

Agile Java Development – Test-driven Development using JUnit & Eclipse

Jason Gorman

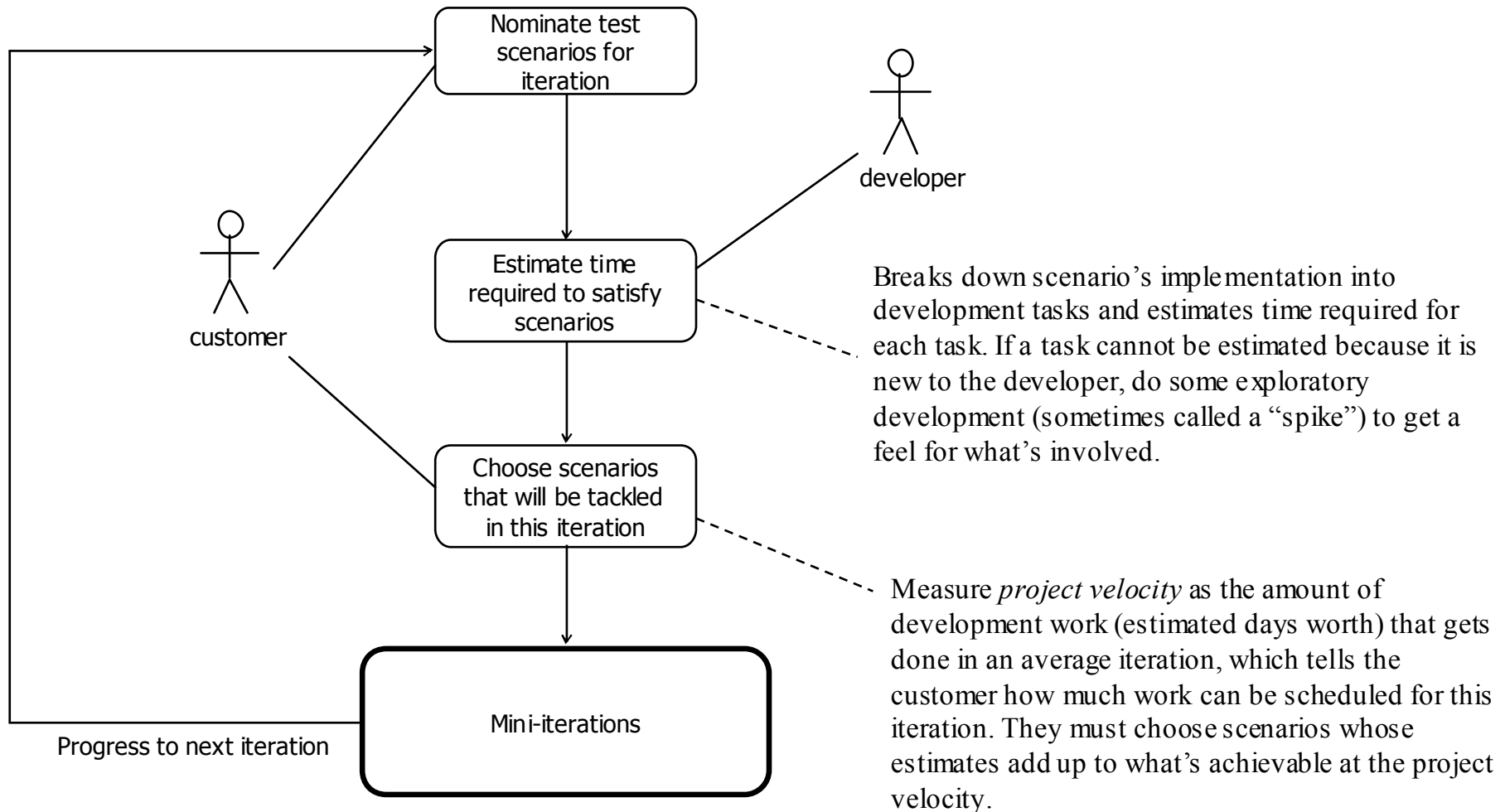
Test-driven Development

- Drive the design and construction of your code on unit test at a time
- Write a test that the system currently fails
- Quickly write the code to pass that test
- Remove any duplication from the code
- Move on to the next test

