CSE 403
Lecture 12

Effective Unit Testing

Reading:

*The Art of Unit Testing*, Ch. 7 (Osherove)

slides created by Marty Stepp
http://www.cs.washington.edu/403/
• **software reliability**: Probability that a software system will not cause failure under specified conditions.
  – measured by uptime, MTTF (mean time till failure), crash data

• **bugs** are inevitable in any complex software system
  – industry estimates: 10-50 bugs per 1000 lines of code
  – a bug can be visible or can hide in your code until much later

• **testing**: A systematic attempt to reveal errors.
  – failed test: an error was demonstrated
  – passed test: no error was found (for this particular situation)
Difficulties of testing

• perception by some developers and managers
  – testing is seen as a novice's job
  – assigned to the least experienced team members
  – done as an afterthought (if at all)

• limitations of what testing can show you
  – it is impossible to completely test a system
  – testing does not always directly reveal the actual bugs in the code
  – testing does not show absence of errors in software
Faults and errors

• **error**: incorrect software behavior
  – *example*: Message box said, "Welcome, null!"

• **fault**: mechanical or algorithmic cause of error (bug)
  – *example*: Account name field is not set properly.
  – Requirements specify desired behavior; if the system deviates from that, it has a fault.
• **fault avoidance**: Prevent errors before system is released.
  – reviews, inspections, walkthroughs, development methodologies, configuration management

• **fault tolerance**: When system can recover by itself.
  – rollbacks, redundancy, mirroring

• **fault detection**: Find faults without recovering from them.
  – debugging, **testing**
Some kinds of testing

- **unit testing**: Looks for errors in subsystems in isolation.

- **integration testing**: find errors when connecting subsystems
  - bottom-up: integrate upward into double, triple, quadruple test
  - top-down: test UI first, then add layers to replace stubs
  - stub/mock: an incomplete object/subsystem in masquerade

- **system testing**: test entire system behavior as a whole, with respect to scenarios and requirements
  - functional testing: test whether system meets requirements
  - performance, load, stress testing
  - acceptance, usability, installation, beta testing
**Unit testing**

- **unit testing**: Looks for errors in subsystems in isolation.
  - generally a "subsystem" means a class or object

- **benefits:**
  1. reduces number of things to test
  2. easier to find faults when errors occur
  3. can test many components in parallel

- In principle, test all objects.
  - Because of time, test important ones involved in use cases.
JUnit and Eclipse

- Adding JUnit to an Eclipse project:
  - click **Project → Properties → Add External JARs... → eclipse folder / plugins / org.junit_x.x.x / junit.jar**

- Create a test case
  - click **File → New → JUnit Test Case**
  - or right-click a file and choose **New Test**
  - Eclipse can create stubs of method tests for you
JUnit assertion methods

- `assertTrue`  \((message, \text{test})\)  \(^{\text{italic} = \text{optional}}\)
- `assertFalse`  \((message, \text{test})\)
- `assertEquals`  \((message, \text{expected}, \text{actual})\)
- `assertNotEquals`  \((message, \text{expected}, \text{actual})\)
- `assertSame`  \((message, \text{expected}, \text{actual})\)
- `assertNotSame`  \((message, \text{expected}, \text{actual})\)
  \(\text{compares with} \equiv\)
- `assertNull`  \((message, \text{obj})\)
- `assertNotNull`  \((message, \text{obj})\)
- `fail`  \((message)\)
  \(- \text{causes the test to immediately fail}\)
  \(\text{(why no pass method?)}\)
require 'test/unit'

class name < Test::Unit::TestCase
  def setup
    ...
  end

  def teardown
    ...
  end

  def name      # a test case
    ...
    assert(condition,  message)
  end
end
Ruby assertions

- assert(boolean, [msg]) - ensures the object/expression is true
- assert_equal(obj1, obj2, [msg]) - ensures obj1 obj2 is true
- assert_not_equal(obj1, obj2, [msg]) - ensures obj1 obj2 is false
- assert_same(obj1, obj2, [msg]) - ensures obj1.equal?(obj2) is true
- assert_not_same(obj1, obj2, [msg]) - ensures obj1.equal?(obj2) is false
- assert_nil(obj, [msg]) - ensures obj.nil? is true
- assert_not_nil(obj, [msg]) - ensures obj.nil? is false
- assert_match(regexp, string, [msg]) - ensures a string matches the regular expression
- assert_no_match(regexp, string, [msg]) - ensures string doesn't match regex
- assert_in_delta(expecting, actual, delta, [msg]) - ensures numbers are within delta
- assert_throws(symbol, [msg]) { block } - ensures a block throws the symbol
- assert_raises(exceptions) { block } - ensures block raises an exception
- assert_nothing_raised(exceptions) { block } - a block doesn’t raise the exceptions
- assert_instance_of(class, obj, [msg]) - ensures obj is the class type
- assert_kind_of(class, obj, [msg]) - ensures obj is or descends from class
- assert_respond_to(obj, symbol, [msg]) - ensures obj has a method called symbol
- assert_operator(obj1, operator, obj2, [msg]) - ensures obj1.operator(obj2) is true
- assert_send(array, [msg]) - ensures that executing method listed in array[1] on the object in array[0] with parameters of array[2+] is true
- flunk([msg]) - Forcibly fails this test
Unit tests in Practice-It

