SOFTWARE TESTING

Content from Chapter 8 of "Head First Software Development", Pilone et al.

Miami University Software Technology & Analysis Group (MUSTANG)
Computer Science & Software Engineering
Miami University, Oxford, Ohio, USA

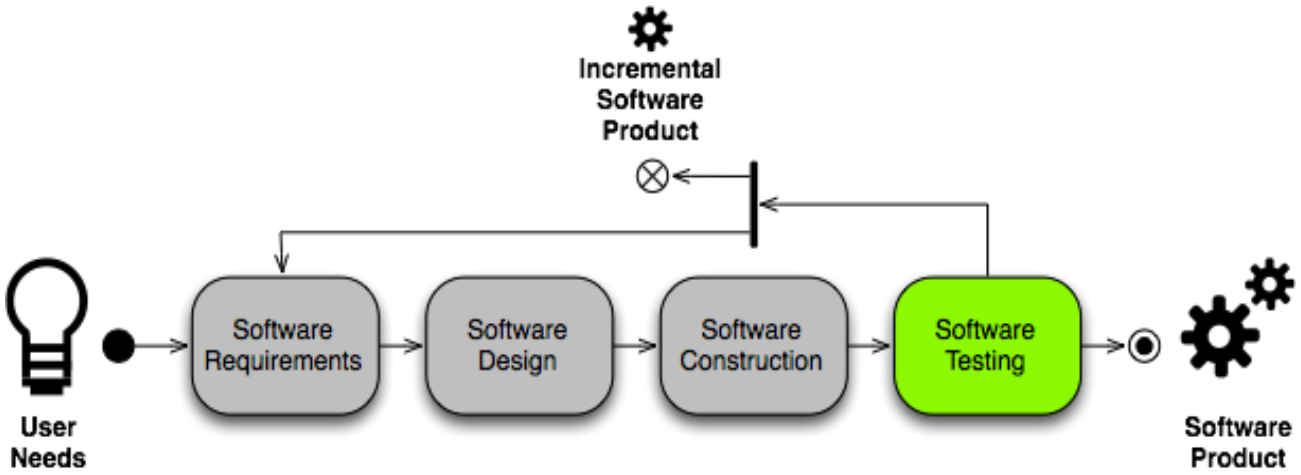
ADMIN

- Midterm Redo Problem Statement
  - Posted under Canvas. 1 week to work on it – Monday 3/29 midnight.
- Wellness Day
  - Thursday groups need to reschedule their meetings.
- Project Week 4 deliverables
  - March 26th instead of April 2nd
- Iteration #1 Demo
  - March 26th
  - Presenting product up to this point (Week 3 and 4 deliverables)
  - Iteration 1 retrospective
  - Burndown and updates
  - Rubric online

Test Driven Development

TRADITIONALLY…

- We code first then write tests that satisfy that code.



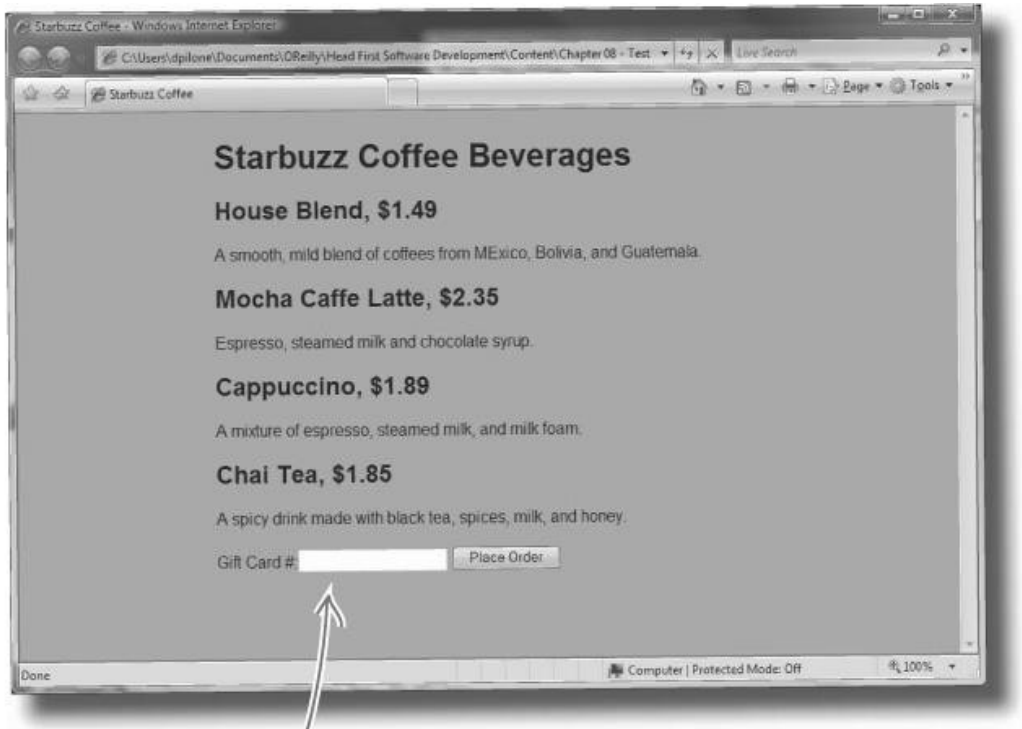
TRADITIONALLY … CODE FIRST?

What If …

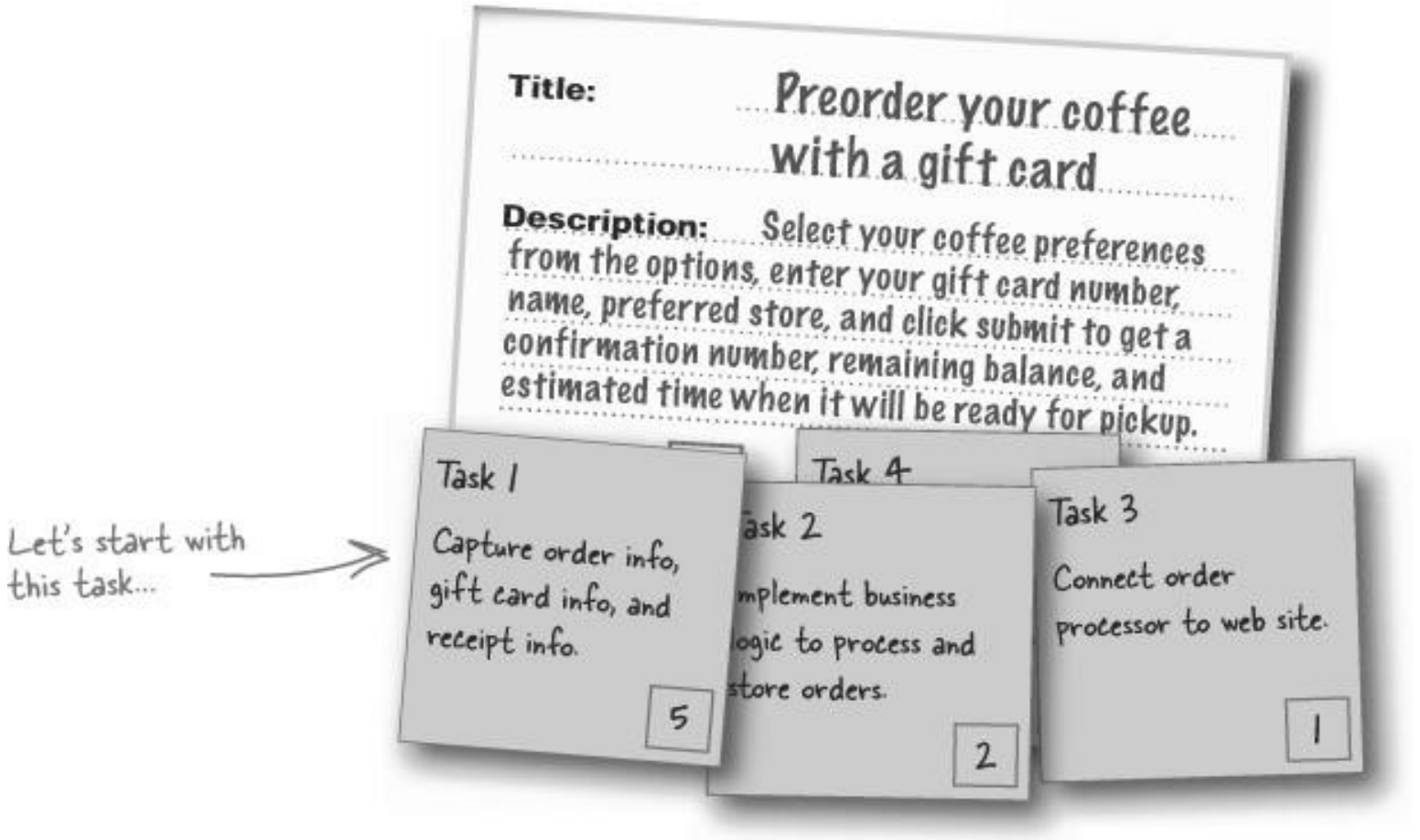
- Take “Testing Mentality” to the extreme.
- Testing as a fundamental part of Software Development
- Write our tests first!