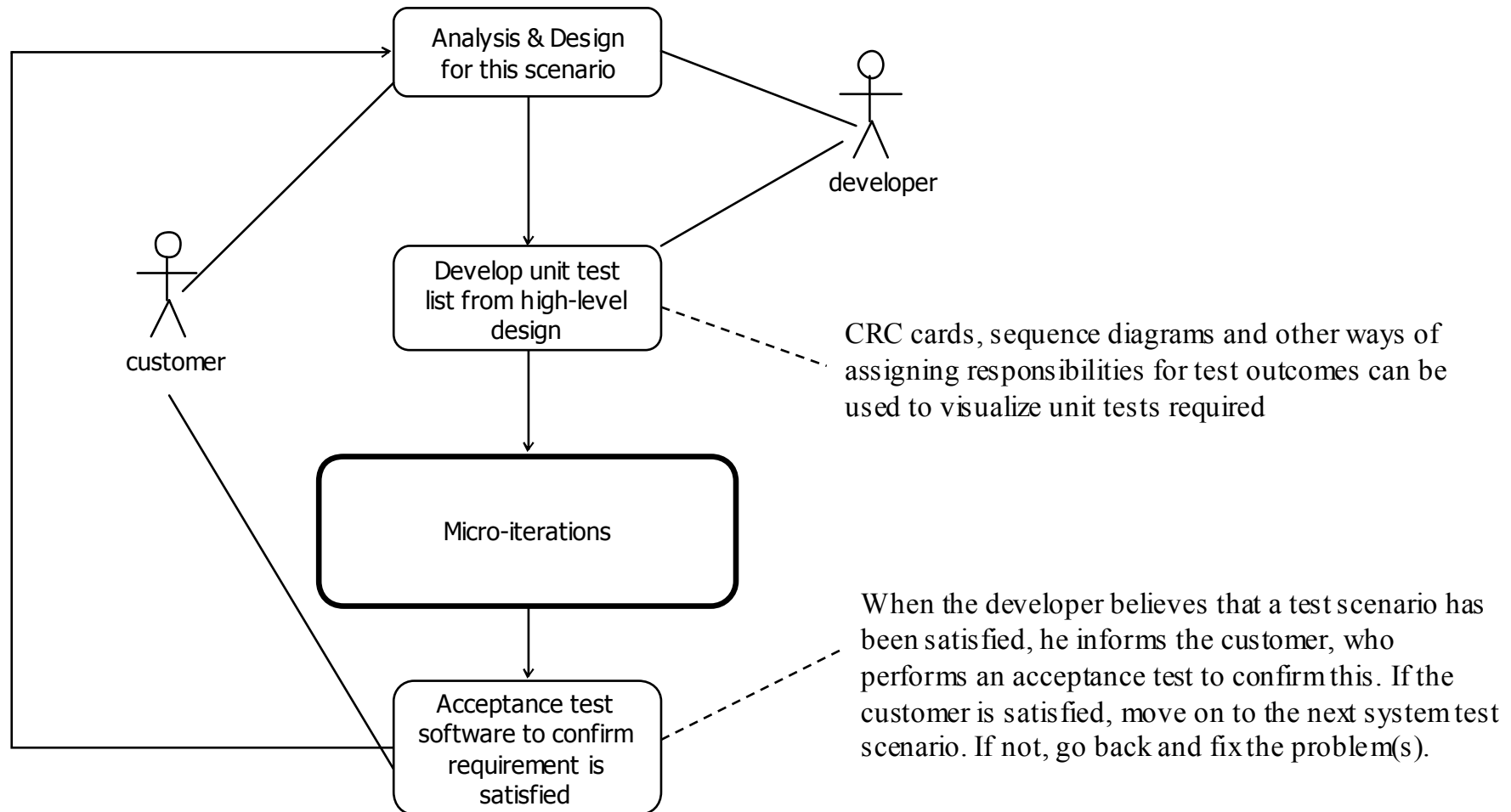
TDD Benefits

- Focuses your efforts on specific concrete test cases and helps pace development (micro-iterations)
- Provides continuous feedback about whether your code is working
 - Boosts confidence when working on complex systems
- Evolves code to do *exactly what you need* and no more
- Provides a suite of tests you can run often to be sure your code is always working
 - Code is easier to change without inadvertently breaking dependant code
 - Means better assurance when integrating your code into CVS, VSS etc

