Test-Driven Development

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 5828 — Lecture 22 — 04/01/2010

Credit where Credit is Due (I)

- 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

Credit where Credit is Due (II)

- Finally, one of the examples is inspired by the Roman Numerals example that is featured in Dive into Python 3 <http://diveintopython3.org/> by Mark Pilgrim.
- The slides devoted to that example are thus distributed using the following license: <http://creativecommons.org/licenses/by-sa/3.0/>.

Side Note

- Pointer to a Podcast on the topic of Test Driven Development
  - <http://faceoffshow.com/2009/03/31/episode-10-test-driven-development/>

Goals

- Review material from Chapter 8 of Pilone & Miles
  - Test-Driven Development
    - Terminology
    - Concepts
    - Techniques
    - Tools

Test-Driven Development

- An agile practice that asserts that **testing is a fundamental part of software development**
  - Rather than thinking of testing as something that occurs after implementation, we want to think of it as something that occurs BEFORE and DURING implementation
  - Indeed, done properly, testing can DRIVE implementation
- The result, increased confidence when performing other tasks such as fixing bugs, refactoring, or reimplementing parts of your software system

Testimonial

On Monday, September 8, 2003, at 03:44 PM, a former student wrote:

> Dr. Anderson -
>
> I hope you don't mind hearing from former students :) Remember me
> from Object Oriented Analysis and Design last spring? I'm now happily
> graduated and working in the so-called 'Real World' (yikes).
>
> I just wanted to give you another testimony on the real-life use of
> test driven development. **My co-workers are stunned that I am actually**
> **using something at work that I learned at school** (well, not really,
> but they like to tease). **For a new software parsing tool I'm**
> **developing, I decided to use TDD to develop it and it is making my**
> **life so easy right now to test new changes.**
>
> Anyways, I just thought of you and your class when I decided to use
> this and I wanted to let you know.
>
> I hope that you are doing well. Best of luck on this new semester.

Test First

- The definition of test-driven development:
  - All production code is written to make failing test cases pass
- Terminology
  - Production code is code that is deployed to end users and used in their “production environments” that is there day to day work
- Implications
  - When developing software, we write a test case first, watch it fail, then write the simplest code to make it pass; repeat

Example (I)

- Consider writing a program to score the game of bowling

```
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

```
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

```
public class Game {
   public void addThrow(int pins) {
   }
   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

```
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)


```
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 we 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());
   }
…
```

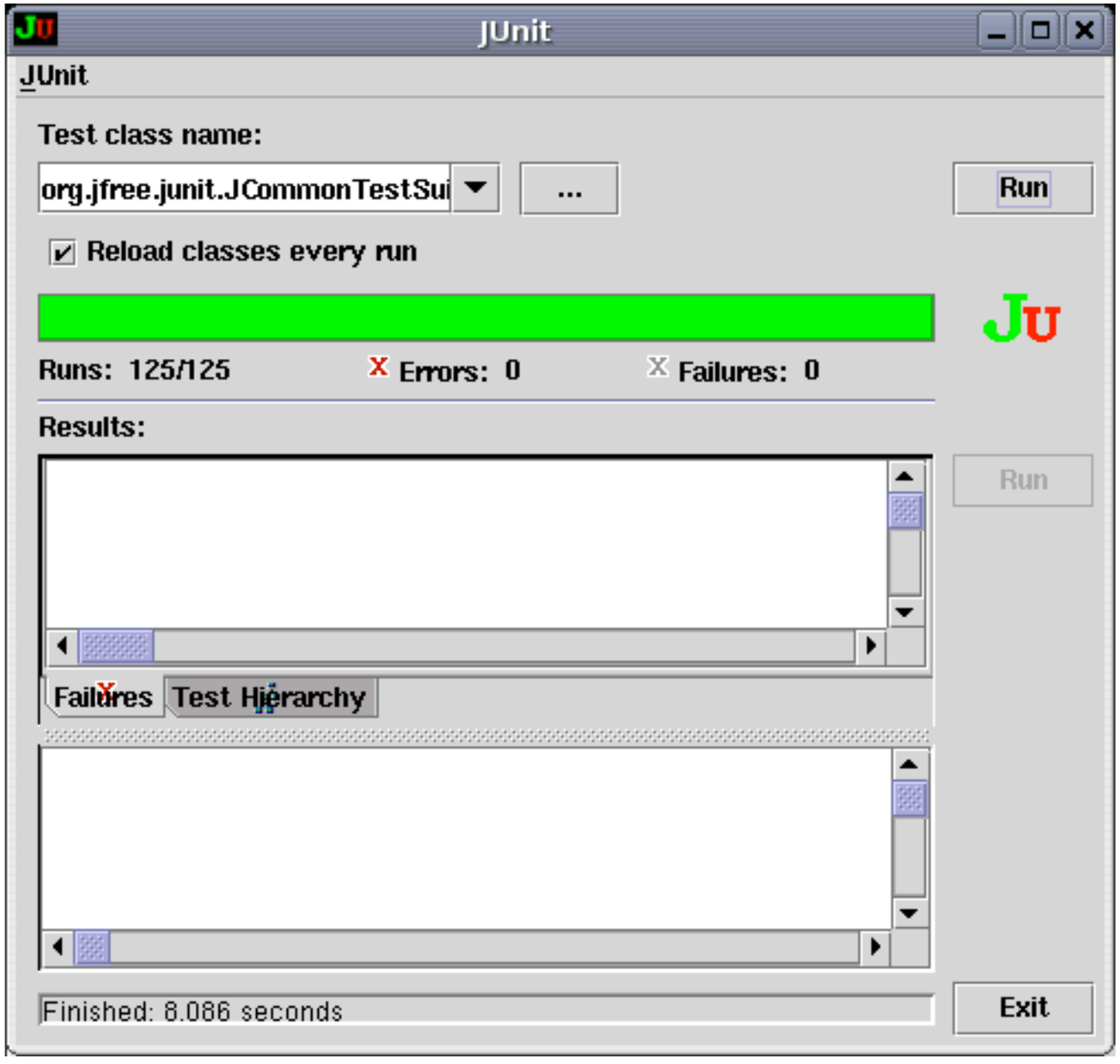
- We’re back to the code not compiling due to scoreForFrame()
  - We’ll need to add a method body for this method and give it the simplest implementation that will make all three of our tests cases pass