TEST FIRST, NOT LAST

- Look at a project from the ground up.
- Starbuzz coffee sells gift cards.
  - Now need a way to accept gift cards as payment
  - Customer already knows how the page should look.



BREAKDOWN INTO LOWER LEVELS



TEST FIRST

Breakdown Each Task – What Objects / Concepts are we going to want to represent?

- The order information
- Gift card information
- Receipt information

Breakdown Each Task

- The order information
  - Customer's name, drink description, store #, and gift card #
- Gift card information
  - Activation date, expiration date, and balance.
- Receipt information
  - Confirmation number, pickup time, and remaining balance on a gift card.

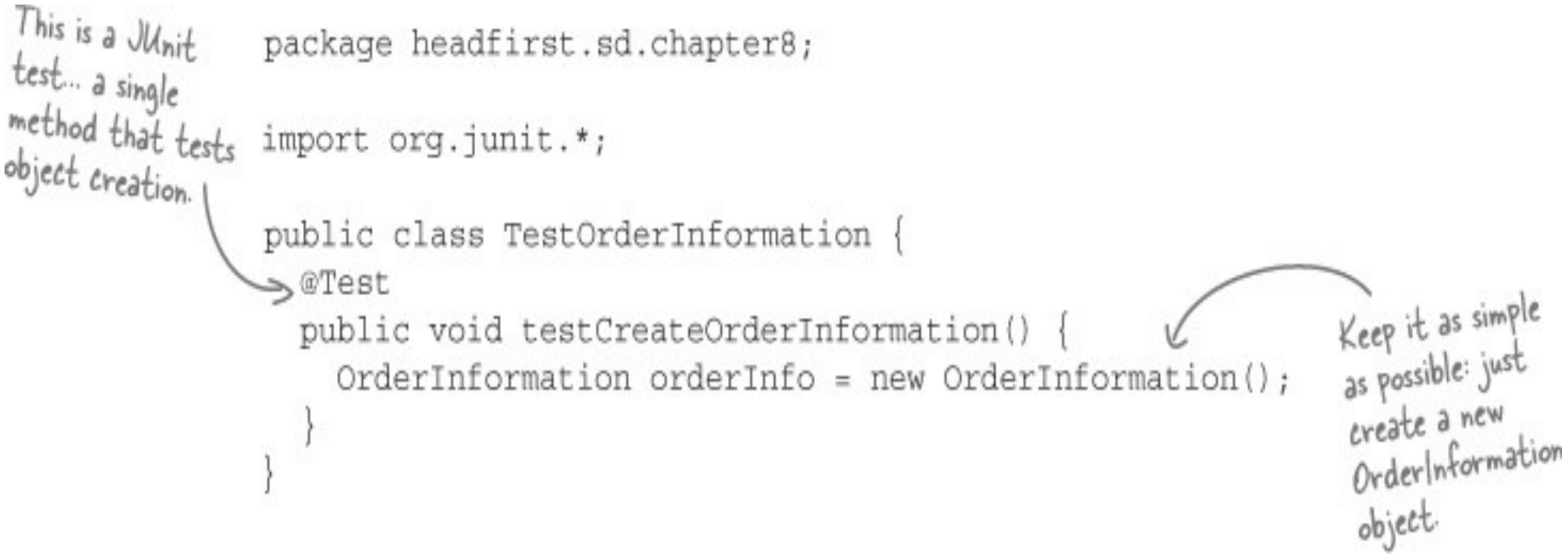
Time For TDD

- Let the tests drive our code
- Testing from the outset
- Writing each line of code specifically as a response to tests

Our First Test for Representing Order Info.

- What exactly are we testing?
- Unit testing ➔ start small
- Smallest test we could write?

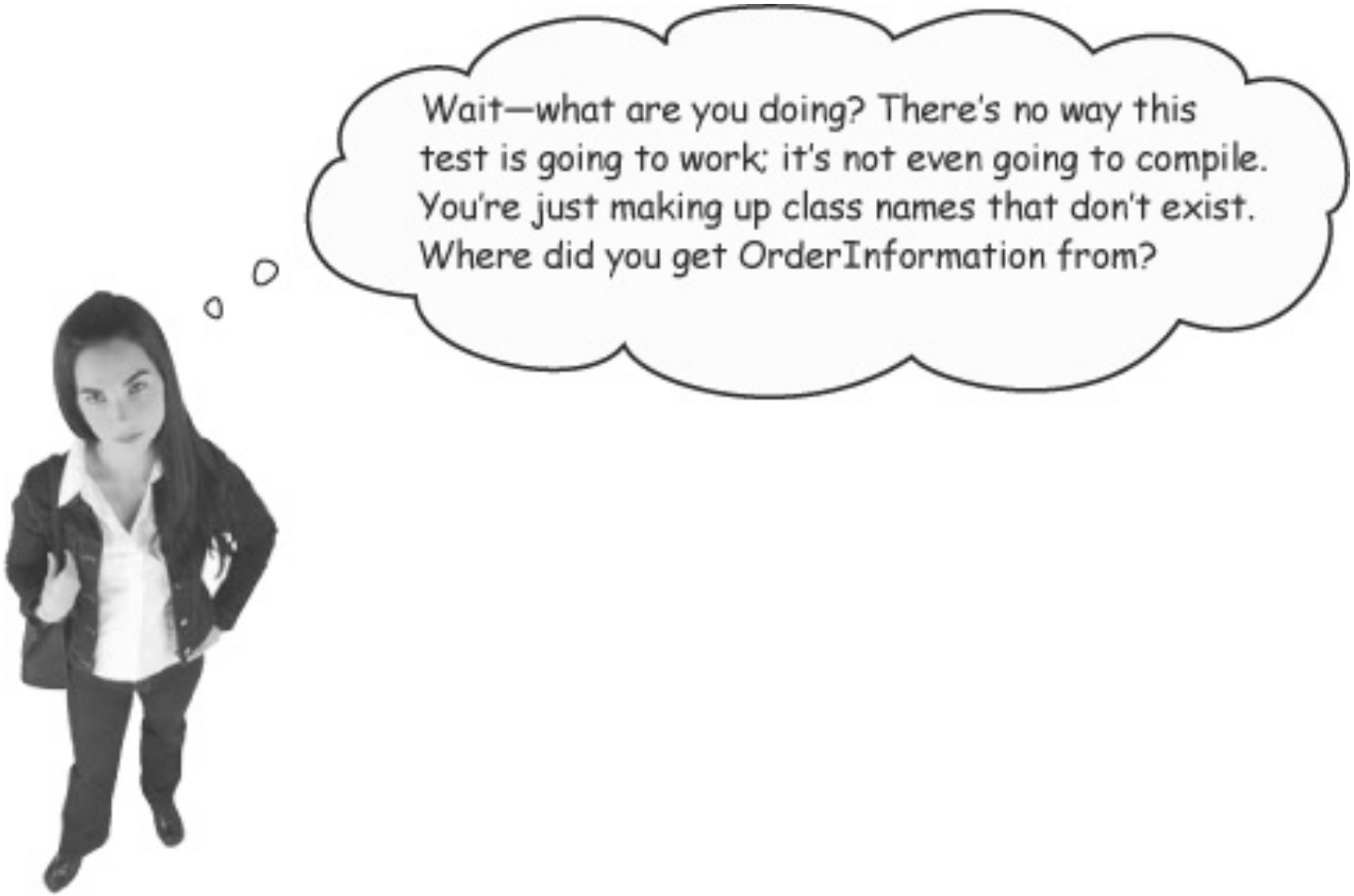
OBJECT CREATION



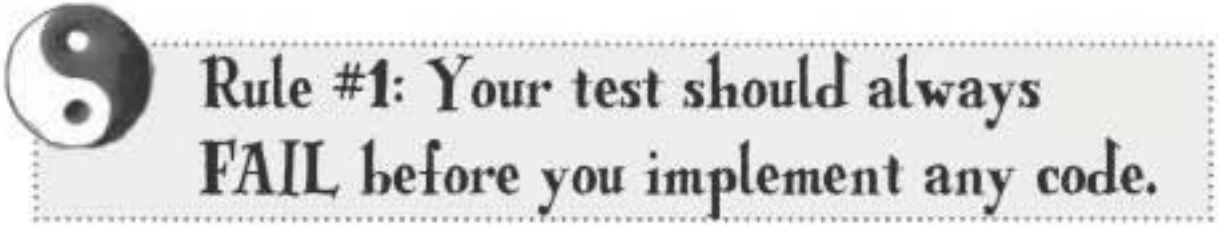
- What do you notice?