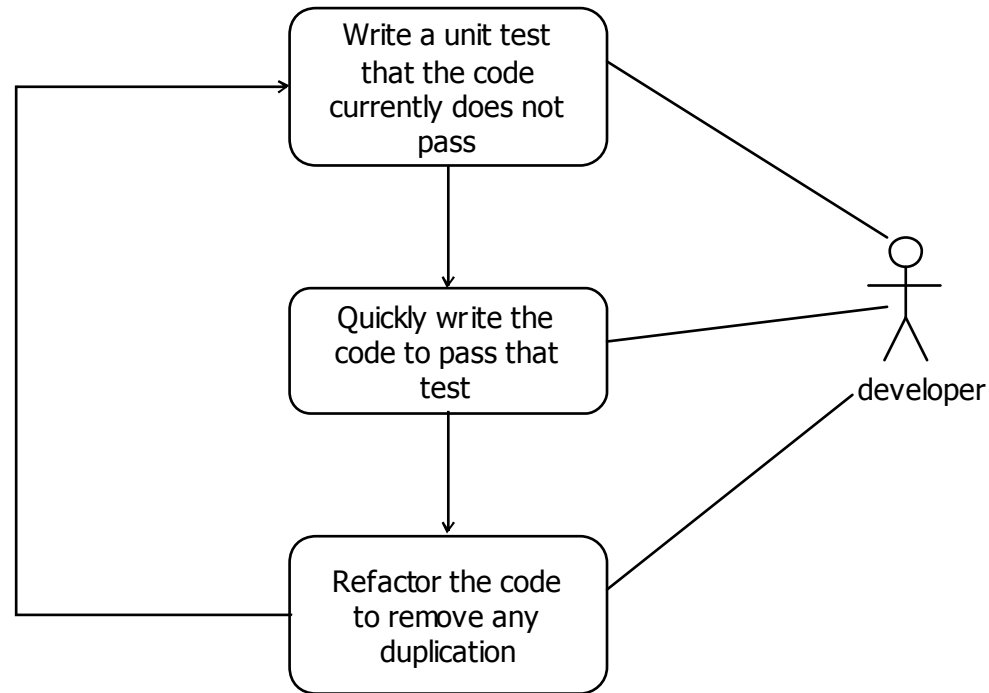
Iterations



Mini-iterations



Micro-iterations



Introducing xUnit

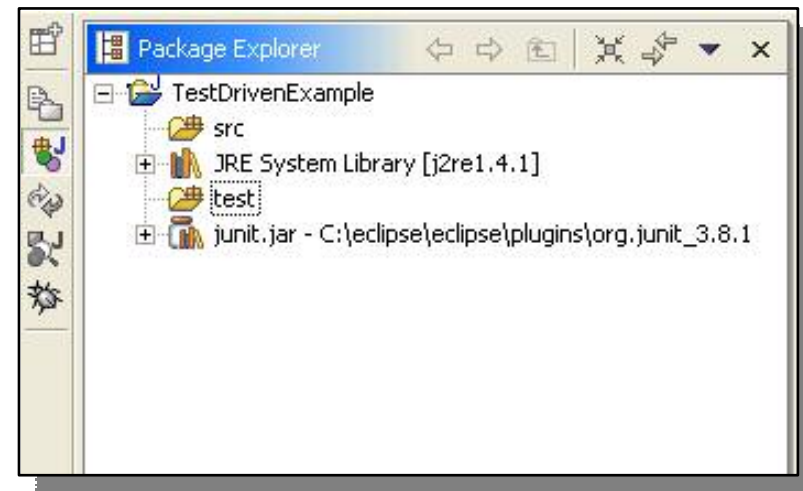
- Original framework developed for Smalltalk by Kent Beck
- Versions for most programming languages (CppUnit for C++, JUnit for Java, NUnit for .NET etc)
- Simple concepts
 - Test fixture : a class that contains one or more test methods
 - Test method : a method that executes a specific test
 - Test runner : an application that finds and executes test methods on test fixtures
 - Assertion : a Boolean expression that describes what must be true when some action has been executed
 - Expected Exception : the type of an Exception we expect to be thrown during execution of a test method
 - Setup : Code that is run before every test method is executed (eg, logging in as a particular user or initializing a singleton)
 - Teardown : Code that is run after every test method has finished (eg, deleting rows from a table that were inserted during the test)

Getting Started with JUnit

1. If you're using [Eclipse](#), JUnit and the IDE plug-in will normally be included as part of the distribution
2. (If you're not using Eclipse you can still use JUnit as a standalone testing tool by downloading it from <http://www.junit.org>)

