Test-Driven Development

CSCI 5828: Foundations of Software Engineering
Lecture 22 — 11/03/2016
Credit where Credit is Due

• Some of the material for this lecture is taken from “Test-Driven Development” by Kent Beck
  • as such some of this material is copyright © Addison Wesley, 2003

• In addition, some material for this lecture is taken from “Agile Software Development: Principles, Patterns, and Practices” by Robert C. Martin
  • as such some materials is copyright © Pearson Education, Inc., 2003
Goals for this lecture

• Introduce the concept of Test-Driven Development (TDD)

• Present several examples
Test-Driven Development

• The idea is simple
  • No production code is written except to make a failing test pass

• Implication
  • You have to write test cases before you write code

• Note: use of the word “production”
  • which refers to code that is going to be deployed to and used by real users

• It does not say: “No code is written except…”
Test-Driven Design in a Nutshell

• This means that when you first write a test case, you may be testing code that does not exist
  
  • And since that means the test case will not compile, obviously the test case “fails”
    
    • After you write the skeleton code for the objects referenced in the test case, it will now compile, but also may not pass
  
  • So, then you write the simplest code that will make the test case pass
Example (I)

• Consider writing a program to score the game of bowling
• You might start with the following test
  
  ```java
  public class TestGame extends TestCase {
    public void testOneThrow() {
      Game g = new Game();
      g.addThrow(5);
      assertEquals(5, g.getScore());
    }
  }
  ```
• When you compile this program, the test “fails” because the Game class does not yet exist. But:
  • You have defined two methods on the class that you want to use
  • You are designing this class from a client’s perspective
Example (II)

• You would now write the Game class

```java
public class Game {
    public void addThrow(int pins) {
    }
    public int getScore() {
        return 0;
    }
}
```

• The code now compiles but the test will still fail: getScore() returns 0 not 5
  • In Test-Driven Design, Beck recommends taking small, simple steps
  • So, we get the test case to compile before we get it to pass
Example (III)

• Once we confirm that the test still fails, we would then write the simplest code to make the test case pass; that would be

```java
public class Game {
    public void addThrow(int pins) {
        // Implement addThrow method
    }
    public int getScore() {
        return 5;
    }
}
```

• The test case now passes!
Example (IV)

• But, this code is not very useful!

• Lets add a new test case to enable progress

```java
public class TestGame extends TestCase {
    public void testOneThrow() {
        Game g = new Game();
        g.addThrow(5);
        assertEquals(5, g.getScore());
    }
    public void testTwoThrows() {
        Game g = new Game();
        g.addThrow(5);
        g.addThrow(4);
        assertEquals(9, g.getScore());
    }
}
```

• The first test passes, but the second case fails (since 9 ≠ 5)
  • This code is written using JUnit; it uses reflection to invoke tests automatically
Example (V)

- We have duplication of information between the first test and the Game class
  - In particular, the number 5 appears in both places
- This duplication occurred because we were writing the simplest code to make the test pass
  - Now, in the presence of the second test case, this duplication does more harm than good
- So, we must now refactor the code to remove this duplication
Example (VI)

```java
public class Game {
    private int score = 0;

    public void addThrow(int pins) {
        score += pins;
    }

    public int getScore() {
        return score;
    }
}

Both tests now pass. Progress!
```
Example (VII)

• But now, to make additional progress, we add another test case to the TestGame class

...  
  public void testSimpleSpare() {
      Game g = new Game()
      g.addThrow(3); g.addThrow(7); g.addThrow(3);
      assertEquals(13, g.scoreForFrame(1));
      assertEquals(16, g.getScore());
  }

...  

• We’re back to the code not compiling due to scoreForFrame()
  • We’ll need to add a method body for this method and give it the simplest implementation that will make all three of our tests cases pass
TDD Life Cycle

• The life cycle of test-driven development is
  • Quickly add a test
  • Run all tests and see the new one fail
  • Make a simple change
  • Run all tests and see them all pass
  • Refactor to remove duplication
• This cycle is followed until you have met your goal
TDD Life Cycle, continued

• Kent Beck likes to perform TDD using a testing framework, such as JUnit.

• Within such frameworks
  • failing tests are indicated with a “red bar”
  • passing tests are shown with a “green bar”

• As such, the TDD life cycle is sometimes described as
  • “red bar/green bar/refactor”
JUnit: Red Bar...

