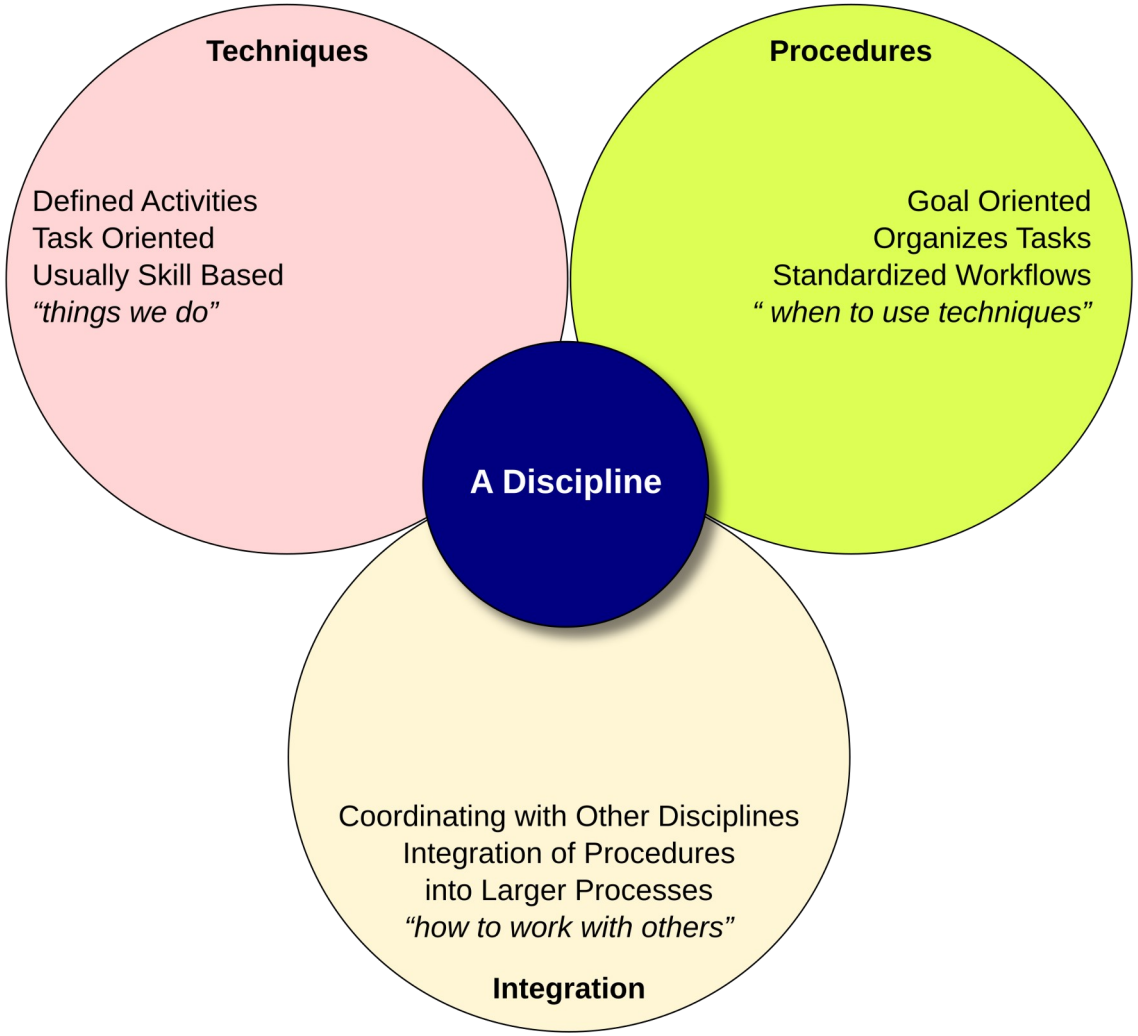

Full Stack Development
Containers, Microservices and UI
5. Test Driven Development