Creating a Test Project in Eclipse

1. Create a Java project
2. Add junit.jar to your project's build path
3. Create two source directories *src* and *test*

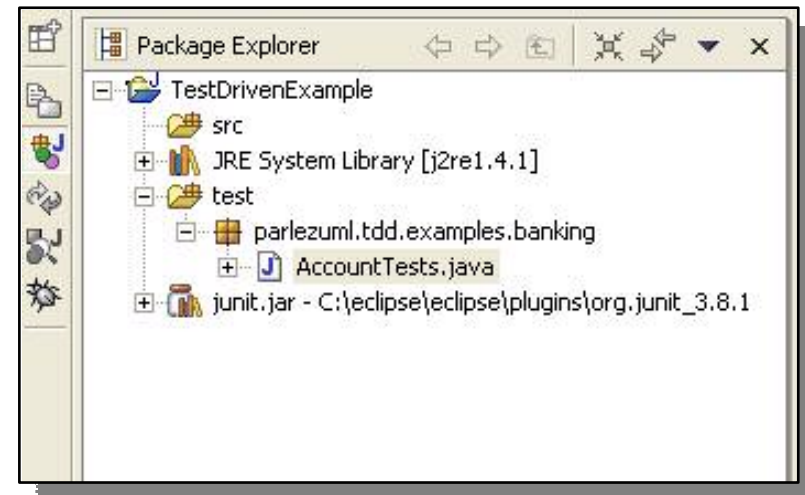


Creating a Test Class

1. Add a new class to the test project with a name that makes it obvious which class it will be testing (eg, <class under test name>Tests)
2. Import `junit.framework.TestCase` and have your test class extend it

```
package parlezuml.tdd.examples.banking;  
  
import junit.framework.TestCase;  
  
public class AccountTests extends TestCase {  
  
}  

```



Creating a Test Method

1. Declare a public method with a name that starts with **test**<description of test>
 - All test methods must be **public** and start with *test* because JUnit uses reflection to find and execute test methods and that's how it knows which methods are test methods
 - Test methods must not take parameters or return values

```
package parlezuml.tdd.examples.banking;

import junit.framework.TestCase;

public class AccountTests extends TestCase {

    public void testWithdrawWithSufficientFunds() {

    }

}
```

Write the Test Assertions First

1. Don't write the code needed to execute the test scenario until you've written the assertions the code needs to satisfy first
2. If there's more than one assertion, consider working one assertion at a time

```
public void testWithdrawWithSufficientFunds() {  
    assertTrue(account.getBalance() == oldBalance - amount);  
}
```

expected value

actual value

Write the Test Body

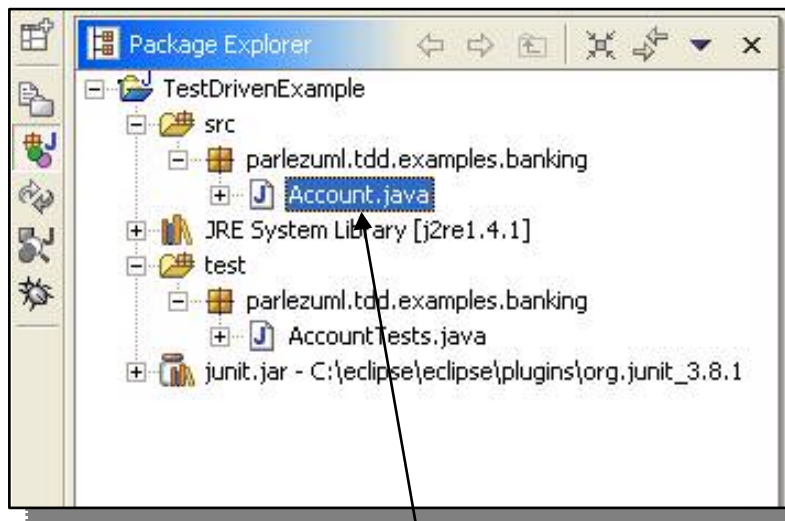
1. Write the test code needed to execute the scenario

initial balance

```
public void testWithdrawWithSufficientFunds() {  
    Account account = new Account(500);  
    float amount = 250;  
    float oldBalance = account.getBalance();  
  
    account.withdraw(amount);  
  
    assertTrue(account.getBalance() == oldBalance - amount);  
}
```

Write the Code To Pass The Test

...and *only* the code needed to pass the test



src and *test* directory structures match each other, making test code easier to find

```
package parlezuml.tdd.examples.banking;

public class Account {

    private float balance = 0;

    public Account(float initialBalance){
        balance = initialBalance;
    }

    public void withdraw(float amount){
        balance = balance - amount;
    }

    public float getBalance() {
        return balance;
    }

}
```

Running the test with the Eclipse JUnit plug-in

The screenshot shows the Eclipse IDE interface. The Package Explorer on the left shows the project structure with 'AccountTests.java' selected in the 'test' folder. The 'Run' menu is open, and the 'Run As' option is selected, leading to a submenu where 'JUnit Test' is chosen. The JUnit view at the bottom shows a green bar at the top, indicating a successful test run. The status bar shows 'Runs: 1/1', 'Errors: 0', and 'Failures: 0'. The test method 'testWithdrawWithSufficientFunds' is highlighted in the JUnit view.