WAIT A MINUTE …

- This won't even compile
- Making up class names that don't exist.


Wait—what are you doing? There's no way this test is going to work; it's not even going to compile. You're just making up class names that don't exist. Where did you get OrderInformation from?

THAT'S CORRECT!



- Writing tests first!
- No code.
- Test won't even compile.
- We want our tests to fail when we first write them!
  - Establishing a measurable success
  - In our example, that will include "OrderInformation" object
- Now clear what we have to do to make sure the test passes.

Now, It's Time To Code

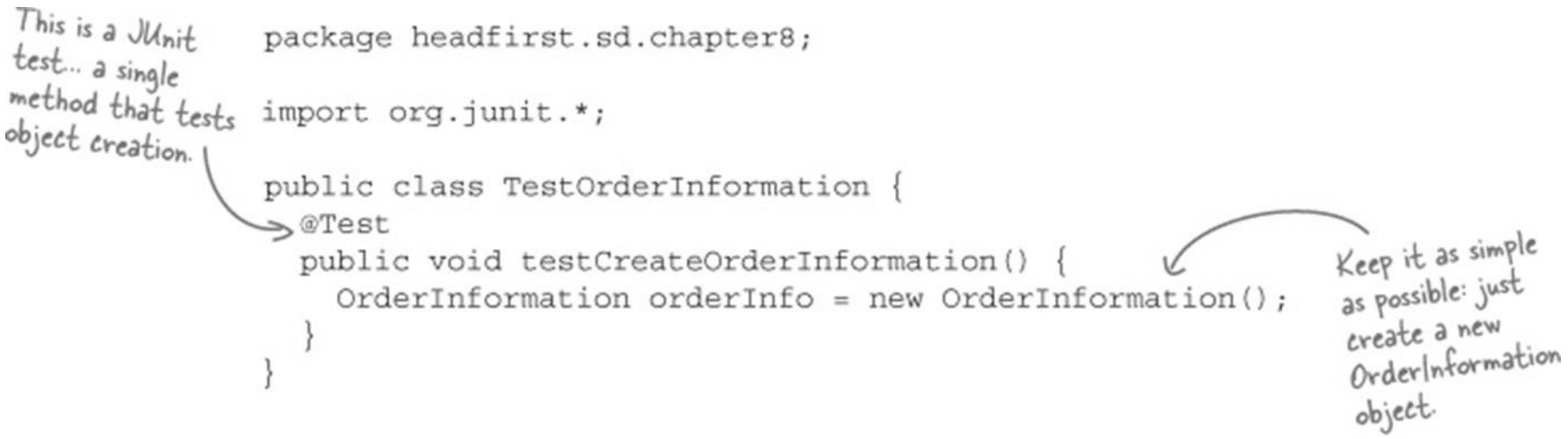
- Before going further by writing more tests or working on the task, write the **simplest** code possible to get the test to compile.

```
File  Edit  Window  Help
hfsd> javac -cp junit.jar
          headfirst.sd.chapter8.TestOrderInformation.java
TestOrderInformation.java:8: cannot find symbol
symbol  : class OrderInformation
location: class headfirst.sd.chapter8.TestOrderInformation
    OrderInformation orderInfo = new OrderInformation();
    ^
TestOrderInformation.java:8: cannot find symbol
symbol  : class OrderInformation
location: class headfirst.sd.chapter8.TestOrderInformation
    OrderInformation orderInfo = new OrderInformation();
                                 ^
2 errors
hfsd>
```

Running our first test isn't even possible yet; it fails when you try to compile.

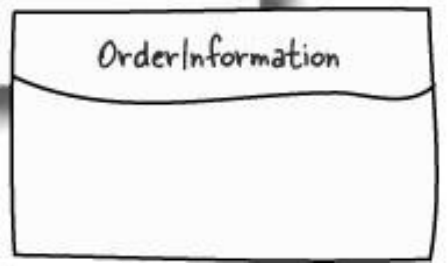
What Would You Do?

- What is the simplest thing we can do to get this test to pass?



GETTING OUR TESTS TO "GREEN"

```
public class OrderInformation {

}
```



Here's the UML for the new class. No attributes, no methods—just an empty class.

```
File Edit Window Help Classy
hfsd> javac -d bin -cp junit.jar *.java

hfsd> java -cp junit.jar;.\bin org.junit.runner.
JUnitCore headfirst.sd.chapter8.TestOrderInformation
JUnit version 4.4
.
Time: 0.018

OK (1 test)

hfsd>
```

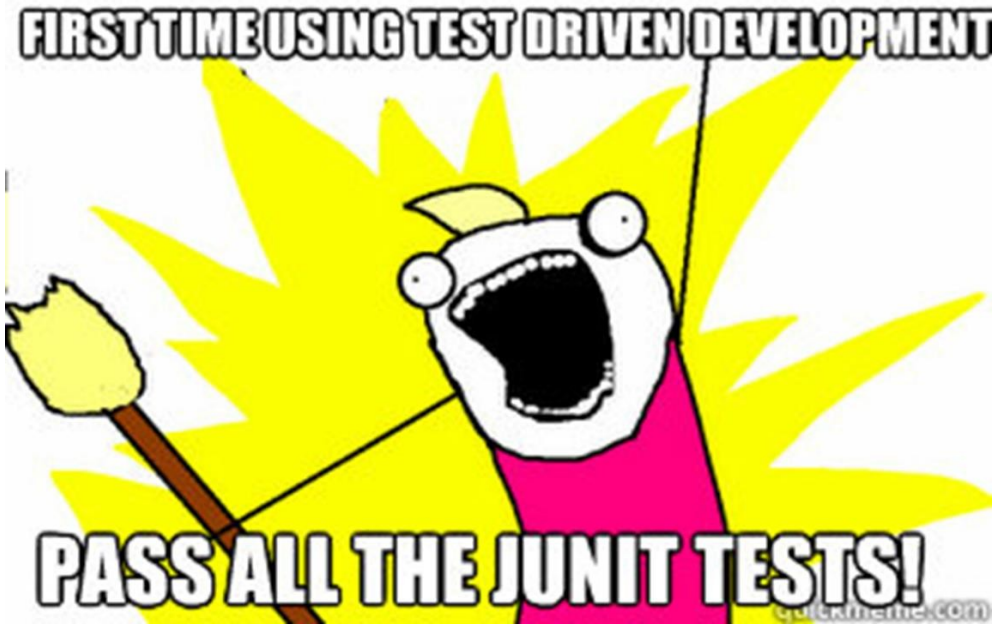
SUCCESS!

