Why (unit) test?

- Programming is incremental by nature
  - We want to verify we haven’t broken anything
- We can’t safely refactor without tests
- Tests not only examine the code’s functionality but also its
  - Ease of use
  - API
  - Boilerplate & setup
  - Error path behavior
  - Coherency and Coupling issues
- Tests are the usage example of our code and are an excellent source for documentation
Testable Code $\Rightarrow$ Good Code

- Tests are usually the first time outside code will use our production code
- Is the production code hard to use?
  - Is object creation long and complex?
  - Are methods cryptic with lots of parameters?
  - How does your production code handle usage errors?
  - Have you been surprised by a method’s behavior?
- Is the production code hard to test?
  - Do you write 10-15 LOC before you reach the assert line?
  - Is it difficult to assert the desired behavior?
  - Do you find it hard isolating the tested feature?
  - Are you not even sure what you should be asserting?
Testing – Do’s & Don’ts

• Do not write bad tests to pass tests
  • **Hard to test? ⇒ Re-think the design!**
  • *private* methods are *private* for a reason
    • If you find yourself wanting to test a *private* method, you should rethink your design (e.g., extract the private method to a helper class)
• Always make sure that tests **can fail**
  • You get this “for free” when using TDD
• If you find yourself repeating the code’s logic in the test, what are you really testing?
  • Tests should only fail if your change your *interface*, not your *implementation*!
  • Prefer **black-box** to **white-box** testing whenever possible
Test types

- **Unit Tests** – Test a single element of the system
- **Integration Tests** – Verify that a few units can work together and are wired up correctly
- **End-to-End Tests** – Test the complete flow of interactions throughout the system
  - No “mocks”/”stubs”/”fakes”
- **Acceptance / Business logic Tests** – Answers the question: “Did we build what we wanted?”
  - Usually centered around a user story
Test types

JUnit

- A unit testing framework for Java
  - http://www.junit.org

- A **Test** is a Method
- A **Test Case** is a class which includes tests methods
  - Aggregates tests for a single tested unit
- A **Test Suite** is a collection of Test Cases / Test Suites (or both)
  - Used to create a hierarchy of tests
- The de-facto testing framework for Java
- Part the **JUnit** family
JUnit 4.0

- Test methods are annotated with `@Test`

- `@BeforeClass` – runs once before first test
  - Setting up the stubs objects / DB Connections

- `@AfterClass` – runs once after the last test
  - Closing DB connections

- `@Before` method – runs before every test
  - Preparing data
  - Could be used for advanced construction

- `@After` method – runs after every test
  - Resetting stubs / cleaning data
JUnit test case life-cycle

@A* - Running all methods annotated with A
JUnit life-cycle - Example

- Local **resources** should be initialized and cleaned before/after every test

```java
public class DbTests {
    private File output;
    @Before public void createOutputFile() {
        output = new File(...);
        output.createNewFile();
    }
    @Test public void testSomething() { ... }
    @Test public void testSomethingElse() { ... }
    @After public void deleteOutputFile() {
        output.delete();
    }
}
```
JUnit life-cycle - Example

- Expensive and **global** resources should be initialized and destroyed before/after all tests

```java
public class DbTests {
    private static DatabaseConnection database;
    @BeforeClass public static void login() {
        database = new DatabaseConnection();
        database.establishConnection();
    }
    @Test public void testSomething() { ... }
    @Test public void testSomethingElse() { ... }
    @AfterClass public static void logout() {
        database.logout();
    }
}
```
JUnit 4.0 - Example

• Assuming we want to test a new Stack interface implementation

```java
import org.junit.Assert;
import org.junit.Test;

public class StackTestCase {

    @Test
    public void peekReturnsTheLastPushedItem() {
        Stack<String> s = new ArrayListStack<>();
        s.push("FIRST-ITEM");
        s.push("LAST-ITEM");
        Assert.assertEquals("LAST-ITEM", s.peek());
    }
}
```

• Smart Asserts:
  • `assertNotEqual`, `assertTrue`, `assertSame`, `assertNotNull`, `assertArrayEqual`, etc.

```
org.junit.ComparisonFailure: expected:<[LAST]-ITEM> but was:<[FIRST]-ITEM>
```
Ideal Test