• When a test fails:
  • You see a red bar
  • Failures/Errors are listed
  • Clicking on a failure displays more detailed information about what went wrong
Example Background: Multi-Currency Money

• Lets design a system that will allow us to perform financial transactions with money that may be in different currencies

  • e.g. if we know that the exchange rate from Swiss Francs to U.S. Dollars is 2 to 1 then we can calculate expressions like

    • 5 USD + 10 CHF = 10 USD

  • or

    • 5 USD + 10 CHF = 20 CHF
Starting From Scratch

• Lets start developing such an example

• How do we start?
  • TDD recommends writing a list of things we want to test
  • This list can take any format, just keep it simple
  • Example
    • $5 + 10 \text{ CHF} = $10 \text{ if rate is } 2:1$
    • $5 \times 2 = $10$
First Test

• The first test case looks a bit complex, let's start with the second
  
  • 5 USD * 2 = 10 USD

• First, we write a test case

```java
public void testMultiplication() {
    Dollar five = new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount)
}
```
Discussion on Test Case

```java
public void testMultiplication() {
   Dollar five = new Dollar(5);
   five.times(2);
   assertEquals(10, five.amount)
}
```

• What benefits does this provide?
  • target class plus some of its interface
    • we are designing the interface of the Dollar class by thinking about how we would want to use it
  • We have made a testable assertion about the state of that class after we perform a particular sequence of operations
What’s Next?

• We need to update our test list
  • The test case revealed some things about Dollar that we will want to address
    • We are representing the amount as an integer, which will make it difficult to represent values like 1.5 USD; how will we handle rounding of factional amounts?
    • Dollar.amount is public; violates encapsulation
    • What about side effects?; we first declared our variable as “five” but after we performed the multiplication it now equals “ten”
Update Testing List

• The New List
  • 5 USD + 10 CHF = 10 USD
  • $5 * 2 = $10
  • make “amount” private
  • Dollar side-effects?
  • Money rounding?

• Now, we need to fix the compile errors
  • no class Dollar, no constructor, no method: times(), no field: amount
First version of Dollar Class

```java
public class Dollar {

    public Dollar(int amount) {
    
    }

    public void times(int multiplier) {
    
    }

    public int amount;

}

• Now our test compiles and fails!
```
Too Slow?

• Note: we did the simplest thing to make the test compile;

• now, we are going to do the simplest thing to make the test pass

• Is this process too slow?

  • **YES**, as you get familiar with the TDD life cycle you will gain confidence and make bigger steps

  • **NO**, taking small simple steps avoids mistakes;

    • novice programmers try to code too much before invoking the compiler;

      • they then spend the rest of their time debugging!
How do we make the test pass?

• Here’s one way

```java
public void times(int multiplier) {
    amount = 5 * 2;
}
```

• The test now passes, we received a “green bar”!

• Now, we need to “refactor to remove duplication”
  • But where is the duplication?
Refactoring

• To remove the duplication of the test data and the hard-wired code of the times method, we think the following

• “We are trying to get a 10 at the end of our test case and we’ve been given a 5 in the constructor and a 2 was passed as a parameter to the times method”

• So, let’s connect the dots…
First version of Dollar Class

```java
public class Dollar {

    public Dollar(int amount) {
        this.amount = amount;
    }

    public void times(int multiplier) {
        amount = amount * multiplier;
    }

    public int amount;
}

• Now our test compiles and passes, and we didn’t have to cheat!
```
One loop complete!

• Before writing the next test case, we update our testing list
  • 5 USD + 10 CHF = 10 USD
  • $5 \times 2 = $10
  • make “amount” private
  • Dollar side-effects?
  • Money rounding?
One more example

• Lets address the “Dollar Side-Effects” item and then move on to another example

• Lets write the next test case
  
  • When we called the times operation our variable “five” was pointing at an object whose amount equaled “ten”; not good
    
    • the times operation had a side effect which was to change the value of a previously created “value object”

  • Think about it, as much as you might like to, you can’t change a 5 dollar bill into a 500 dollar bill; the 5 dollar bill remains the same throughout multiple financial transactions
Next test case