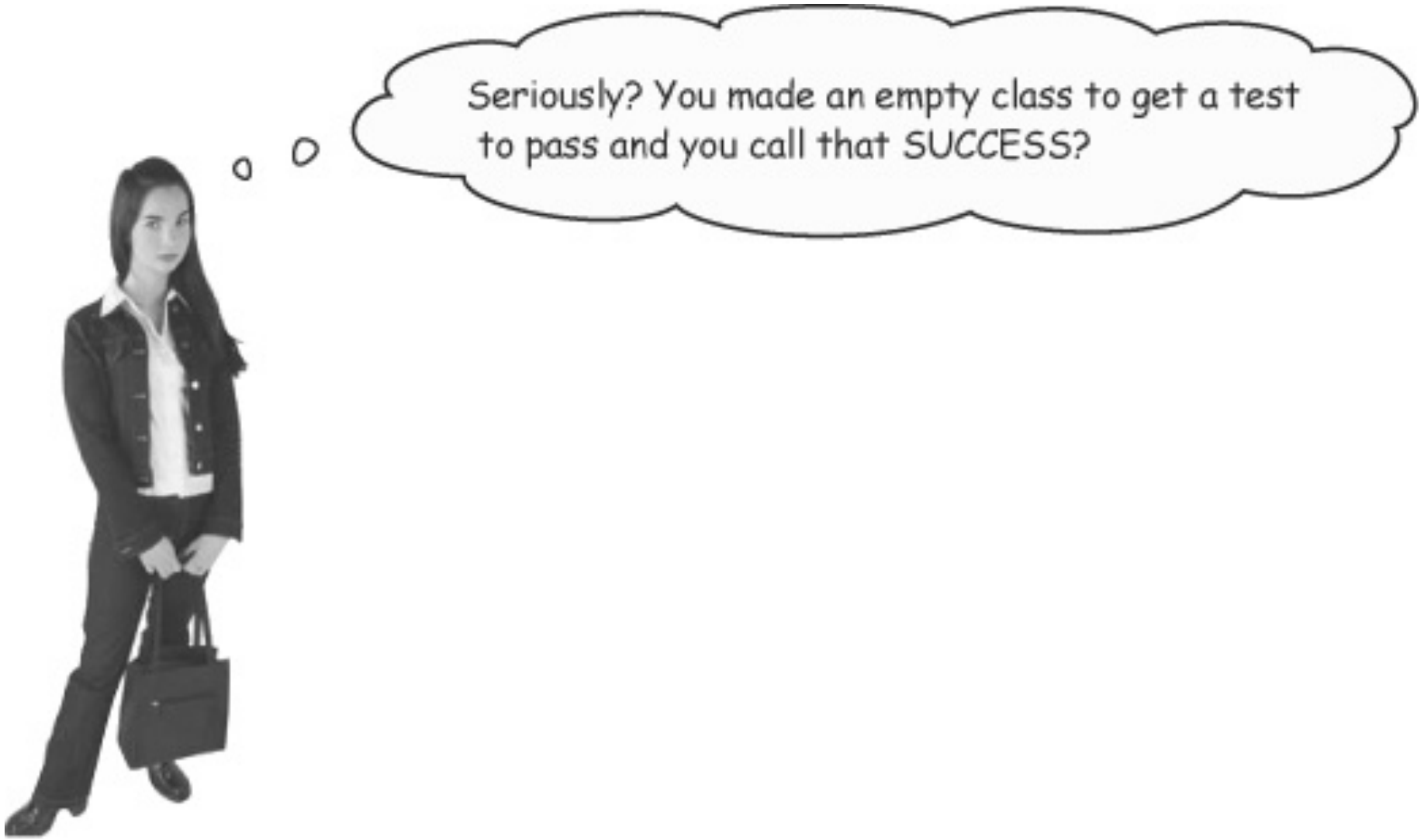
The test compiles now, as does the OrderInformation class.

Rule #2: Implement the SIMPLEST CODE POSSIBLE to make your tests pass.

Now What?

- Ready to write next test
  - Still focus on our first task!
- Finished the first round of TDD.




Seriously? You made an empty class to get a test to pass and you call that SUCCESS?

K.I.S.S. (?)

- Resist the urge to add anything you might need in the future
- If you need it later, you'll write a test then and the corresponding code after
- Obviously we can't stop after our first round…
- Focusing on small bits of code is the heart and soul of TDD!
- Also, this is the YAGNI principle...You Ain't Gonna Need It.

TDD Cycle – Red, Green, Refactor

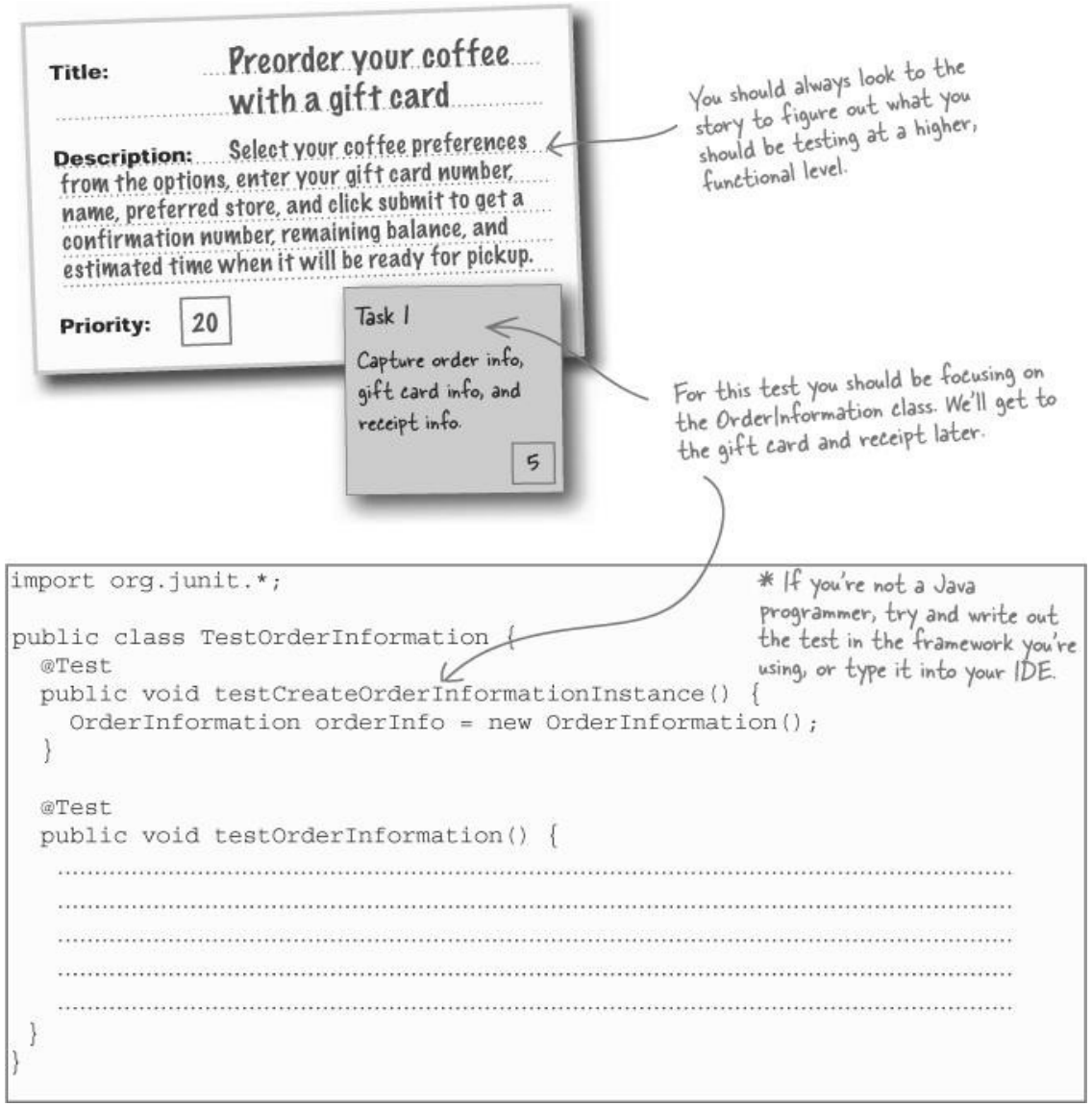
Red: Write a test that fails

Green: Implement functionality
to make test pass. Simplest code

Refactor: Clean up code,
duplication, ugliness, old code, etc.

Move on to next test, continue cycle

Next Test


Title: Preorder your coffee with a gift card
Description: Select your coffee preferences from the options, enter your gift card number, name, preferred store, and click submit to get a confirmation number, remaining balance, and estimated time when it will be ready for pickup.
Priority: 20
Task 1
Capture order info, gift card info, and receipt info.
5
You should always look to the story to figure out what you should be testing at a higher, functional level.
For this test you should be focusing on the OrderInformation class. We'll get to the gift card and receipt later.
* If you're not a Java programmer, try and write out the test in the framework you're using, or type it into your IDE.
import org.junit.*;
public class TestOrderInformation {
@Test
public void testCreateOrderInformationInstance() {
OrderInformation orderInfo = new OrderInformation();
}
@Test
public void testOrderInformation() {
}
}

NEXT TEST

To get the rest of the OrderInformation class together, you need to add coffee preference, gift card number, customer name, and preferred store to the order information.

```
import org.junit.*;

public class TestOrderInformation {
  @Test
  public void testCreateOrderInformationInstance() { // existing test }

  @Test
  public void testOrderInformation() {
   OrderInformation orderInfo = new OrderInfomation();
   orderInfo.setCustomerName("Dan");
   orderInfo.setDrinkDescription("Mocha cappa-latte-with-half-whip-skim-fracino");
   orderInfo.setGiftCardNumber(123456);
   orderInfo.setPreferredStoreNumber(8675309);
   assertEqual(orderInfo.getCustomerName(), "Dan");
   assertEqual(orderInfo.getDrinkDescription(),
     "Mocha cappa-latte-with-half-whip-skim-fracino");
   assertEqual(orderInfo.getGiftCardNumber(), 123456);
   assertEqual(orderInfo.getPreferredStoreNumber(), 8675309);
  }
}
```

Our test simply creates the OrderInformation, sets each value we need to track, and then checks to make sure we get the same values out.

You might want to use constants in your own code, so you don't have any typos beween setting values and checking against the returned values (especially in those long coffee-drink names).

