Goals for this lecture

- Introduce the concept of Test-Driven Development (TDD)
- Present an example

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
Writing Test Cases First

- 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 then make the test case 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; note that this cycle simply adds testing to the “add functionality; refactor” loop of refactoring covered last week.

TDD Life Cycle, continued

- Kent Beck likes to perform TDD within 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”.

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 CHF = \$10 if rate is 2:1
  - \$5 * 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 clean up
    - 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; beginning 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?
  - Hint: its between the Dollar class and the test case
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 hook things up

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 * 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 general lessons
- So, let’s write the next test case
  - When we called the times operation our variable “five” was pointing at an about whose amount equaled “ten”; not good
    - the times operation had a side effect which was to change the value of a previous 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);
  }
  ```
- Note: the last “assert” is redundant; it is implicitly shown to be true by the second “assert”; I decided to make it explicit

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
  ```java
  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 case);

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?

Assert First

```java
public void testCompleteTransaction {
    // ...
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
```

Now write the code that will make these asserts possible

Assert First, continued

```java
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
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

---

What’s Next?

- **OO Heuristics**
  - Rules of Thumb for distinguishing between good OO designs and bad OO designs
- **Review for the Final**
- **FCQs**
  - Your chance to rate me and this course! :-)
  - I need a volunteer to pick up the FCQ forms for me and administer them at the end of class next Tuesday; anyone?