• the system tests submitted student code using JUnit
• the system (written in Java/JSP) also tests itself using JUnit

```java
@Test
public void testSolutions() throws Exception {
    for (Category category : Category.parseAll(Category.getCategory())) {
        for (Problem problem : problems) {
            int i = 1;
            for (String solution : codeProblem.getSolutions()) {
                codeProblem = (JavaCodeProblem) Problem.parseProblem(category);
                System.out.print(" solution " + i + ": ");

                totalTime++;
                long startTime = System.nanoTime();
                try {
                    codeProblem.runTests(solution);
                } catch (Exception e) {
                    System.out.println("exception while running tests:");
                    e.printStackTrace();
                }
                totalTime += System.nanoTime() - startTime;
                System.out.println("passed " + codeProblem.getTestCount());
            }
            String tests = "";
            if (codeProblem.getPassed()) {
                for (practiceit.Test test : codeProblem.getTests()) {
                    tests += test.toString() + "; solution:
                }
                assertEqual(codeProblem.getNumber() + "; solution + " + tests + codeProblem.getPassTests(), codeProblem.getPassed());
            }
        }
    }
}
```
Unit tests in Grade-It

- grading scripts test student homework using JUnit test cases
- the web grading system tests itself using PHPUnit/Simpletest

Differences

No differences found

Test passed!

FATAL ERROR: parameter 0 cannot be null
#0 check_not_null( ) called at [C:\Documents and Settings\stepp\My
#1 Criteria::readScoresheet( ) called at [C:\Documents and Settings
#2 TestCases->processTestCasesXml(DOMElement Object ( ) , ) called at [C:\Documents and Settings
#3 TestCases->printTestCases() called at [C:\Documents and Settings

Exception: TestTestCases -> testTestCases1 -> Unexpected exception of type
\src\common.php line 197]
Really compiling RandomWalk_Test1.java Really compiling RandomWalk_T
Unexpected PHP error [Trying to get property of non-object] severity [E

5/5 test cases complete: 102 passes, 0 fails and 2 exceptions.
Qualities of good tests

• test cases free of bugs
  – a broken test isn't much help

• readable test case code

• easy to add/update tests

• easy/fast to run
  – unit tests are often run on each build or checkin, so fast = good
Bugs in tests

• hard to find
  – developers assume that tests are correct

• manifest in odd ways
  – sometimes test initially passes, then begins to fail much later
    • code under test may have been altered in a subtle way
    • test case may have relied on invalid assumptions
    • API of code under test may have changed

• often test wasn't written by developer
  – bug assigned back and forth
What's wrong with this?

```java
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);
    }
}
```
Well-structured assertions

```java
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
```
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 and equals methods)

    @Test
    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);
    }
}
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("date after +4 days", expected, d);
    }

    @Test
    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);
    }

    private static final int DEFAULT_TIMEOUT = 2000;
}

// almost every test should have a timeout so it can't
// lead to an infinite loop; good to set a default, too

public class DateTest {
  @Test(timeout = DEFAULT_TIMEOUT)
  public void 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(timeout = DEFAULT_TIMEOUT)
  public void 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 variable names to expected/actual values

  private static final int DEFAULT_TIMEOUT = 2000;
}
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 y1, int m1, int d1, int add,
                            int y2, int m2, int d2) {
        Date actual = new Date(y, m, d);
        actual.addDays(add);
        Date expect = new Date(y2, m2, d2);
        assertEquals("after +" + add + " days", expect, actual);
    }

    // can also use "parameterized tests" in some frameworks ...
}
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_multipleCalls_wrapToNextMonth2x() {
        Date d = addHelper(2050, 2, 15, +14, 2050, 3, 1);
        addhelper(d, +32, 2050, 4, 2);
        addhelper(d, +98, 2050, 7, 9);
    }