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 CHF = $10 if rate is 2:1
    - $5 * 2 = $10

First Test

- The first test case looks a bit complex, lets start with the second
  - 5 USD * 2 = 10 USD
- First, we write a test case

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

Discussion on Test


```
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


```
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

- Here's one way

```
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, lets connect the dots…

First version of Dollar Class


```
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 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

```
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

```
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

Roman Numerals (I)

- Let's develop a class that can manipulate roman numerals
  - Roman numerals can express integers from 1 to 3999
- They do this using the following set of symbols
  - I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000
- There are rules concerning how these characters can be combined
  - For instance, the 10s characters (X,C,M) can be repeated up to three times
  - The 5s characters (V, L, D) cannot be repeated
  - Character sequences can be additive (III = 3) or subtractive (IX = 9)
    - Can be complex 99 is written as XCIX (100-10 + 10-1)

Roman Numerals (II)

- We start by developing a testing list
  - able to convert legal roman numerals to integers
  - able to convert integers in the range 1 to 3999 into roman numerals
  - able to add two roman numerals, checking for boundary conditions
  - able to subtract two roman numerals, checking for boundary conditions
- We will not complete the example but we'll make progress on a few of these

Test Case: Create a

- Let's use Python's Unit Test framework
  - We write the test case as if all the code we need is available

```
1  import roman
2  import unittest
3
4  class TestRomanNumerals(unittest.TestCase):
5
6    def testCreateAndGetValue(self):
7      thousand = roman.RomanNumeral("M")
8      self.assertEqual(thousand.value(), 1000)
9
10 if __name__ == "__main__":
11   unittest.main()
12
```

Several Failures on the Path to Green

- module import fail: no file named roman.py ➟ create one
- no class called RomanNumeral ➟ create one
- wrong number of arguments for constructor ➟ add self and value arguments
- no method called value() ➟ create a “blank” one
- test now runs and reports failure!! ➟ write simplest code to make it work
- test passes but contains duplication ➟ add another test case to make it fail
  - end of step 2, onto step 3 directory
- original test passes, but new test fails ➟ write simplest code to make it work
  - note, because of the tests, this is no longer trivial code to write

Making Progress; But Long way to go

- We now have a class that can successfully handle Roman Numerals that consist only of “M” characters
  - We haven’t fully completed any of the items on our test list
- We have lots of different directions we could go in
  - Add tests to check that we handle bad input
  - Add tests to add support for other roman numeral characters
  - Add tests to add basic support for addition or subtraction
  - etc.
- Let’s focus on bad input to see the test-code-refactor loop one more time

Test Case: Handle Bad

- Let's add test cases that handle
  - wrong input types (being handed a number or array rather than a string)
  - wrong values (producing a value that is outside the legal set of values)
- Then, we'll add a test case that can handle basic addition

Several Failures on the Path to Green (Again)

