Module 4 – Topic 4.2 – Lesson 4.2.2

Test Driven Development (TDD)

Lesson Outline

- The rationale of test driven development
- The process of test driven development
- Tool support – JUnit

You will practice test-driven development and refactoring in the Laboratory 2 in Week 9.

TDD Rationale and Process

Recall Continuous Testing in XP

- Test-driven development – tests are written for a feature before the source code is written

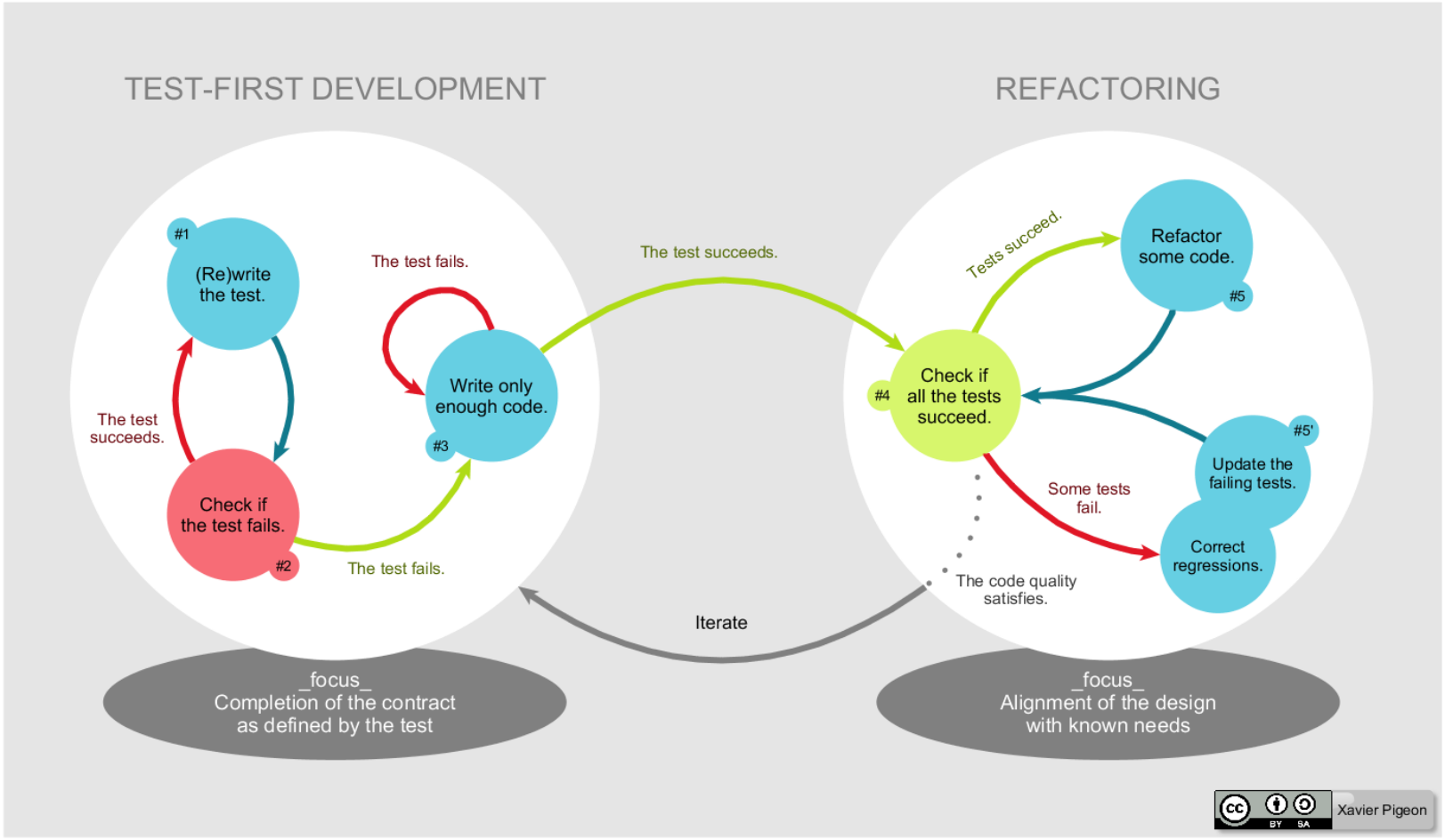
- Tests are executable forms of requirements
  - Acceptance tests for the client to test that each feature of the overall product works as specified
  - Unit test for the developers to test lower-level functionality