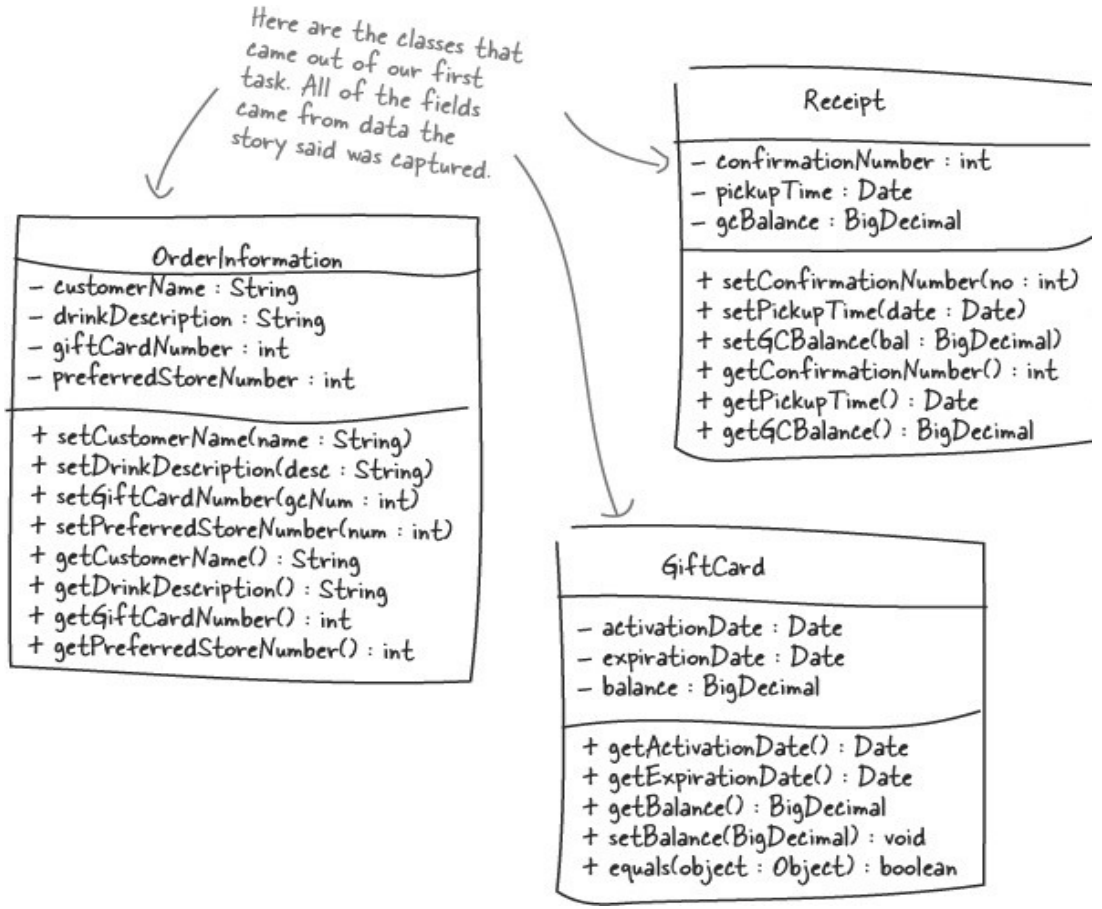
Implement Code For That Test

Good Practices

- Each test should verify only one thing
  - Keep tests straightforward and focused
  - Checks only one single thing
    - Our constructor tested created a new order
    - Second test verified multiple methods but tested one piece of functionality (Stores the right information)
- Avoid duplicate test code
  - Some testing frameworks have setup and teardown code
  - Mock-up test objects (?)
- Keep tests in a MIRROR DIRECTORY of source code
  - Keep tests in separate directory called tests at the same level of source
  - Helps avoid problems of directory to package names

When Is A Task Complete?

- When all the tests we need are present and pass



Move On To Next Task

REPEAT TDD CYCLE – RED, GREEN, REFACTOR

Red: Write a test that fails

Green: Implement functionality
to make test pass. Simplest code

Refactor: Clean up code,
duplication, ugliness, old code, etc.

Move on to next test, continue cycle

Simplicity Means

- Avoiding dependences
- What if we have a task that involves another piece of functionality?

ISSUE?

- Need a method to talk to the database, but the database access is part of another task haven't dealt with yet…
- Should we write the database access code?


And our problem is that the code for this task is all tied up with other tasks, and with database code, right?

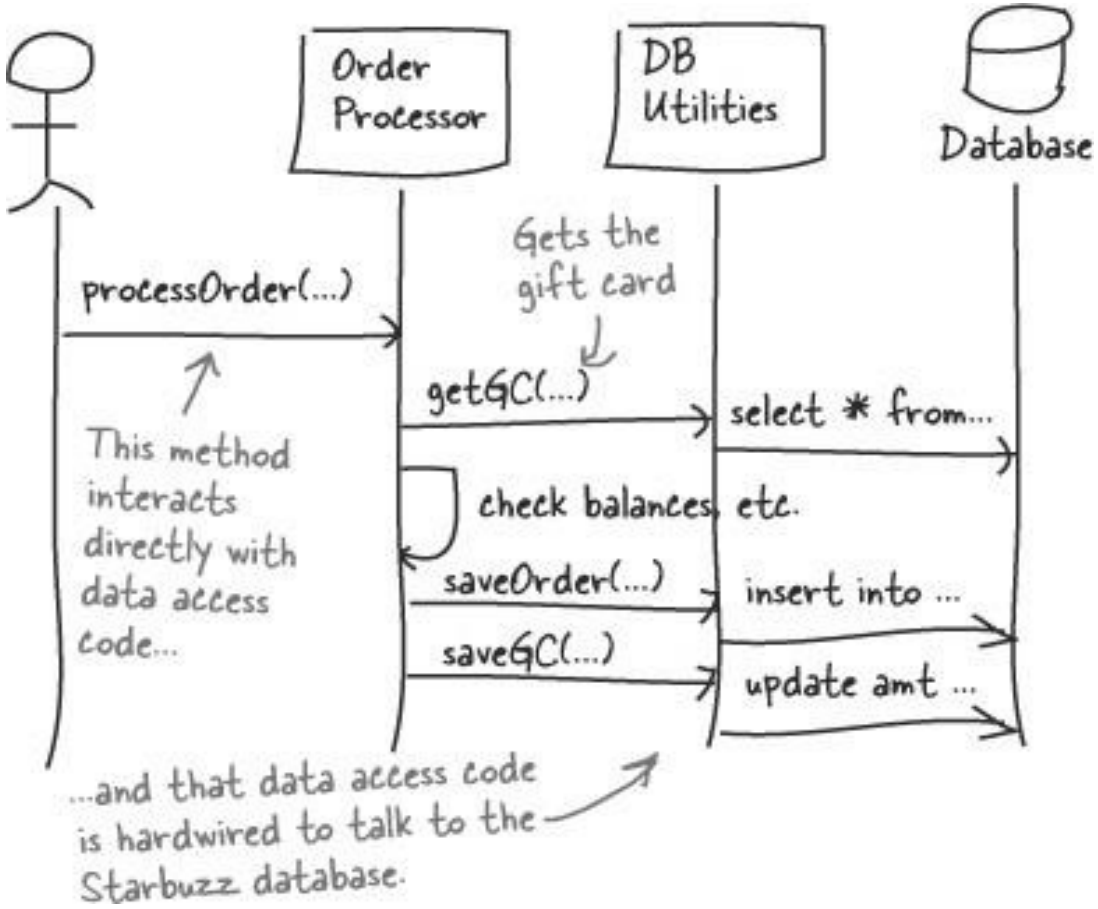
Dependencies & Coupling

- All real world code has dependencies
- When things get hard to test, examine your design

- Look for high coupling and try to fix!