    // Helpers can box you in; hard to test many calls/combine.
    // Create variations that allow better flexibility
    private Date addHelper(int y1, int m1, int d1, int add,
                           int y2, int m2, int d2) {
        Date date = new Date(y, m, d);
        addHelper(date, add, y2, m2, d2);
        return d;
    }

    private void addHelper(Date date, int add,
                           int y2, int m2, int d2) {
        date.addDays(add);
        Date expect = new Date(y2, m2, d2);
        assertEquals("date after +" + add + " days", expect, d);
    }
    ...
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("after adding one day to 2050/2/15,
                      should have gotten " + expected + "
                      but instead got " + actual,
                      expected, actual);
    }
    ...
}
Good assertion messages

```java
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("add one day to 2050/2/15",
                     expected, actual);
    }

    // JUnit will already show
    // the expected and actual
    // values in its output;
    //
    // don't need to repeat them
    // in the assertion message
```
public class DateTest {
    // test every day of the year
    @Test
    public void tortureTest() {
        Date date = new Date(2050, 1, 1);
        int month = 1;
        int day = 1;
        for (int i = 1; i < 365; i++) {
            date.addDays(1);
            if (day < DAYS_PER_MONTH[month]) {day++;}
            else {month++; day=1;}
            assertEquals(new Date(2050, month, day), date);
        }
    }

    private static final int[] DAYS_PER_MONTH = {
            0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
    // Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
Trustworthy tests

• Test one thing at a time per test method.
  – 10 small tests are much better than 1 test 10x as large.

• Each test method should have few (likely 1) assert statements.
  – If you assert many things, the first that fails stops the test.
  – You won't know whether a later assertion would have failed.

• Tests should avoid logic.
  – minimize if/else, loops, switch, etc.
  – avoid try/catch
    • If it's supposed to throw, use expected=... if not, let JUnit catch it.

• Torture tests are okay, but only in addition to simple tests.
public class DateTest {
    // shared Date object to test with (saves memory!!1)
    private static Date DATE;

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_sameMonth() {
        DATE = new Date(2050, 2, 15); // first test;
        addhelper(DATE, +4, 2050, 2, 19); // DATE = 2/15 here
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_nextMonthWrap() { // second test;
        addhelper(DATE, +10, 2050, 3, 1); // DATE = 2/19 here
    }

    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_multipleCalls() { // third test;
        addDays_sameMonth(); // go back to 2/19;
        addhelper(DATE, +1, 2050, 2, 20); // test two calls
        addhelper(DATE, +1, 2050, 2, 21);
    }

    ...
}
Isolation/order "smells"

- Tests should be self-contained and not care about each other.

- "Smells" (bad things to avoid) in tests:
  - *Constrained test order*: Test A must run before Test B. (usually a misguided attempt to test order/flow)
  - *Tests call each other*: Test A calls Test B's method (calling a shared helper is OK, though)
  - *Mutable shared state*: Tests A/B both use a shared object. If A breaks it, what happens to B?
Useful language features

- Elegant tests use the expressive features of your language.

- Java and many languages support variable numbers of params:

  ```java
  public void depositAll(Account a, double... amounts) {
    for (double amount : amounts) {
      a.deposit(amount);
    }
  }
  ...
  
  Account a = new Account("Shirley", 10.00);
  a.depositAll(4.00, 5.67, 8.90);
  a.depositAll(100.50);
  ```
Tests and data structures

- Need to pass lots of arrays? Use array literals
  ```java
  public void exampleMethod(int[] values) {
    ...
  }
  ...
  exampleMethod(new int[] {1, 2, 3, 4});
  exampleMethod(new int[] {5, 6, 7});
  ```

- Need a quick `ArrayList`? Try `Arrays.asList`
  ```java
  List<Integer> list = Arrays.asList(7, 4, -2, 3, 9, 18);
  ```

- Need a quick set, queue, etc.? Many collections can take a list
  ```java
  Set<Integer> list = new HashSet<Integer>(
    Arrays.asList(7, 4, -2, 9));
  ```
More data structures

• Need a quick Map or something else? Roll your own helper

```java
// pre-populates a map from given keys to given values
public static <K, V> Map<K, V> asMap(List<K> keys, List<V> values) {
    Map<K, V> map = new HashMap<K, V>();
    for (int i = 0; i < keys.size(); i++) {
        map.put(keys.get(i), values.get(i));
    }
    return map;
}

Map<String, Integer> taAges = asMap(
    Arrays.asList("Marty", "Logan", "Kelly", "Marisa"),
    Arrays.asList(23, 14, 39, 25);
);
```