• A recipe for a good test:

```java
@Test
public void testShouldNameExplicitlyStatesExpectation() {
    (1) – Initialize collaborators (optional)
    (2) – Initialize object under test (optional)
    (3) – Exercise behavior
    (4) – Assert outputs
}
```

• If (1) and (2) are needed at all (e.g., initialization in constructor/@Before), it will be as short as possible (ideally just calls to `new`)
• (3) will be a single method invocation
• (4) will be a single assert

• Of course this ideal cannot always be achieved
• But if you can’t, you should always ask yourselves why?
Test names – Bad

It should be easy pin-pointing the problem when a test fails without looking at its code

• Consider the following test:

   ```java
   @Test
   public void addItemTest() {
       Product p = new Product();
       p.setAvailable(false);
       ShoppingCart cart = new ShoppingCart();
       cart.addItem(p);
       assertEquals(0, p.numItems());
   }
   ```

   • How about `addItemWithOutOfStockProduct`?
     • Describes the method and the parameter
     • Still not a good name – does not describes the desired behavior
Test names – Good

- It should be trivial understanding what is the expected behavior from the name alone
- We don’t have to mention the name of the invoked method

```java
@Test
public void doNotCountOutOfStockItem() {
    Product p = new Product();
    p.setAvailable(false);
    ShoppingCart cart = new ShoppingCart();
    cart.addItem(p);
    assertEquals(0, p.numItems());
}
```

- Good rule of thumb: the test name should include an explicit expectation
  - Use **verbs** such as should, does, returns, throws, etc.
JUnit – Testing for Exceptions

- The hard way – Using the `try/catch` pattern

```java
@Test
public void badEncodeArgument() {
    Encoder encoder = new Encoder("UTF-8");
    try {
        encoder.encode(null);
        fail("Exception should have been thrown");
    } catch (NullPointerException e) {
        assertEquals("Text should not be null", e.getMessage());
    }
}
```

- ☑ Checks for everything
- ☑ Can continue the test after the exception has been thrown
  - But please, don’t!
- 😞 Easy to forget the `fail` resulting in a test that never fails
- 😞 Quite long and verbose
  - Test code is just as important as our “regular” code! We want to minimize code duplication and boilerplate
JUnit – Exceptions (Cont.)

- The easy way (JUnit 4.0) – expected value in @Test annotation

```java
@Test(expected = IllegalArgumentException.class)
public void badEncodeArgument() {
    Encoder encoder = new Encoder("UTF-8");
    encoder.encode(null);
}
```

- 😞 Can’t check the exception message
- 😞 Can’t verify which line threw the exception
- 😊 Very short
- 😊 Is “good enough” in most cases
JUnit – Exceptions (Cont.)

• The smart way (JUnit 4.7) – Using @Rule annotation and ExpectedException

```java
@Rule public ExpectedException thrown = ExpectedException.none();

@Test
public void badEncodeArgument() {
    Encoder encoder = new Encoder("UTF-8");
    thrown.expect(NullPointerException.class);
    thrown.expectMessage("Text should not be null");
    encoder.encode(null);
}
```

• ☺ Can check for everything
• ☺ Still quite short

• JUnit Rules offers a few additional candies
JUnit - Timeout

• In JUnit 4.0 – Use the timeout annotation value

```java
@Test(timeout = 10)
public void testTimeout() {
    // Tested code
}
```

• Test will fail after 10 milliseconds.

• Global timeouts using @Rule (JUnit 4.7)

```java
@Rule public MethodRule globalTimeout = new Timeout(10);

@Test
public void test1() {
    // Tested code
}

@Test
public void test2() {
    // Tested code
}
```
Testing – Leveraging OOP

- Tests are classes too!
- Can use inheritance to our advantage
- Use abstract test classes against abstract classes / interfaces
Testing – OOP (Cont.)

- All test methods of the parent class are run for each concrete test class
- If A extends B, then A “is-a” B, and should therefore satisfy all of B’s tests!

```java
public abstract class Stack<T> {
    public void push(T element);
    public int size();
}

public class ArrayStack<T> extends Stack<T> {
    // … implementation code
}

public class LinkedListStack<T> extends Stack<T> {
    // … implementation code
}
```
public abstract class StackTest {
    private final Stack<Object> stack;

    protected StackTest(Stack<Object> stack) {
        this.stack = stack;
    }

    @Test
    public void sizeIsOneAfterOnePush() {
        stack.push(new Object());
        Assert.assertEquals(1, stack.size());
    }
}
Testing – OOP (Cont.)