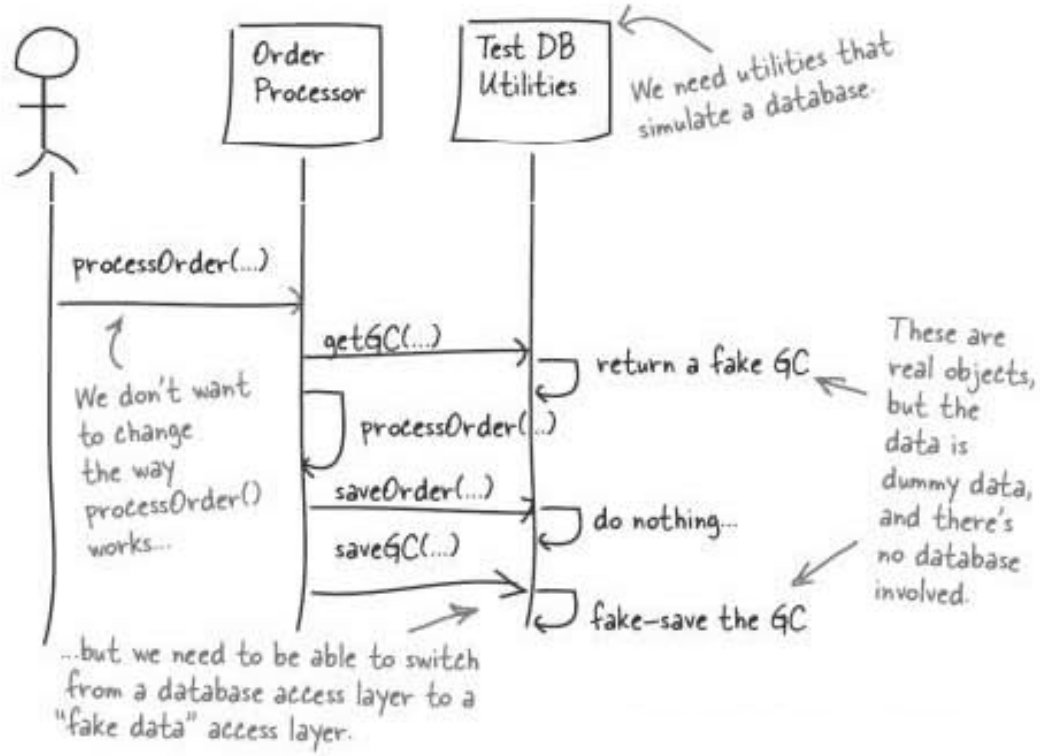
Dealing With Dependencies

- Attempt to remove them
- See if components are tightly coupled, or interdependent

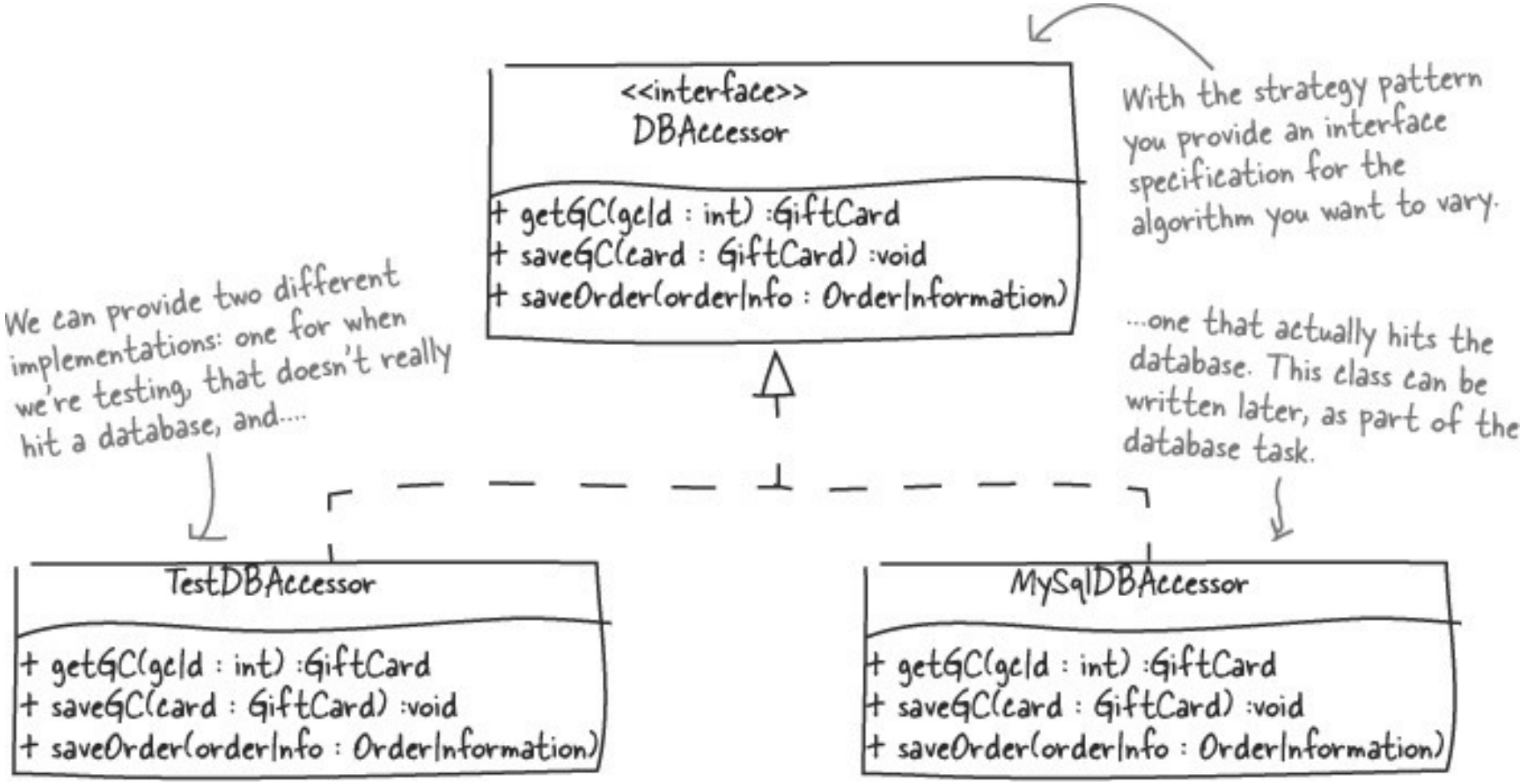


NOTHING TO REMOVE. NOW WHAT?

- Need a way to get data without requiring a database
- What about a “fake data access layer”?

