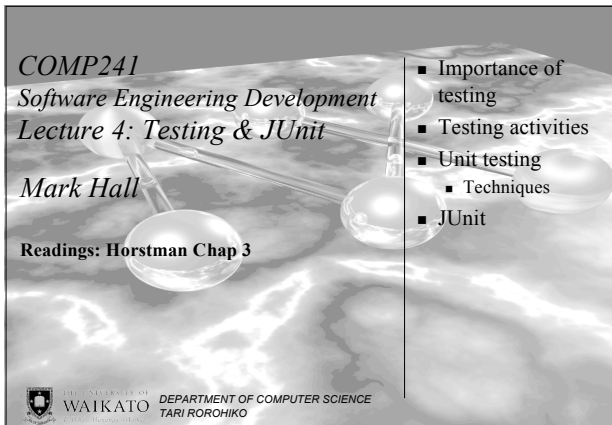


COMP241
Software Engineering Development
Lecture 4: Testing & JUnit

Mark Hall

Readings: Horstman Chap 3

- Importance of testing
- Testing activities
- Unit testing
 - Techniques
- JUnit



The University of Waikato DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

Why Are Tests Important?

- **Philosophy:** Everybody can change someone else's code as long as the tests still run
- **Regression testing:** test whether functionality that was working before a change is still working after the change

Other Benefits of Tests

- Documentation
- Helps to think about what to do before starting the implementation
- Helps with debugging
- Prevents bugs from reoccurring
- Allows refactoring with confidence

Testing Activities

- **Unit testing**
 - Focus on individual modules
- **Integration testing**
 - Checks the connection of modules
- **System testing**
 - Checks the complete system
- **Usability testing**
 - Tests the user's understanding of the system

Testing is hard

- Testing's goal runs counter to the goal of other development activities
 - A successful test is one that *breaks* the software!
- Testing can never prove the absence of errors
 - No errors could indicate poor/incomplete test cases
- Testing alone does not improve software quality
 - Test results are an *indicator* of quality

Unit Testing

- Focus on objects and subsystems
- Ensures that each module works properly before integrating modules together
- Easier to pinpoint and correct faults
- Allows parallelism

Black and White-box Testing

- Black-box testing
 - Don't (or can't) look past the interface when testing a routine
 - Look at just inputs and outputs
- White-box testing
 - Look at internal source as well as inputs and outputs

Unit Testing Techniques

- Why isn't it possible to prove that a program is correct by testing it?
 - Test every conceivable combination of input values
- e.g. program that stores a name (20 chars), address (20 chars) and phone number (10 digits) in a file:
 $26^{20} \times 26^{20} \times 10^{10} = 10^{66}$ possibilities

Unit Testing Techniques

- Incomplete Testing
 - Pick test cases most likely to find errors
 - Of the 10^{66} possible test cases, only a few are likely to disclose errors that others don't

Unit Testing Techniques

- Equivalence testing
 - A good test case covers a large part of the possible input data
 - If two test cases flush out the same errors, you only need one
 - Divide input into equivalence classes
 - Groups that you believe will be treated similarly by the routine
 - Test at least one example input for each class
- Boundary testing (off-by-one errors)
 - $\text{num}-1$, when you meant num ; \geq when you meant $>$
 - Choose input that tests the boundary of an equivalence class

Unit Testing Techniques

- Equivalence and boundary testing help the tester to do efficient and thorough black-box testing
- Determining the equivalence classes is not always easy
 - Must understand required input and domain-specific rules that govern what input is acceptable
 - Knowledge of computer science and software design

Example

```
/* Returns the number of days in
 * the given month and year.
 * @param month the integer
 *         representation for the
 *         month (1 is Jan, etc.)
 * @param year the year
 * @throws IllegalArgumentException
 *         if month < 1 or month > 12
 */
int getNumDaysInMonth(int month,
                    int year);
```

JUnit 3.8

- A unit test framework for Java
- Provides automatic test runs
- Provides automatic result checks
- Lots of extensions are available
- Make sure that `/usr/share/java/junit.jar` is in your CLASSPATH

Getting started with JUnit 3.8

1. Create a subclass of `TestCase`
2. Write a test method (name should start with `test`)
3. Run the test

1. Create a subclass of TestCase

```
import junit.framework.TestCase;

public class MyCalendarTest
    extends TestCase {
    ...
}
```

2. Write a test method

```
import junit.framework.TestCase;

public class MyCalendarTest
    extends TestCase {

    public void testFeb2000() {
        MyCalendar cal = new MyCalendar();
        final int result =
            cal.getNumDaysInMonth(2, 2000);
        assertEquals(29, result);
    }
}
```

3. Run the test

By using the `TestRunner` class:

- Console UI:

```
java junit.textui.TestRunner MyCalendarTest
```

- Graphical UI:

```
java junit.swingui.TestRunner MyCalendarTest
```

Writing tests using fixtures

- Fixture: known set of objects for multiple tests to operate on
1. Create a subclass of `TestCase`
 2. Add an instance variable for each part of the fixture
 3. Override `setUp()` to initialize the variables
 4. Override `tearDown()` to release any permanent resources you allocated in `setUp`

Execution Order

- `setUp()` will be executed before and `tearDown()` will be executed after each test method-invocation
- The ordering of test-method invocations is not defined
- Test methods must be written to be independent of one another

Example of using fixtures

```
import junit.framework.TestCase;

public class MyCalendarTest
    extends TestCase {
    protected void setUp() {
        myCalendar_ = new MyCalendar();
    }

    protected void tearDown() { }

    MyCalendar myCalendar_;
}
```

Grouping Tests Together

```
public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new
        MyCalendarTest("testFeb2000"));
    suite.addTest(new
        MyClassTest("testFeb1900"));
    return suite;
}

public static Test suite() {
    return new TestSuite(MyCalendarTest.class);
}
```

JUnit Terminology

- Test Fixture: Set of objects that can be reused by multiple tests
- TestCase: class that defines the fixture to run multiple tests
- TestSuite: Collection of Tests
- Failure: checked for
- Error: Exception (not checked for)

JUnit Best Practice

- Keep tests short and simple
- Don't assume order in which tests run
- Avoid tests with side effects
- Use OO techniques (subclassing etc.)
- Avoid loading data from hard-coded locations
- Use `assertSame`, `assertEquals` etc. instead of `assert(...==...)` and `assert(...equals(...))`
- Follow naming conventions