Unit testing

After B. Baudry and B. Combemale

Goals of unit testing

- Build confidence on the correctness of a class, when used separately
  - this regards the management of its internal state
  - each operation must be tested

Test and encapsulation

- The oracle needs access to the internal state to check for success or failure
- This goes against encapsulation
- Test points for test probe is a possible answer (by providing a test interface)

Overall process

- Test initialization operations
  - using retrieve accessors
- Test accessors
  - update then retrive
- Test service operations

Overall process (cont'd)

- At least one test case for each operation
- Some operations/methods depend on other objects
  - test stubs can be used to solve this

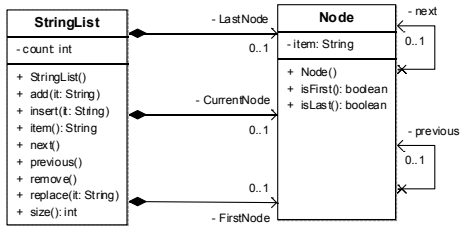
Junit test structure

- A test case is implemented
  - by an operation with an annotation
  - setup, calls and oracle are in the method
- A test class groups a set of test operations
  - usually a test class for each tested class

Structure for a test method

- Initialization
  - to set the object's internal state in a state suitable for the test
- A call to the object's operation
- A oracle statement
  - to compare the call's result with the expected result

Example



Example (cont'd)

- Nine operations
  - therefore at least nine test cases and therefor 9 test operations
- No access to the attributes
  - count, lastNode, currentNode, , firstNode
  - white box testing

The Junit framework

- Goals
  - to provide elementary services to ease test case definition
  - to automatically look for, execute and store test case execution results
  - to ease systematic testing and prevent regression

JUnit v4

- Packages
  - import org.junit.Test;
  - import static org.junit.Assert.*;
- Test cases
  - any public operation
  - annotated with @Test
  - they can expect exceptions
    - @Test(expected = *class*)

Before/after

- Setup operations are tagged
  - use @Before
  - import arg.junit.Before
  - execution order undefined if several setup operations
- Same logic for clean up using @After

BeforeClass/AfterClass

- An operation annotated with @BeforeClass
  - is run before execution of the test case sequence
- Mutatis mutandis for @AfterClass

Test suite

- Allow for control of which test cases are executed
  - @RunWith(Suite.class)
  - @SuiteClasses({MyList.class, Other.class})
  - class MyTestSuite {
    - // Empty
  - }

Parameters

- Test cases can receive parameters
- @RunWith(Parameterized.class)
- class TryIt {

Parameters' values

- @Parameters
- public static List<Object[]> getParameters() {
  - return Arrays.asList(
    - new Object[][] {
    - {10L, 20L}, {15L, 30L}, {0L, 0L}});
    - }
  - // We return a list of n-uplets

- // We need attributes to store the parameters
- // In our case
  - long mArgument;
  - long mResult;
- // The JUnit engine will inject values in the test class constructor

Test case body

- public TryIt(long argument, long result) {
  - mArgument = argument;
  - mResult = result;

Now the test cases

- @Test
- public void testOne {
  - assertEquals(argument * 2, result);
- }

What happens

- The test class will be instantiated for each parameter value
  - testOne will run three times

Another example

- interface Stack {
  - void push(Integer i);
  - void pop() raises EmptyStackException;
  - int top();
  - int height();
- }

Testing order

- Initialization (constructor/getter)
  - zero height
  - top must fail
- No other possibilities with an empty stack

Testing order (cont'd)

- A sequence of push/pop
- Then empy the stack and check it is empty

Test class initialization

- class StackTest {
  - private Stack stack;

```
@Before
public void initStack() {
  stack = new StackImpl();
}
```


Test initialization 1

```
@Test
public void testInit() {
  assertEquals(stack.height(),0);
}
```


Test initialization 2

```
@Test(expected = EmptyStackException)
public void testEmptyStack() {
  stack.pop();
}
```

Test 1 push

```
@Test
public void testOnePushTop () {
    assertEquals(stack.height(),0);
    stack.push(10);
    assertEquals(stack.height(),1)
    assertEquals(stack.top(),10);
}
```

Testing clear

```
@Test(expected = EmptyStackException)
public void testTwoPushPop () {
    stack.assertEquals(stack.height(),0);
    stack.push(10);
    stack.pop();
    assertEquals(stack.height(),0);
    pop();
}
```

Questions?

- You will use Junit in practical sessions

Writing mock classes


Mock classes

- Mock classes help in unit testing
  - they provide a simulated, controlled environment for the class under test
  - in practice, they are a vital part of the tests as they are a part of the oracle


Example

- For an implementation of the Observer design pattern
  - how does one check that the subject really calls the registered observers?
  - a mock observer is needed, that will record the subject's activity


Writing mocks is boring

- Many pieces of trivial software (stubs)
- And complex pieces (mechanisms for oracle design)
- But fortunately we now have mock generators

Mockito

- Mockito is a framework for defining mocks
  - It using Java introspection and intercession to define mock objects at runtime
  - Mockito is really powerful to design complex oracles

A simple Mockito example

- For an Observer design pattern implementation
  - In this example Mockito will generate a fake Observer object
  - This object will report on calls to its update() operation

Setting up the mock

- @Test
- public void testUpdateIsCalled()
- Observer<String> fake = Mockito.mock(Observer.class)
- subject.attach(fake)

Checking

- // Should trigger one call to the update
- // operation of the fake observer
- subject.setValue("Test")
- Mockito.verify(fake).update(subject)

Effect

- If the verification is not successful the surrounding test case will fail

More details on Mockito

- https://code.google.com/p/mockito/