Test-Driven Development

Kenneth M. Anderson
University of Colorado, Boulder

CSCI 5828 — Lecture 21 — 03/31/2009

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

Side Note

- Pointer to Recent 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());
   }
…
```

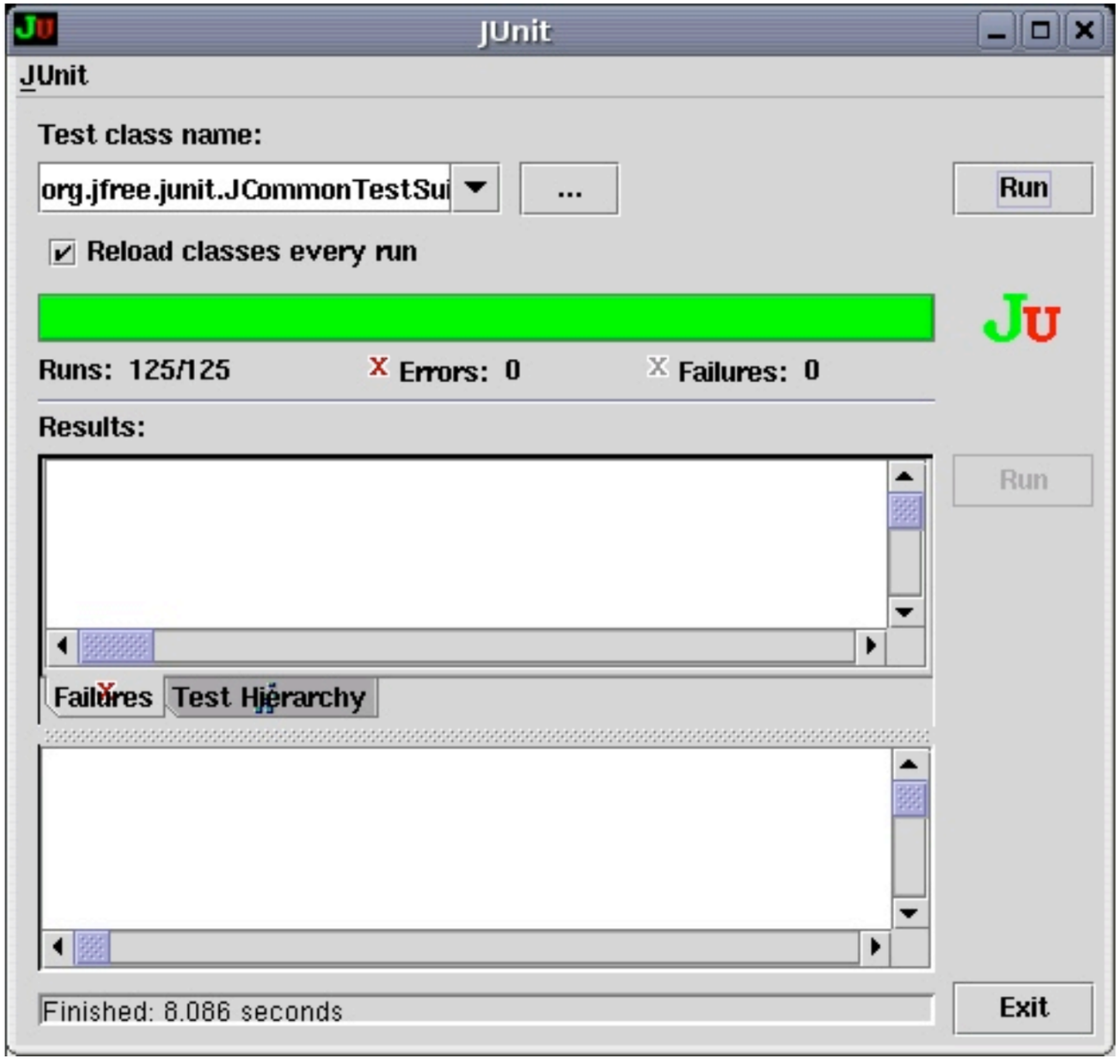
- We’re back to the code not compiling due to scoreForFrame()
  - We’ll need to add a method body for this method and give it the simplest implementation that will make all three of our tests cases pass

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