Test-Driven Development (TDD)

- An agile software development practice
- Focuses on small incremental changes
- Relies on test automation
- A very short development cycle

1. Add a test
2. Run all tests, and confirm that all new test fails
3. Write production code to cause the test to pass
4. Run all tests again
5. Refactor code if necessary
6. Repeat

TDD Process (Visualized)



https://en.wikipedia.org/wiki/Test-driven_development

TDD Pros and Cons

- More tests, less debugging
- Validates not only code, but also design
- Better test coverage, greater confidence
- Better code structure

- Difficult in certain scenarios
  - UI, network, …
- Needs management support
- Developer blind spots
  - False sense of quality
- Maintenance overhead for tests

TDD Tool Support

Use JUnit as an example

JUnit

- An open source unit testing framework for Java
- Provides facilities for
  - Setting up and tearing down test fixtures
  - Writing assertions on expected results of test execution
  - Organizing tests into test suites
  - Running tests

JUnit tutorial: http://www.javavids.com/tutorial/junit.html

Google "unit testing language_you_want_to_use" to find unit testing framework for the language you want to use. All such unit testing frameworks share the same concepts.

JUnit Basics

- JUnit tests are methods in a test class
- Naming convention of test classes: name of class under test + Test
  - e.g., ShoppingCartTest.java for ShoppingCart.java
- Tests are annoted with @Test
- Test names should reflect purpose

```
@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {

   // MyClass is tested
   MyClass tester = new MyClass();

   // Tests
   assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
   assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
   assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
 }
```

Test Fixtures

- A fixed state of a set of objects used as a baseline for running tests, e.g.,
  - preparing input data
  - setting up database connection
  - initializing objects to fixed states
- Through methods with annotations
  - @Before, @After
  - @BeforeClass, @AfterClass

```
private ManagedResource myManagedResource;
private static ExpensiveManagedResource myExpensiveManagedResource;
```

```
@BeforeClass
public static void setUpClass() {
  System.out.println("@BeforeClass setUpClass");
  myExpensiveManagedResource = new ExpensiveManagedResource();
}

@AfterClass
public static void tearDownClass() throws IOException {
  System.out.println("@AfterClass tearDownClass");
  myExpensiveManagedResource.close();
  myExpensiveManagedResource = null;
}
```

Writing JUnit Tests

- With annotation @Test + optional parameters

```
@Test
public void testAssertNotNull() {
  org.junit.Assert.assertNotNull("should not be null", new Object());
}
```

  - Exception testing: @Test (expected=Exception.class)

```
@Test(expected= IndexOutOfBoundsException.class)
public void empty() {
     new ArrayList<Object>().get(0);
}
```

  - Timeout: @Test (timeout=1000)

```
@Test(timeout=1000)
public void testWithTimeout() {
  ...
}
```

- Tests can be ignored with @Ignore

```
@Ignore("Test is ignored as a demonstration")
@Test
public void testSane() {
    assertThat(1, is(1));
}
```

Making Assertions

- Assertions allows us to compare expected output with actual outputs
- A JUnit test runner runs tests, checks assertions and report test results

assertTrue()	assertFalse()
assertEquals()	assertNotEquals()
assertNull()	assertNotNull()
assertSame()	assertNotSame()
assertArrayEquals()	…

*parameters omitted

Assertion Examples

- https://github.com/junit-team/junit4/wiki/Assertions

```
@Test
public void testAssertEquals() {
  org.junit.Assert.assertEquals("failure - strings are not equal", "text", "text");
}

@Test
public void testAssertFalse() {
  org.junit.Assert.assertFalse("failure - should be false", false);
}

@Test
public void testAssertNotNull() {
  org.junit.Assert.assertNotNull("should not be null", new Object());
}
```

TDD Best Practices

- Make test cases independent of each other and without side effects
- Test only one thing in a test
- setUp, test, tearDown
- Name tests sensibly and consistently
- Write descriptive message in assertions