**

Next Week

- Lab review, and more on testing