• The behavior we want is

```java
public void testMultiplication() {
    Dollar five = new Dollar(5);
    Dollar product = five.times(2);
    assertEquals(10, product.amount);
    product = five.times(3);
    assertEquals(15, product.amount);
    assertEquals(5, five.amount);
}
```
Test fails

• The test fails because it won’t compile;

• We need to change the signature of the times method; previously it returned void and now it needs to return Dollar

    public Dollar times(int multiplier) {
        amount = amount * multiplier;
        return null;
    }

• The test compiles but still fails; as Kent Beck likes to say “Progress!”
Test Passes

• To make the test pass, we need to return a new Dollar object whose amount equals the result of the multiplication

```java
public Dollar times(int multiplier) {
    return new Dollar(amount * multiplier);
}
```

• Test Passes;

• Cross “Dollar Side Effects?” off the testing list; second loop complete!

• There was no need to refactor in this situation
Discussion of the Example

• There is still a long way to go
  • only scratched the surface

• But
  • we saw the life cycle performed twice
  • we saw the advantage of writing tests first
  • we saw the advantage of keeping things simple
  • we saw the advantage of keeping a testing list to keep track of our progress

• Plus, as we write new code, we will know if we are breaking things because our old test cases will fail if we do;
  • if the old tests stay green, we can proceed with confidence
Principles of TDD

• Testing List
  • keep a record of where you want to go;
    • Beck keeps two lists, one for his current coding session and one for “later”; You won’t necessarily finish everything in one go!

• Test First
  • Write tests before code, because you probably won’t do it after
  • Writing test cases gets you thinking about the design of your implementation;
    • does this code structure make sense?
    • what should the signature of this method be?
Principles of TDD, continued

• Assert First

  • How do you write a test case?

    • By writing its assertions first!

  • Suppose you are writing a client/server system and you want to test an interaction between the server and the client

    • Suppose that for each transaction

      • some string has to have been read from the server, and

      • the socket used to talk to the server should be closed after the transaction

    • Lets write the test case
public void testCompleteTransaction {

    ... 

    assertTrue(reader.isClosed());

    assertEquals("abc", reply.contents());

}

• Now write the code that will make these asserts possible
public void testCompleteTransaction {
    Server writer = Server(defaultPort(), "abc")
    Socket reader = Socket("localhost", defaultPort());
    Buffer reply = reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}

• Now you have a test case that can drive development
  • if you don’t like the interface above for server and socket, then write a different test case
  • or refactor the test case, after you get the above test to pass
Principles of TDD, continued

• Evident Data
  • How do you represent the intent of your test data
  • Even in test cases, we’d like to avoid magic numbers; consider this rewrite of our second “times” test case

```java
class Dollar { public static final Dollar FIVE = new Dollar(5); public static final Dollar TEN = new Dollar(10); public int amount; public Dollar(int amount) { this.amount = amount; } public Dollar times(int multiplier) { return new Dollar(this.amount * multiplier); } public void testMultiplication() { Dollar five = new Dollar(5); Dollar product = five.times(2); assertEquals(5 * 2, product.amount); product = five.times(3); assertEquals(5 * 3, product.amount); }
}

• Replace the “magic numbers” with expressions
Summary

• Test-Driven Design is a “mini” software development life cycle that helps to organize coding sessions and make them more productive
  • Write a failing test case
  • Make the simplest change to make it pass
  • Refactor to remove duplication
  • Repeat!
Reflections

• Test-Driven Design builds on the practices of Agile Design Methods
  • If you decide to adopt it, not only do you “write code only to make failing tests pass” but you also get
    • an easy way to integrate refactoring into your daily coding practices
    • an easy way to introduce “integration testing/building your system every day” into your work environment
      • because you need to run all your tests to make sure that your new code didn’t break anything; this has the side effect of making refactoring safe
    • courage to try new things, such as unfamiliar design pattern, because now you have a safety net
But how does it integrate with life cycles?

- With traditional software life cycles, TDD can be “test-driven development”
  - You’ll do requirements, use cases, class diagrams, etc. ➔ top down
  - Then TDD, coding from scratch to test your design ➔ bottom up
- With agile life cycles, TDD can be “test-driven design”
  - You create a new user story and use TDD to “discover” the classes that will help you implement that feature ➔ bottom up