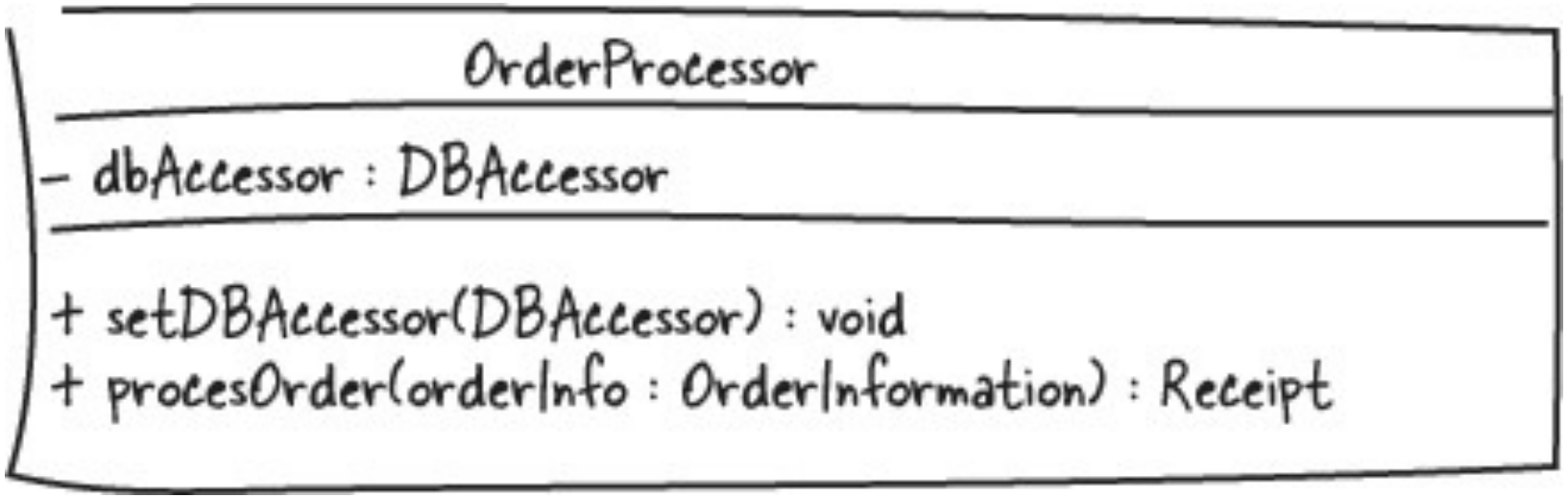


ALLOW FOR MULTIPLE IMPLEMENTATIONS

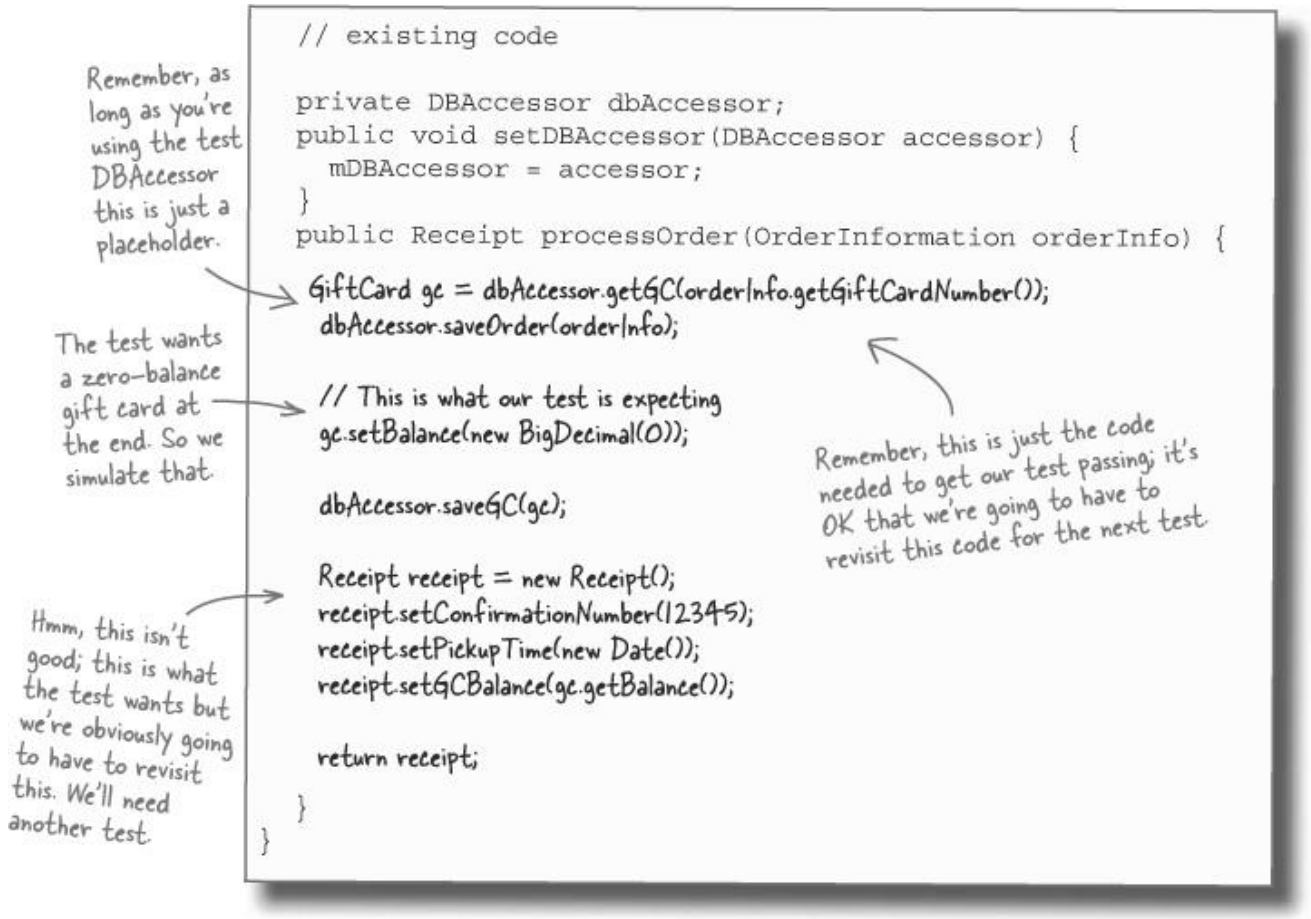


What Does That Do?

- Allows our method to use either approach, without caring which one is being used.
- Just talks to the interface. Details are hidden!



ORDERPROCESSOR IS ISOLATED FROM DATABASE



MAKE SURE TO SEPARATE TEST CODE

- Keep it with test code, not production code

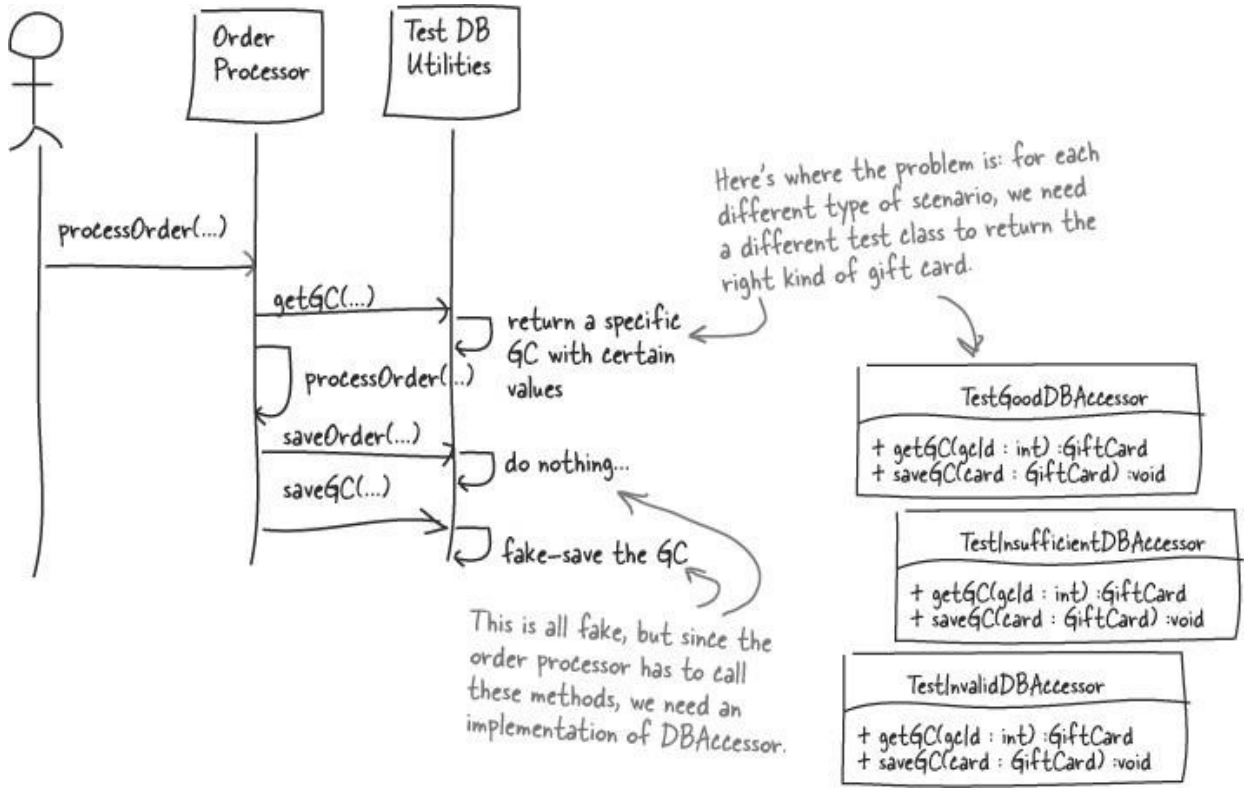
```
public class TestOrderProcessing {                    ← All this code is in our test class, which is in a separate directory from production code.
  // other tests

  public class TestAccessor implements DBAccessor {   ← Here's a simple DBAccessor implementation that returns the values we want.
    public GiftCard getGC(int gcId) {
      GiftCard gc = new GiftCard();
      gc.setActivationDate(new Date());
      gc.setExpirationDate(new Date());
      gc.setBalance(new BigDecimal(100));
    }
    // ... the other DBAccessor methods go here...
  }                                                    ← Since this is only used for testing, it's defined inside our test class.

  @Test
  public void testSimpleOrder() {
    // First create the order processor
    OrderProcessor orderProcessor = new OrderProcessor();
    orderProcessor.setDBAccessor(new TestAccessor());  ← Set the OrderProcessor object to use the test implementation for database access—which means no real database access at all.

    // Then we need to describe the order we're about to place
    OrderInformation orderInfo = new OrderInformation();
    orderInfo.setCustomerName("Dan");
    orderInfo.setDrinkDescription("Bold with room");
    orderInfo.setGiftCardNumber(12345);
    orderInfo.setPreferredStoreNumber(123);

    // Hand it off to the order processor and check the receipt
    Receipt receipt = orderProcessor.processOrder(orderInfo);   ← With the testing database accessor, we can test this method, even without hitting a live database.
    assertNotNull(receipt.getPickupTime());
    assertTrue(receipt.getConfirmationNumber() > 0);
    assertTrue(receipt.getGCBalance().equals(0));              ← Remember, this was all about the simplest code possible to return the expected values here.
  }
}
```

Testing Produces Better Code

- Well organized code
  - Production code & testing code = separate
- Code that always does the same thing
  - Always writing production code
  - Traditional may result in code that does one thing in testing but another in production
- Loosely coupled code
  - More flexible system
  - Low coupling and high cohesion (database and logic code is concentrated into separate classes)