1. Select the test class in the Package Explorer
2. Select **Run... Run As... JUnit Test** from the main menu
3. The test results will be displayed in the JUnit view

Green bar means the test passes

Move On To The Next Test

```
public void testDepositAmountGreaterThanZero() {  
  
    Account account = new Account(500);  
    float amount = 250;  
    float oldBalance = account.getBalance();  
  
    account.deposit(amount);  
  
    assertTrue(account.getBalance() == oldBalance + amount);  
}
```

```
package parlezuml.tdd.examples.banking;  
  
public class Account {  
  
    private float balance = 0;  
  
    public Account(float initialBalance) {  
        balance = initialBalance;  
    }  
  
    public void withdraw(float amount) {  
        balance = balance - amount;  
    }  
  
    public float getBalance() {  
        return balance;  
    }  
  
    public void deposit(float amount) {  
        balance = balance + amount;  
    }  
}
```

Remove Any Duplication

...from the code under test *and* from the test code (remember that the test code needs to be as easy to change as the “model” code its testing)

code under test

```
public class AccountTests extends TestCase {  
  
    private Account account;  
    private float amount;  
    private float oldBalance;  
  
    public void setUp() {  
  
        account = new Account(500);  
        amount = 250;  
        oldBalance = account.getBalance();  
  
    }  
  
    public void testWithdrawWithSufficientFunds() {  
  
        account.withdraw(amount);  
  
        assertTrue(account.getBalance() == oldBalance - amount);  
  
    }  
  
    public void testDepositAmountGreaterThanZero() {  
  
        account.deposit(amount);  
  
        assertTrue(account.getBalance() == oldBalance + amount);  
  
    }  
  
}
```

JUnit runs the **setUp()** method before every test method, and the **tearDown()** method after every test method