• Extend abstract test class to initialize the concrete test subjects

• ArrayStack:

```java
public class ArrayStackTest extends StackTest {
    public ArrayStackTest() {
        super(new ArrayStack<Object>());
    }
    // specific ArrayStack tests
}
```

• LinkedListStack:

```java
public class LinkedListStackTest extends StackTest {
    public LinkedListStackTest() {
        super(new LinkedListStack<Object>());
    }
    // specific LinkedListStackTest tests
}
```
Appendix

Other nice to know features
Testing – Order (or lack thereof)

- **Do not assume order between tests**
  - There’s no order guaranties in TestCases
  - There is one in TestSuites
- **Avoid side effects in tests!**

```java
public class StackTest {

    private static Stack<Object> s = new Stack<>();

    @Test
    public void pushFirstAndCheckSize() {
        s.push(new Object());
        assertEquals(1, s.size());
    }

    @Test
    public void pushSecondAndCheckSize() {
        s.push(new Object());
        assertEquals(2, s.size());
    }
}
```
Testing & Debugging

• When you work on a bug, write a test case first
  • Your work is done when the test case succeeds
  • If you write the test after the bug is fixed, you cannot know if the test succeeded in reproducing the bug
  • Helps with regression testing, i.e., ensuring the bug won’t reappear later

• You can actually debug using unit tests!
  1. Write a test that fails
  2. If you still aren’t sure what the bug is, write a smaller unit test that fails
  3. Repeat step 2 as necessary

• This could be preferable to running a debugger, as the tests remain when done
Ignoring tests

- You can ignore tests using the `@Ignore` annotation
  - Test will not run, but will still be visible in the test reports
- Commented out tests will usually be forgotten about
  - Won’t be deleted and possibly will not be fixed
- Ignored tests are easily visible in the output window

- Make sure you disable tests temporarily
  - Bad tests should either be deleted or fixed
  - Ask yourself – why are you ignoring the test in the first place?
Testing – Resource Management

• Do not load data from hard-coded locations on a file system

```java
@Before
public void setUp () {
    InputStream inp = new FileInputStream("C:\\TestData\\dataSet1.dat");
    ...
}
```

• Use `Class.getResource()` or `Class.getResourceAsStream()`

```java
@Before
public void setUp () {
    InputStream inp = getClass().getResourceAsStream("dataSet1.dat");
    ...
}
```

• Will search the file relatively to the Test Case .class file
• Maven will automatically create a src & test resource folders
Appendix:
Source code organization

- Option 1: Test class in same folder as subject
  - A single hierarchy of folders
  - Locality

- Option 2 (much more common): Keep tests in the mirrored directory structure of source location
  - A duplicated hierarchy of folders
  - Easy to find all tests
  - Easy to separate source code from its tests
  - Easier bytecode separation
Code Coverage - EclEmma
Code Coverage

• Code coverage is correlated with good testing, but does not equate it
  • You can achieve very high coverage with bad tests (e.g., testing without asserts)
  • You can write great tests with low coverage
• 100% coverage may look “pretty”, but should not be a goal in and off itself
  • You should always suspect tests that were written solely for achieving high coverage
  • Do you really need to test for NullPointerException for each of your parameters?
Extra reading:
Test types – misc.

- **Behavior-driven development**
  - Using *Domain Specific Language* (DSL) to structure tests around stories and features

- **Property tests**
  - Large randomized input to check invariants and properties of output

- **Mutability testing**
  - Changing (mutating) the source code or tests slightly, to see if our tests fail
  - This ensures tight coupling between tests and source code

- **Design by Contract**
  - Asserting *pre and post-conditions* on class’s methods
  - Fails in *runtime*, possible to check even at compile time
Extra reading:

• Test Driven Development (Kent Beck)
  • http://www.amazon.com/Test-Driven-Development-By-Example/dp/0321146530

• Tests vs Types
  • http://kevinmahoney.co.uk/articles/tests-vs-types

• Code coverage considered harmful?
  • http://erengu.com/misc/code-coverage.html