Test-Driven Development and Refactoring

Dalibor Dvorski (ddvorski@conestogac.on.ca)

School of Engineering and Information Technology

Conestoga College Institute of Technology and Advanced Learning

Test-Driven Development

test-driven development (TDD) (i.e. test-first development):

- a software development process that repeatedly involves the writing of test cases, followed by initial code written to pass the aforementioned test cases, followed by improved rewritten code
    - the writing of test cases is done using unit testing
        - **unit testing**: method of testing software where the individual components of code are tested to determine if they were written correctly
    - the improved rewritten code is referred to as refactoring
        - **refactoring**: method of rewriting code without changing its functionality (i.e. behaviour) in order to improve on its non-functional features
- write a little test code, then write a little production code, make it pass the test, then write some more test code, etc.
- practice promoted by the iterative and agile XP method and applicable to the UP

Test-Driven Development

Advantages of test-driven development include:

- *The unit tests actually get written* – Human (or at least programmer) nature is such that avoidance of writing unit tests is very common, if left as an afterthought.
- *Programmer satisfaction leading to more consistent test writing* – This is more important than it sounds for sustainable, enjoyable testing work. If, following the traditional style, a developer first writes the production code, informally debugs it, and then as an afterthought is expected to add unit tests, it doesn't feel satisfying. This is "test-last development", i.e. "just-this-one-time-I'll-skip-writing-the-test" development. It's human psychology. However, if the test is written first, we feel a worthwhile challenge and question in front of us: Can I write code to pass this test? And then, after the code is cut to pass the tests, there is some feeling of accomplishment – meeting the goal. And a very useful goal – an executable, repeatable test. The psychological aspects of development can't be ignored – programming is a human endeavour.

Test-Driven Development

- *Clarification of detailed interface and behaviour* – The sounds subtle, but it turns out in practice to be a major value of TDD. Consider your state of mind if you write he test for an object first: As you write the test code, you must imagine that the object code exists. e.g. if in your test code you write `sale.makeLineItem(description, 3)` to test the `makeLineItem` method (which doesn't yet exist), you must think through the details of the public view of the method – its name, return value, parameters, and behaviour. That reflection improves or clarifies the detailed design.
- *Provable, repeatable, automated verification* – Obviously, having hundreds or thousands of unit tests that build up over the weeks provides some meaningful verification of correctness. And because they can be run automatically, it's easy. Over time, as the test base builds from 10 tests to 50 tests to 500 tests, the early, more painful investment in writing tests starts to really feel like its paying off as the size of the application grows.
- *The confidence to change things* – In TDD, there will eventually be hundreds or thousands of unit tests, and a unit test class for each production class. When a developer needs to change existing code – written by themselves or others – there is a unit test suite that can be run, providing immediate feedback if the change caused an error.

Unit Testing Frameworks

- *JUnit* (http://junit.org/):
  framework to write unit tests for Java programs
- *NUnit* (http://nunit.org/):
  framework to write unit tests for .NET programs
- *PHPUnit* (http://phpunit.de/):
  framework to write unit tests for PHP programs
- ~~*JsUnit* (https://github.com/pivotal/jsunit/):~~
  ~~framework to write unit tests for JavaScript programs~~
  - *not actively developed or supported*
    – instead use Jasmine (http://jasmine.github.io/)

$\Bigg\}$ family of unit testing frameworks known as *xUnit*, all of which are based on sUnit – a unit testing framework for Smalltalk programs

TDD with JUnit

e.g. Suppose we are using JUnit and TDD to create the `Sale` class. Before programming the `Sale` class, we write a unit testing method in a `SaleTest` class that does the following:

1. Create a `Sale` – the thing to be tested (i.e. the fixture).
2. Add some line items to it with the `makeLineItem` method (the `makeLineItem` method is the public method we wish to test).
3. Ask for the total, and verify that it is the expected value, using the `assertTrue` method. JUnit will indicate a failure if any `assertTrue` statement does not evaluate to `true`.

Each testing method follows this pattern:

1. Create the fixture.
2. Do something to it (some operation that you want to test).
3. Evaluate that the results are as expected.

A key point to note is that we do not write all the unit tests for `Sale` first; rather, we write only one test method, implement the solution in class `Sale` to make it pass, and then repeat.

TDD with JUnit

§ 21.1 of *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* presents a sample `SaleTest` class.

Refactoring

What are the activities and goals of refactoring? They are simply the activities and goals of good programming: remove duplicate code, improve clarity, make long methods shorter, remove the use of hard-coded literal constants, etc.

Code that's been well-refactored is short, tight, clear, and without duplication – it looks like the work of a master programmer. Code that doesn't have these quantities "smells bad" or has "code smells". i.e. there is a poor design. "Code smells" is a metaphor in refactoring – they are hints that something may be wrong in the code. The name "code smells" was chosen to suggest that when we look into the smelly code, it might turn out to be alright and not need improvement. That's in contrast to "code stench" – truly putrid code crying out for a clean up! Some "code smells" include: duplicated code, big methods, classes with many instance variables, classes with lots of code, strikingly similar subclasses, little or no use of interfaces in the design, high coupling between many objects, etc.

The remedy to "smelly code" are refactorings. Like patterns, refactorings have names, such as Extract Method. There are about 100 named refactorings.

Refactoring

refactoring	*description*
Extract Method	Transform a long method into a shorter one by factoring out a portion into a private helper method.
Extract Constant	Replace a literal constant with a constant variable.
Introduce Explaining Variable (specialization of Extract Local Variable)	Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.
Replace Constructor Call with Factory Method	Replace the `new` operator and constructor call with invoking a helper method that creates the object (hiding the details).

Refactoring

```
public class Player
{
 private Piece piece;
 private Board board;
 private Die[] dice;

 public void takeTurn()
 {
  int rollTotal = 0;

  for (int i = 0; i < dice.length; i++)
  {
   dice[i].roll();
   rollTotal += dice[i].getFaceValue();
  }

  Square newLocation = board.getSquare(piece.getLocation(), rollTotal);
  piece.setLocation(newLocation);
 }
}
```

This example demonstrates the common Extract Method refactoring. Notice that the `takeTurn` method in the `Player` class has an initial section of code that rolls the dice and calculates the total in a loop. This code is itself a distinct, cohesive unit of behaviour; we can make the `takeTurn` method shorter, clearer, and better supporting High Cohesion by extracting that code into a private helper method called `rollDice`. Notice that the `rollTotal` value is required in `takeTurn`, so this helper method must return the `rollTotal`.

Assigned Readings

1. Chapter 21: Test-Driven Development and Refactoring from
   Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.
   - Improve the `Player` class by refactoring.
2. Chapter 22: UML Tools and UML as a Blueprint from
   Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.