Demo

- TDD of Fibonacci Generator
  - 0, 1, 1, 2, 3, 5, 8, …
- This is a simple example
  - you can find longer examples in TDD books and on the web

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)

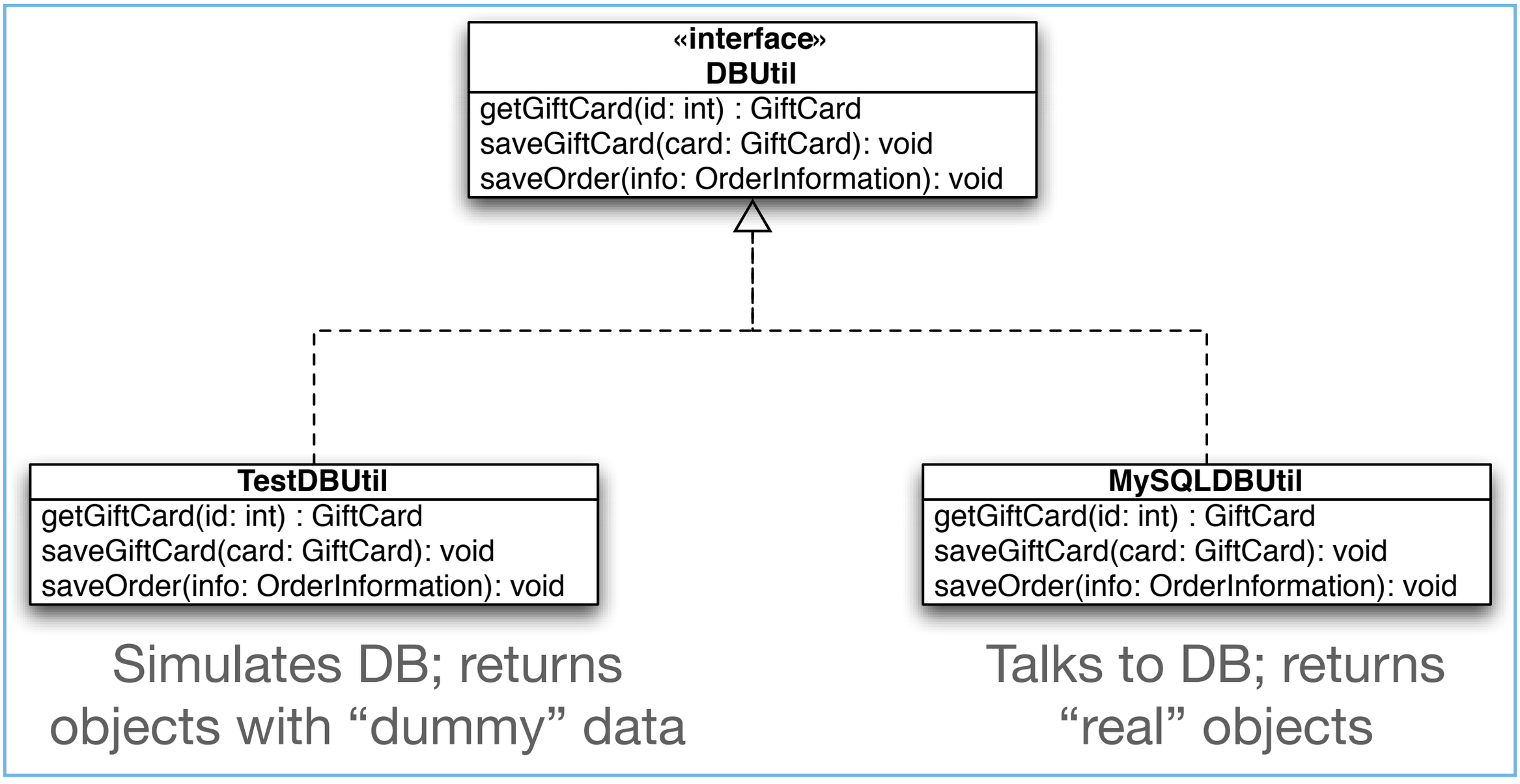
TDD: tests across tasks

- 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
    - And they'll use the strategy pattern to do it
  - 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)



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

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