- add test case to handle non-string args to the constructor
  - Here we want to give it bad input and see if it raises an exception
    - All such tests will currently fail since the constructor just accepts whatever it is given
  - Start by passing a number, check to see if it raises an exception ➟ fail
  - Add code to check for int ➟ pass; now pass collection ➟ fail
    - Make it pass but then erase code written so far and now write code to raise exception whenever a non-string is passed
      - This is the refactor step, as we were adding duplication based on the types of the parameters passed in between code and test case
- End of step 4; now make sure that we test the contents of the string
  - accept “M”, “MM”, and “MMM” for now, all else should fail

Test Case: Handle Addition

- All we'll be able to do is handle 1000 + 1000 and 1000 + 2000
  - but this will ensure that we've got the basics in place
    - can handle correct additions
    - can flag additions that produce numbers outside the legal range
- Getting to Green
  - Add a sum method that follows the "value" pattern seen above
  - Generates ValueError if the value goes outside of the legal range
  - First a test case to handle an illegal addition
  - Then a test case to handle a legal addition
    - We'll encounter familiar steps
      - fails because there is no sum method
      - fails because it doesn't throw an exception
      - etc.

End of Example

- Still a long way to go, but you should now have the feel of what test-driven development is like
  - Start with a system that needs a new feature
  - Write a test that documents what the expected results of the feature are
  - Add simplest code to make test pass
  - Make test more complicated, or add new test to reveal duplication
  - Once duplication is found, refactor to produce general code
  - Loop until feature is implemented and all tests pass

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

Assert First


```
public void testCompleteTransaction {
   …
   assertTrue(reader.isClosed());
   assertEquals(“abc”, reply.contents());
}
```

- Now write the code that will make these asserts possible

Assert First, continued


```
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

```
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

TDD in our Book

- Largely follows what I've presented above
  - Rule 1: Watch tests fail before you implement code
  - Rule 2: Implement the simplest code possible to make the test pass
    - You add more tests to make the code evolve
  - Life Cycle: Red, Green, Refactor
- But also adds a few new points…

Tests Drive Implementation

- Each test should verify only one thing
  - Why is this important?
- Avoid duplicate test code
  - Testing takes time; don't waste it by running the same test twice!
  - Use setup and teardown methods in testing frameworks to eliminate redundant initialization/finalization code
- Keep your tests in a MIRROR directory of your source code
  - src/ and test/ become top-level folders in your project dir.

TDD and Task Completion

- A task can be declared complete when all of its associated tests pass
  - How many tests are needed?
    - As discussed last time you need a criteria for knowing when you are done
      - Have you covered all of the functionality associated with the task?
      - If you're doing code coverage, have you achieved your target percentage for statement and branch coverage?

TDD: client perspective

- Writing tests first lets you work on specifying the API of the classes involved in the test
  - OrderInfo info = new OrderInfo()
  - info.setCustomerName(“Dan”)
  - …
  - Receipt r = orderProcessor.process(info);
  - assertTrue(r.getConfirmationNumber() > 0)

