Software Engineering — Lab 2

Uppsala University

Spring 2013

Introduction

In this block we'll mainly focus on application testing. While there are many ways to test code we will focus on unit testing. We will be creating the same application three times. First through test-after development (TAD), then test-driven development (TDD), and finally through exceptionally strict TDD. Try to complete the exercises in the same order as outlined below.

Please feel free to work in pairs and take turns on writing code. Good luck!

1 Testing existing code (FizzBuzz)

You've been provided with a project containing a working implementation of the FizzBuzz problem (from the lectures). Your job is to write unit tests with reasonable coverage. When you are done, it should be hard to introduce errors to the application without also making tests fail. At a minimum, write tests for the following cases:

1. `Fizz` is returned, when given some number divisible by 3
2. `Buzz` is returned, when given some number divisible by 5
3. `FizzBuzz` is returned, when given some number divisible by 3 *and* 5
4. The input number itself is returned, when given some input number not divisible by 3 *nor* 5

If you can't remember what the rules of the FizzBuzz game is — the above cases is basically an explanation for the rules of the game (if you change the word ”some” for ”any”).

Are these tests enough? While the answer to this question of course greatly depends on how you have written your tests, it is quite likely that the answer is no. Do keep this question in mind as you approximate the section on ”sliming”.

2 Test-driven development (FizzBuzz)

Create a new console project. We're now going to build the FizzBuzz application again, but this time through test-driven development (TDD). Basically this means that we must build our application only by iterating the steps below:

1. Introduce a (or change an existing) test
2. Watch the test fail (for the right reason)
3. Write the minimal code needed to make it pass

Different schools of thought argue the meaning of "minimal code needed". However, in this exercise, think of it as: writing the simplest production code possible, that reflects the "idea" of the test, making it pass.

An easy way to remember the above steps is to think of the idiom red-green-refactor. Basically it means that we should produce a state where some test fail (red). Then write enough code to make it pass (green). Only then, are we allowed to refactor code.

New requirements

This time we'll introduce a new requirements. Apart from the `FizzOrBuzz(int n)` method you should add another public method to the Fizzbuzzer:

```
string Count(int n);
  // Where n represent the number to count to
  // Example, if n = 5 then the output should be
  // "1 2 Fizz 4 Buzz"
```

3 "Sliming" (FizzBuzz)

Create a new console project. Once again, we'll build the FizzBuzz application (with the new requirement). Once again, will we use use TDD. However this time we will "slime" every step of implementation.

Sliming (fake it 'til you make it) is a tactic used as a means to force yourself to ensure high test-coverage. Usually, when we are "writing the minimal code needed, we tend to write "sensible" code (the implementation we actually want). However "sliming" propose we should really write the simplest possible code, no matter how dumb it is.

Say the goal is to produce the function `F(x)=x*x`. Then say we write a test, that calls `F(2)`, and expects (asserts) a return value of `4`. If we are "sliming", then we write the simplest function we can possibly write to ensure the test is satisfied. So we write the following function:

```
int f(int x){
  return 4;
}
```

Unintelligent code indeed. Say we continue by writing a new test, calling `F(10)`, that expects a return value of `100`. Since we are sliming, we will change our production code to the following:

```
int f(int x){
  return x == 2 ? 4 : 100;
}
```

Utterly stupid code, and as humans we know this is not even close to the final behaviour we want. However, by doing this process, we are forced to write more (or better) tests. We only stop sliming when it becomes harder than writing the implementation we actually want to write.

4 Contemplation (TDD vs TAD)

What are the positives and negatives of TDD? TAD? Write a few sentances (max half-a-page) about your thoughts on these concepts. When are they good to use? Less good? Was sliming helpful? Always? Who are tests for? Is testing useful for only one or many things?

5 Debugging using tests (Matrix)

You have been provided with code for a class that handles mathematical operations on Matrices. A class which supposedly was written by a really lousy programmer. It does not behave as expected. Your job is to use either TDD or TAD to provide the matrix class with desired behaviour (free from logical errors). Use testing as a means of localizing errors.

A matrix is a two-dimensional array of elements (in this case numbers). If you are not familiar with matrices, consult Wikipedia (as a suggestion) to learn how matrix multiplication is performed, and how to calculate the inverse of a 2x2 matrix. Knowledge about matrix operations is vital to write good test cases, and to solve this assignment.

6 Test coverage (Phone book)

You've been provided with a class, simliar to the previously discussed phone book example. Your job is to write tests for this class, attempting to achieve *as high test coverage as possible.* In a perfect world it will be impossible to introduce errors to the class without any tests failing.

You may change the code of the class if you wish, but you are *not* allowed to introduce new public methods to the class, nor change the method signatures of the existing ones.

When you believe you have achieved a high level of coverage. Grab a friend and let her or him try to make your tests report false-positives. A false-positive is when a test is passing, but the functionality it is supposed to test has actually been broken. It is easy to get snowblind watching your own code so having someone else attempting to "break" it is usually a good idea.

7 Contemplation (Test-coverage)

Does full test-coverage ensure perfect code? Write a few sentances (max half-a-page) about your thoughts on the positives and negatives of test-coverage. What is it good for? Not good for? How good is it? When is it relevant? When not? Is there a golden mean? Etc.

8 TDD practice (Bowling game kata)

If you are finished with all of the above, this *optional* exercise allows you to practice your TDD skills by doing the Bowling game kata. You've been provided with some basic code to get you started with the exercise. The bowling game class exposes two public methods whose bodies you need to implement.

```
void Roll(int pins)
  // Where pins is the number of pins knocked
  // down by the player in this roll

int Score()
  // Returns the current score at the given time
```

The rules of bowling

1. Each game consist of 10 frames (rounds)
2. Each frame consist of 2 rolls
3. Each roll can knock down 0-10 pins
4. The score for the frame is the total number of pins knocked down in that frame + plus bonuses for strikes and spares
5. A spare is when the player knocks down all 10 pins in two rolls
6. The score of a spare is the number of pins knocked down in the next roll
7. A strike is when the player knocks down all 10 pins in the first roll
8. The score of a strike is the number of pins knocked down in the next frame
9. A frame only consist of one roll if the player rolls a strike
10. In the tenth frame the player will have one extra roll *if* a strike or spare is rolled

Remember! Red-green-refactor.

9 Refactoring (Movie rental case)

Now, it's time to forget about testing (for now) and instead focus on refactoring. The lousy programmer has been at it again. He or she has built a system for a movie rental shop handling customer orders. However, since the lousy programmer was focused on meeting the deadline the code is way more procedural than object oriented.

The `Customer` class has a `Statement()` method, that returns a string of information on a given customers current rentals. However, as a requirement to print the statement as a HTML formatted string just came in, the lousy programmer realized the code is not at all prepared for change.

Your job is now to refactor the system so that the new method can be implemented without introducing significant duplication.

Ideas on what to refactor

A good place to start is to have a look at the `Statement()` method itself. It is way too long! Apart from that, think about the following points:

Method extraction Can some behavior be extracted into a different method?

Replacing case statements Can we use inheritance instead of case statements?

Creation methods Can we use creation methods to make instantiation safer?

Moving behavior Should a given behavior really be defined on another class? A rule-of-thumb is to consider the Law of Demeter, which say that a class should *only* talk to its closest neighbors. Meaning, it's OK to say:

```
instanceOfSomeClass.Method()
```

But it's not OK to say:

```
instanceOfSomeClass.instanceOfSomeOtherClass.Method()
```

Good luck!