# Software Testing

Making the Right Software, and Making Software Right

# QUIZ

- Join YACRS session 1268

# Software Failure



There are three ways to describe problems with the software system. A *failure* is:





A: an unacceptable behaviour exhibited 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 *failure* is:

A: **an unacceptable behaviour exhibited 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 Testing



A software engineer, working in a hurry, accidentally added a duplicated line of code.



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 line of code.



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 scale it will help you produce better code
- More sophisticated techniques mean more efficiency

# Software Testing

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

# How Much Testing?



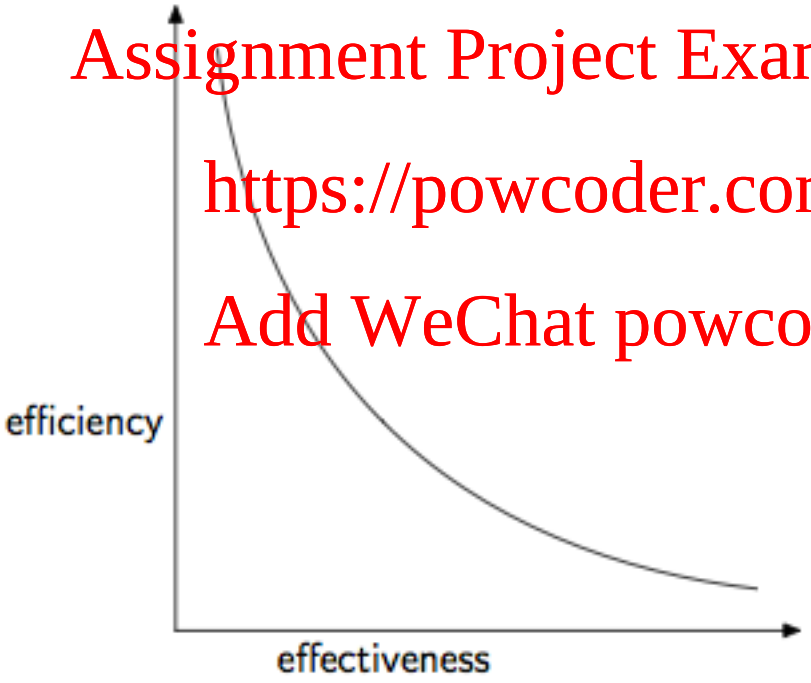


Figure: effectiveness vs. efficiency in testing

# Validation versus Verification

- Validation = Build the Right Product
- Verification = Build the Product Right

# Types of Testing



- Black/Opaque Box Testing
  - When you don't have access to the source code
  - Test Inputs versus Outputs, checking functionality
- White/Clear/Glass Box Testing
  - 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 include covering 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: Validating a month's integer representation.



| 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 Input
- 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 – File -> New -> 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(expected=Exception)
- @Test(timeout= )
- @Before
- @After
- @BeforeClass
- @AfterClass
- @Ignore

# Break

# A JUnit test class



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

public class SimpleTest {
    ...

    @Test
    public void myFirstSimpleTest() { // a test case method
        ...
    }
}
```

- 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(`**`expected, actual`**`)` | fails if the values are not equal |
| `assertSame(`**`expected, actual`**`)` | fails if the values are not the same (by ==) |
| `assertNotSame(`**`expected, actual`**`)` | fails if the values *are* the same (by ==) |
| `assertNull(`**`value`**`)` | fails if the given value 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 getDay(), getMonth(), getYear()
public void addDays(int days)          // advances by days
public int daysInMonth()
public String dayOfWeek()              // e.g. "Sunday"
public boolean equals(Object o)
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);
          assertEquals(d.getYear(), 2050);
          assertEquals(d.getMonth(), 2);
          assertEquals(d.getDay(), 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
        assertEquals(2, d.getMonth());     // value should
        assertEquals(19, d.getDay());      // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        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 expected = new Date(2050, 2, 19);
        assertEquals(expected, d); // use an expected answer
    }                              // object to minimize tests

                                   // (Date must have toString
    @Test                          //  and equals 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);
        assertEquals("date after +4 days", expected, actual);
    }
    // give test case methods really long descriptive names

    @Test
    public void test_addDays_wrapToNextMonth_2() {
        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 test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals(
            "should have gotten " + expected + "\n" +
            " but instead got " + actual\n",
            expected, actual);
    }
    ...
}
```

# Tests with a timeout

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



- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 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);
        assertEquals("date after +4 days", expected, d);
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void test_addDays_wrapToNextMonth_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* throw the given exception
  - If the exception is *not* thrown, the test fails.
  - Use this to test for expected errors



```
@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 case 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() { \\ f1
        collection = new ArrayList<Object>();
    }

    @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
- B: f1 > f2 > f1 > f3
- C: 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(){ \\ f1
        collection = new ArrayList<Object>();
    }

    @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
- **B: f1 > f2 > f1 > f3**
- C: 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 createOutputFile() {
    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 oneTimeTearDown() { //f2
    // one-time cleanup code
  }

  @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 daysInMonth()`
- `public String dayOfWeek()` `// e.g. "Sunday"`
- `public boolean equals(Object o)`
- `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 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);
        assertEquals("date after +4 days", expected, actual);
    }
    // give test case methods really long descriptive names

    @Test
    public void test addDays_wrapToNextMonth_2() {
        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 addDays_wrapToNextMonth_2() {
        addHelper(2050, 2, 15, +14, 2050, 3, 1);
    }

    // use lots of helpers to make actual tests extremely short
    private void addHelper(int y, int m, int d, int add,
                           int y2, int m2, int 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.java
      SomeClassTest.java

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 like to add a method `subtractWeeks` to our `Date` class, that shifts this `Date` backward in time by 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 development



Test cases can be used to re-state the requirements in a testable form.



Test first development prevents the occurrence of failures.

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 development



**Test cases can be used to re-state the requirements in a testable form.**



Test first development prevents the occurrence of failures.

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 equivalence classes would you use to test for valid credit card numbers?



A: 5

B: 6

C: 7

D: 8

# Equivalence Classes

- Too Few Digits – Null Input
- Too Few Digits – 1-15 digits
- Correct Digits without Valid Leading Digits
- Correct Digits with Valid MasterCard Format
- 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 difference between 0 and 1 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 difference between 0 and 1 digit length for invalid input. *There might be null pointer errors hiding here.* **These boundaries are between two invalid inputs, and in general we would focus on boundaries between valid and 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