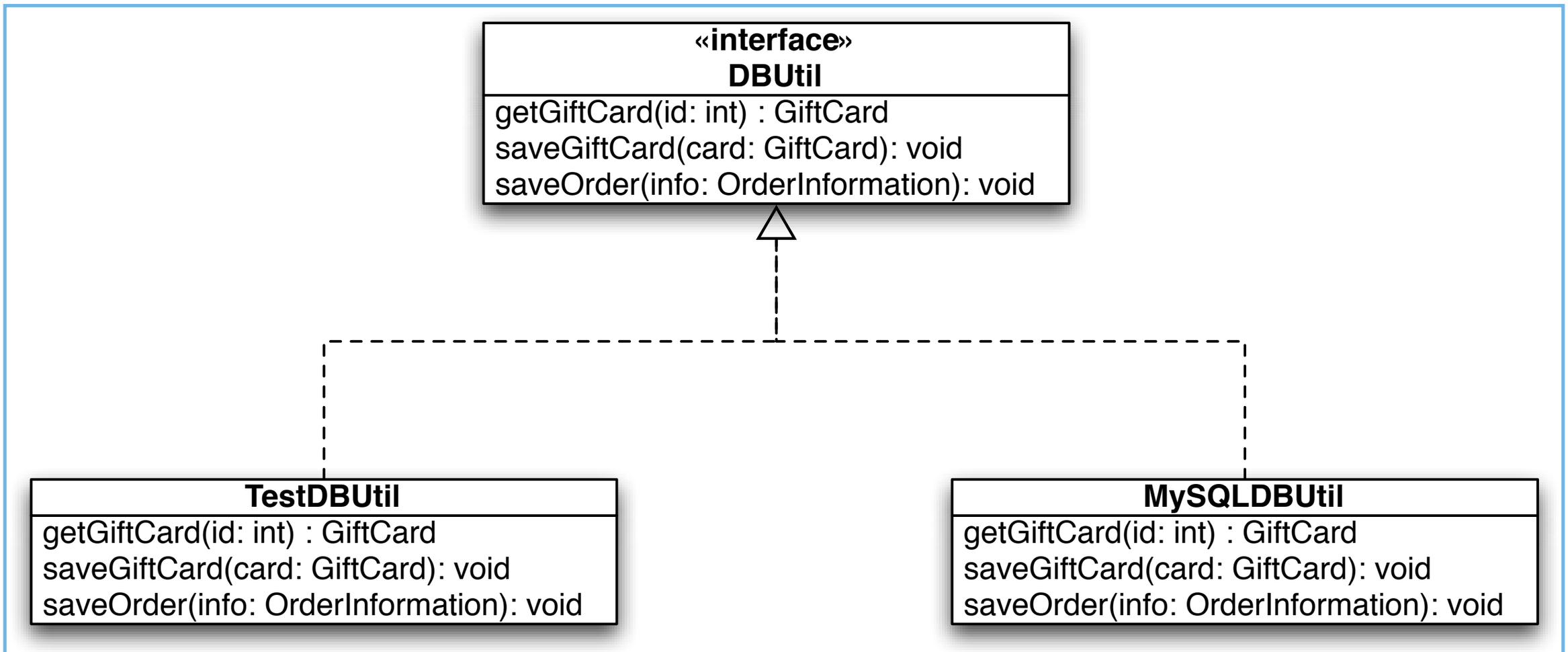
TDD: tests across tasks

- Occasionally you will be in a situation in which you need to write tests that will require you to access code associated with a different task
  - If that other task has not yet started, the code will not exist
- Should we give up in such a situation?
  - No! This is an opportunity to design the API of those classes while making progress on the current task

Accessing a DB

- In the textbook, the developers need to access the DB while working on the task that handles order processing
  - They decide to simulate DB access with a TestDBUtil class
  - When they switch to working on the task associated with creating the real DB, they'll write a "real" DBUtil class
- Note: the TestDBUtil class does not belong in the src/ directory of your project; its code that will only be used by tests, so it should live under the test/ dir.

Strategy Pattern (one part of it)



Simulates DB; returns objects with “dummy” data

Talks to DB; returns “real” objects

TDD leads to better code

- TDD not only leads to more tests that help us find faults in our code, it also
  - produces better organized code:
    - production code in one place, testing in another
    - packages and classes are designed from a client perspective
  - produces code that always does the same thing
    - Avoids the “if (debug) {}” trap
  - Loosely coupled code
    - Encourages the creation of highly cohesive and loosely coupled code because that type of code is easier to test!

More tests always means more code

- The original version of XP
  - had 10 million lines of production code;
  - had 15 million lines of test code!
- The book however now discusses “corner cases”
  - testing not only the success case but all the ways a particular function might fail;
  - this, in turn, leads to lots of different objects that are similar but do slightly different things (to test different cases)
- This leads to a discussion of “mock objects”; see book for details

Things to Avoid

- Not using a criteria to determine when you are “done”
  - You need to be systematic if you want to ensure that you cover all the cases associated with a particular function
- Not using real data
  - When testing, you’ll sometimes create data to test the system; that’s good but you need to make sure you test your system on realistic data (perhaps received from the customer)
- Forgetting to clean up after yourself: “ghosts from the past”
  - Need to make sure that results from previous tests are not influencing the results of tests that come after

Wide Applicability

- Unit Tests can be created in lots of different contexts
  - GUIs, Web services, Javascript, embedded software, etc.
- Even, performance…
  - You can unit test performance in a number of ways
    - Examine spec for performance constraints
    - Time individual methods, classes, modules, subsystems
    - Make an assertion that elapsed time is less than or equal to the time specified in the spec.
    - Or, create a timer and start it, run code and cancel timer; if timer goes off, assert(false) to trigger test failure

Wrapping Up

- Development Techniques
  - Write tests first, then code to make those tests pass
  - After they pass, look for duplication between test code and production code; refactor the latter to eliminate duplication while ensuring that tests still pass
- Development Principles
  - TDD forces you to focus on functionality; “client” perspective
  - Automate your tests to make refactoring safer
  - Covering all of your functionality leads to code coverage

Coming Up

- Lecture 23: Safety & Liveness Properties
  - Read Chapter 7 of the Concurrency textbook
  - May also move on to Chapter 8 in that lecture
- Lecture 24: Ending an Iteration
  - Read Chapter 9 of Head First Software Development