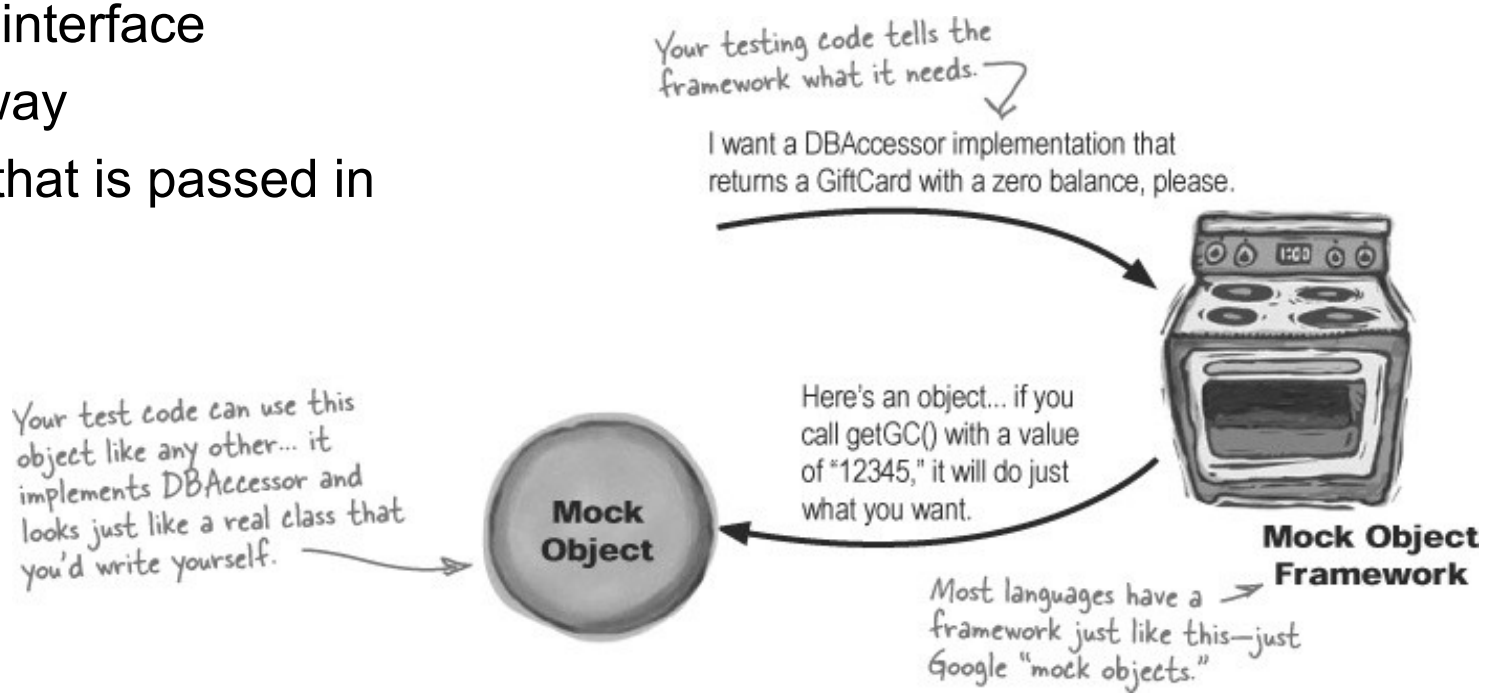
To Cover All Cases

- We need lots of different but similar cases
- For example,

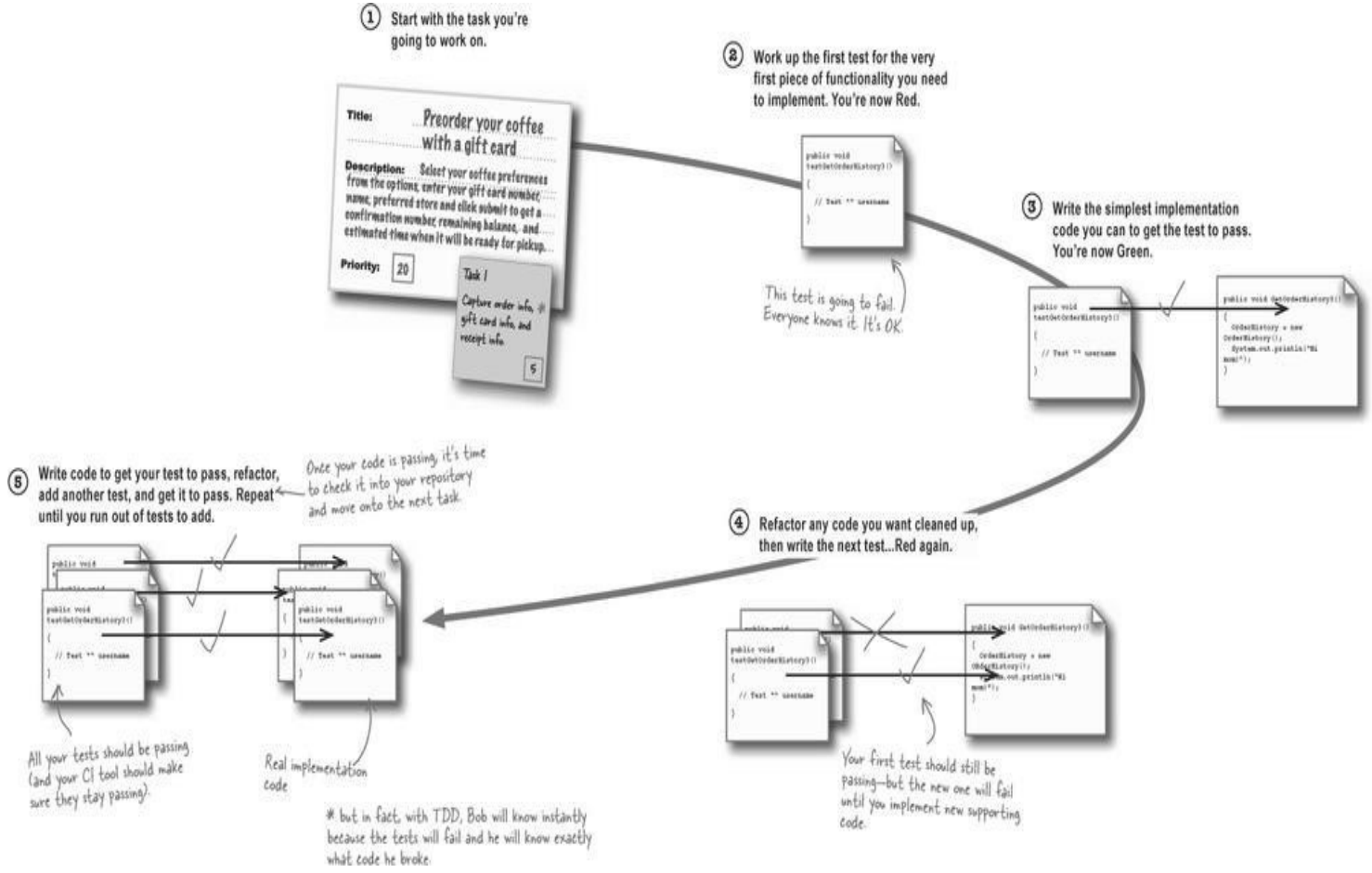


INSTEAD, LET'S GENERATE OBJECTS

- Create a facility (Class) that
  - Creates a new object
    1. that conforms to an interface
    2. Behaves a certain way
    3. Based on the input that is passed in



SUMMARY


1 Start with the task you're going to work on.
Title: Preorder your coffee with a gift card
Description: Select your coffee preferences from the options, enter your gift card number, name, preferred store and click submit to get a confirmation number, remaining balance, and estimated time when it will be ready for pickup.
Priority: 20
Task 1
Capture order info, gift card info, and receipt info.
5
2 Work up the first test for the very first piece of functionality you need to implement. You're now Red.
This test is going to fail. Everyone knows it. It's OK.
3 Write the simplest implementation code you can to get the test to pass. You're now Green.
4 Refactor any code you want cleaned up, then write the next test...Red again.
Your first test should still be passing—but the new one will fail until you implement new supporting code.
5 Write code to get your test to pass, refactor, add another test, and get it to pass. Repeat until you run out of tests to add.
Once your code is passing, it's time to check it into your repository and move onto the next task.
All your tests should be passing (and your CI tool should make sure they stay passing).
Real implementation code
* but in fact, with TDD, Bob will know instantly because the tests will fail and he will know exactly what code he broke.

Retrospective Questions

- What (amount/depth) of code do we use to satisfy each test?
- What is the TDD cycle? (Each step and meaning)
- What is the strategy pattern? How does it help TDD?
- What are the benefits of TDD? Are you convinced?