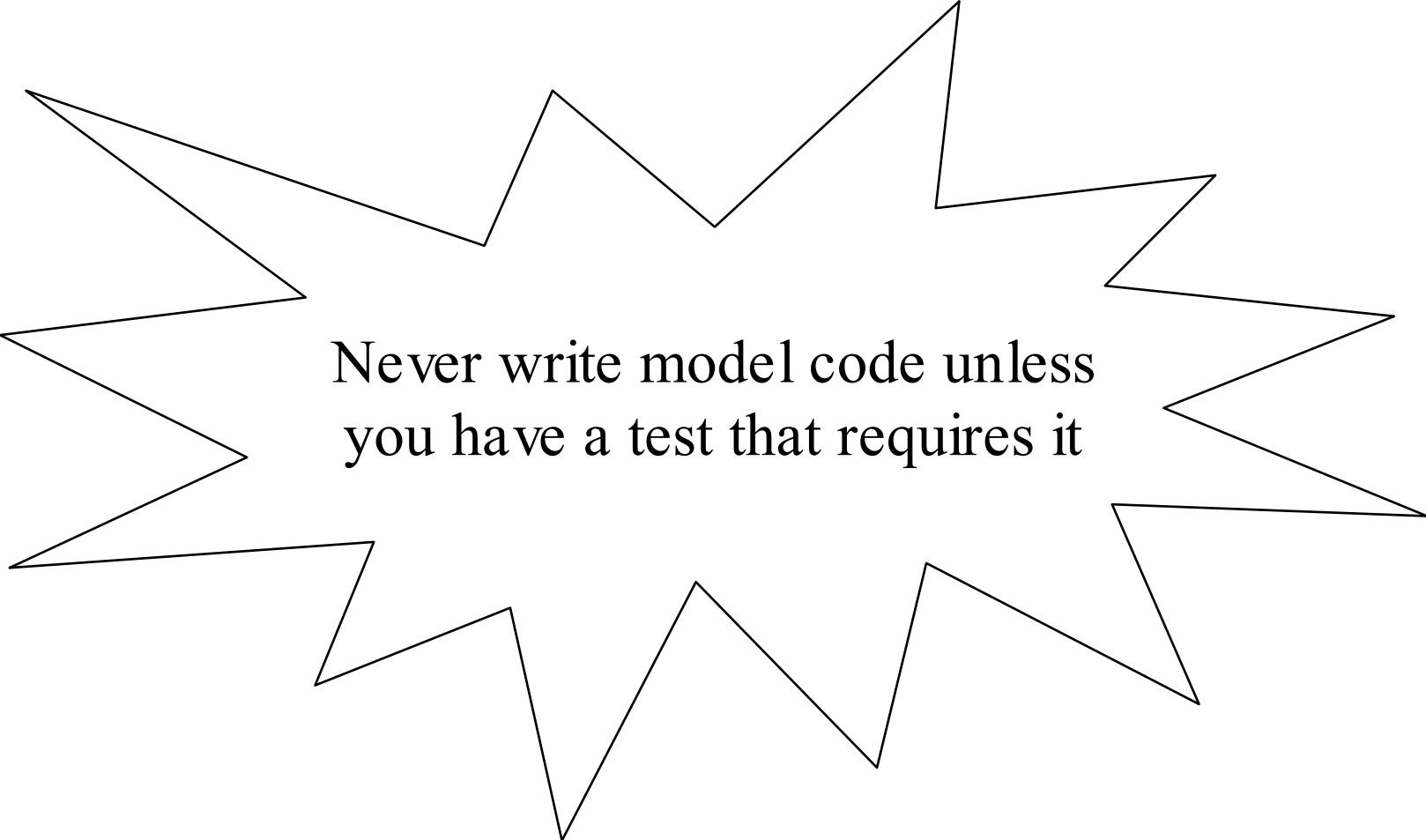
And So On...

```
public void testWithdrawWithSufficientFunds() {  
  
    try {  
        account.withdraw(amount);  
    } catch (AccountException e) {  
        fail();  
    }  
  
    assertTrue(account.getBalance() == oldBalance - amount);  
}  
  
public void testWithdrawWithInsufficientFunds() {  
  
    try {  
        account.withdraw(1000);  
        fail(); // shouldn't get this far, so fail test  
    } catch (AccountException e) {  
        // do nothing as getting here means correct exception was thrown  
    }  
}
```

```
public void withdraw(float amount) throws AccountException {  
    if(amount > balance){  
        throw new AccountException();  
    }  
    balance = balance - amount;  
}
```

The idea of TDD is to flesh out the design of your code one unit test at a time

TDD – The Golden Rule



Never write model code unless
you have a test that requires it

TDD Best Practices

- Only work on one test at a time
 - Don't write a whole bunch of tests and then try to pass them all in one go
- Write test assertions self-explanatory so other developers can readily see what the code is supposed to be doing just by reading the tests
 - `balance == oldBalance - amount` rather than `balance == 50`
- Make the structure of your test projects follow the structure of the projects they are testing so developers can easily navigate from the tests to the code they are testing and vice-versa
- Keep the tests small
 - More than 15 minutes on the same test is probably too long. You will lose focus.
 - If a test looks complicated, think of ways to break it down. If a test involves several methods or classes that you'll have to write to pass the test then either:
 - Write tests for those first and work your way up to the bigger test (bottom-up TDD)
 - *Fake them* to pass this test and then work your way down to implement each of them one test at a time (top-down TDD)
- If you're working in a team, integrate your code changes every few unit tests (say, every hour or two)* to make sure there aren't any nasty surprises waiting for you in someone else's changes
- If you're working alone, end every day with one broken test to help you quickly refocus the next morning
- If you're working in a team, never ever leave with broken tests!
- Automate acceptance tests, too (this is a challenge, I must warn you) so that you can be confident when you're done and ready to release it to the customer, and also so that you know everything that's been accepted will stay like that...
- Take regular breaks and work reasonable hours. TDD is supposed to set a pace that can be sustained indefinitely.

TDD & Continuous Integration

- Before integrating your code changes tell all team members that you are about to. Until you have a successful build on the integration server, nobody else should consider integrating their changes
- Get any changed files that have been checked in to your team's repository (eg, VSS, CVS) since you last checked out the code and merge those with your changes
- Run *all* of the tests
- Do not integrate if any of the tests fail
- If all tests pass then check in your code changes
- Do a build with the latest code in the repository on the integration server and run all of the tests against that build
- If all the tests pass then you have a successful build and other developers can start to integrate
- If any tests fail then you have a broken build, which means nobody else can start integrating – your team's priority is now to find out why the test(s) fail and fix the problem ASAP