Software Reengineering (COM7206)

Testing in Practice

Neil Walkinshaw

Department of Computer Science
University of Leicester

Testing in Practice

Testing is a big area

- Lots of different types of testing
  - Systems, integration, unit, mocks, . . .
- Lots of theoretical concepts
  - Oracles, specifications, adequacy, etc.

Testing in Practice

Testing is a big area

- Lots of different types of testing
  - Systems, integration, unit, mocks, . . .
- Lots of theoretical concepts
  - Oracles, specifications, adequacy, etc.

Relatively simple in practice

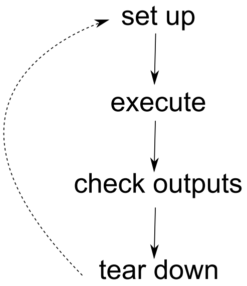
- Easier to demonstrate many of these notions in a practical setting
- Will use the JUnit Java testing framework
  - Can show how many of these concepts can be used routinely to assist development.

JUnit

An extremely successful framework

- Was developed by Kent Beck and Ward Cunningham around 1998
  - Motivated by agile software development
  - Needed a mechanism to rapidly and easily test Java code during development.
- Produced a lightweight framework to test Java classes
- Test oracles can be integrated into Java code
  - Can even be written before the code is!
- Routinely integrated into IDEs such as Eclipse

The Anatomy of a Unit Test



Instantiate the necessary data structures for the test, and put the SUT into a suitable initial state, and generating inputs, along with any necessary data to check the outputs.

Run the inputs on the system, and obtain outputs

Check that the outputs are correct

Remove the data objects produced by the set up, to ensure that they cannot interfere with subsequent test executions.

Set up

What are we trying to test?

- Need to get the SUT into base-state
  - E.g. *A bounded stack should not grow once it is full*
  - To test this property, it is necessary to first set up a full stack
- If the SUT interacts with other classes, these have to be set up
  - 'Fake' objects that are created just for testing are known as *mock* objects.
  - 'Fake' methods that simulate responses from elsewhere in the system are known as *method stubs*.
- A test set-up formalises presumptions about the system.
  - A set-up should be as generic / minimal as possible
    - Can you suggest why?

Test execution

Input generation

- We need to generate inputs to the system
- These must collectively exercise every facet of behaviour
  - May want to achieve code coverage, or other *adequacy criteria*.

Executing the inputs

- In some cases this might be simple
  - E.g. Execute a single call to “pop()” on the BoundedStack.
- In some cases it might be more complex
  - E.g. A sequence of calls to “push”, parameterised by a different letter in the alphabet, until the entire English alphabet has been exhausted, followed by a sequence of calls to “pop” until the stack is empty.

Check Outputs

The oracle

- Need to ensure that the outputs are correct
- Carried out by “assert” statements
- Two key challenges:
    - Ensure that the assert statements are broad enough to cover all relevant requirements
    - Ensure that the assert statements are detailed enough to ensure that they expose *every* possible fault

Tear-down

Re-setting the system

- It is vital that tests do not interfere with each other.
  - The outcome of one test must not affect the outcome of another.
- Test executions can change the state of the system
  - They might write data to a hard-disk, reconfigure a network connection, flush memory, etc.
- **Vital that these changes do not affect other tests**
  - Could undermine the presumptions under which other tests are executed.

Test Subject

A Bounded Stack

```
public class BoundedStack {

  private Stack stack;
  private int limit;

  public BoundedStack(int i){
    stack = new Stack();
    limit = i;
  }

  public void push(Object o){
    if(stack.size()<limit)
      stack.push(o);
  }

  public Object pop(){
    return stack.pop();
  }

  public int getSize(){
    return stack.size();
  }
}
```

Test Subject

A Bounded Stack

```
public class BoundedStack {

  private Stack stack;
  private int limit;

  public BoundedStack(int i){
    stack = new Stack();
    limit = i;
  }

  public void push(Object o){
    if(stack.size()<limit)
      stack.push(o);
  }

  public Object pop(){
    return stack.pop();
  }

  public int getSize(){
    return stack.size();
  }
}
```

Generated JUnit outline

```
public class BoundedStackTest {

  @Before
  public void setUp() throws Exception {
  }

  @After
  public void tearDown() throws Exception {
  }

  @Test
  public void testBoundedStack() {
    fail("Not yet implemented");
  }

  @Test
  public void testPush() {
    fail("Not yet implemented");
  }

  @Test
  public void testPop() {
    fail("Not yet implemented");
  }
}
```

Set-up and tear-down

What needs to be set up before each test?

Set-up and tear-down

What needs to be set up before each test?

- In this instance, nothing.
- Different tests will need to vary the limits on the stack size.
- Can leave instantiation of the stack to individual tests.

What needs to be torn-down?

- The test executions do not involve any persistent data
  - Nothing explicitly required here either.
- Could make sure that any test objects have been removed from memory.
  - Java does this automatically on a periodic basis with garbage collector
  - Could ensure that it is called as soon as test finishes by calling System.gc()

Writing the test cases

Oracles and Inputs

- What do we want to check, and how do we invoke it?
  - It should not be possible pop an empty stack.
  - Elements should not be added to the stack once it has reached its limit.

Writing the test cases

Oracles and Inputs

- What do we want to check, and how do we invoke it?
  - It should not be possible pop an empty stack.
  - Elements should not be added to the stack once it has reached its limit.

```
/*
 * We want an empty stack to throw an exception
 * if pop is called.
 */
@Test
public void testPopEmptyStack() {
  BoundedStack s = new BoundedStack(100);
  try{
    s.pop();
  }
  catch(Exception e){
    assertNotNull(e);
  }
}
```

Writing the test cases

Oracles and Inputs

- What do we want to check, and how do we invoke it?
  - It should not be possible pop an empty stack.
  - Elements should not be added to the stack once it has reached its limit.

```
/*
 * We want an empty stack to throw an exception
 * if pop is called.
 */
@Test
public void testPopEmptyStack() {
  BoundedStack s = new BoundedStack(100);
  try{
    s.pop();
  }
  catch(Exception e){
    assertNotNull(e);
  }
}
```

```
/*
 * Pushing an additional item to a full stack
 * should not add the item to the stack.
 */
@Test
public void testPushFullStack() {
  BoundedStack s = new BoundedStack(2);
  s.push("test element 1");
  s.push("test element 2");
  s.push("test element 3");
  String popped = (String)s.pop();
  assertEquals(popped, "test element 2");
}
```

Full test case attached on Blackboard