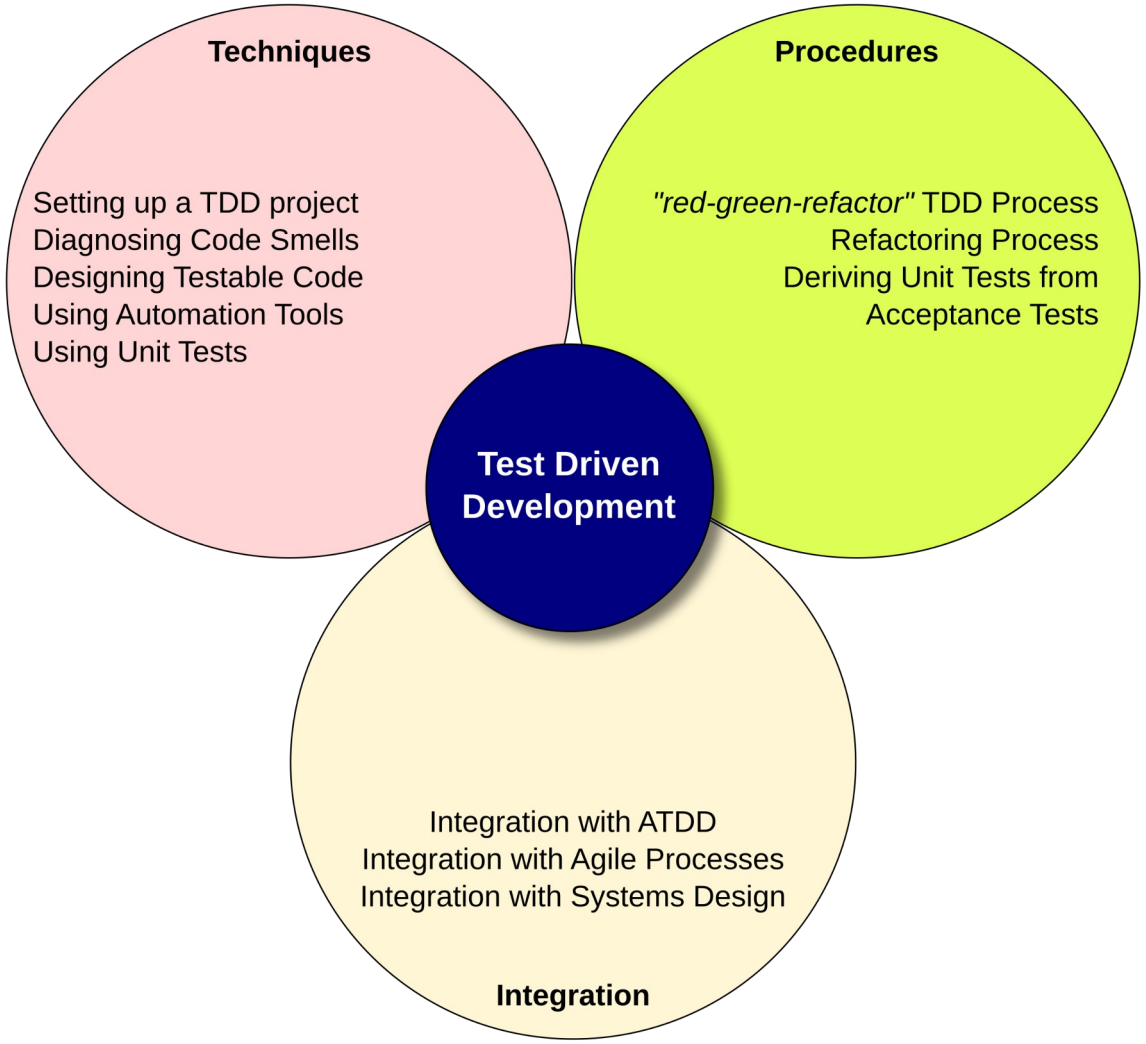
# Introducing TDD

- Features of Test Driven Development
  - TDD is a programming discipline
  - TDD has become a standard best practice for programming
  - TDD enables programmers to write better code faster
    - but only if it is followed correctly!
- Requires programmers follow best object-oriented design and coding practices
  - Characterized by the integrated user of automated tools
  - Integrates well with Agile and other practices

# Definition of "Discipline"



# TDD as a Discipline



I don’t like user interface-based tests.

In my experience, tests based on user interface scripts are too brittle to be useful.  When I was on a project where we used user interface testing, it was common to arrive in the morning to a test report with twenty or thirty failed tests.

A quick examination would show that most or all of the failures were actually the program running as expected. Some cosmetic change in the interface had caused the actual output to no longer match the expected output.

Our testers spent more time keeping the tests up to date and tracking down false failures and false successes than they did writing new tests.

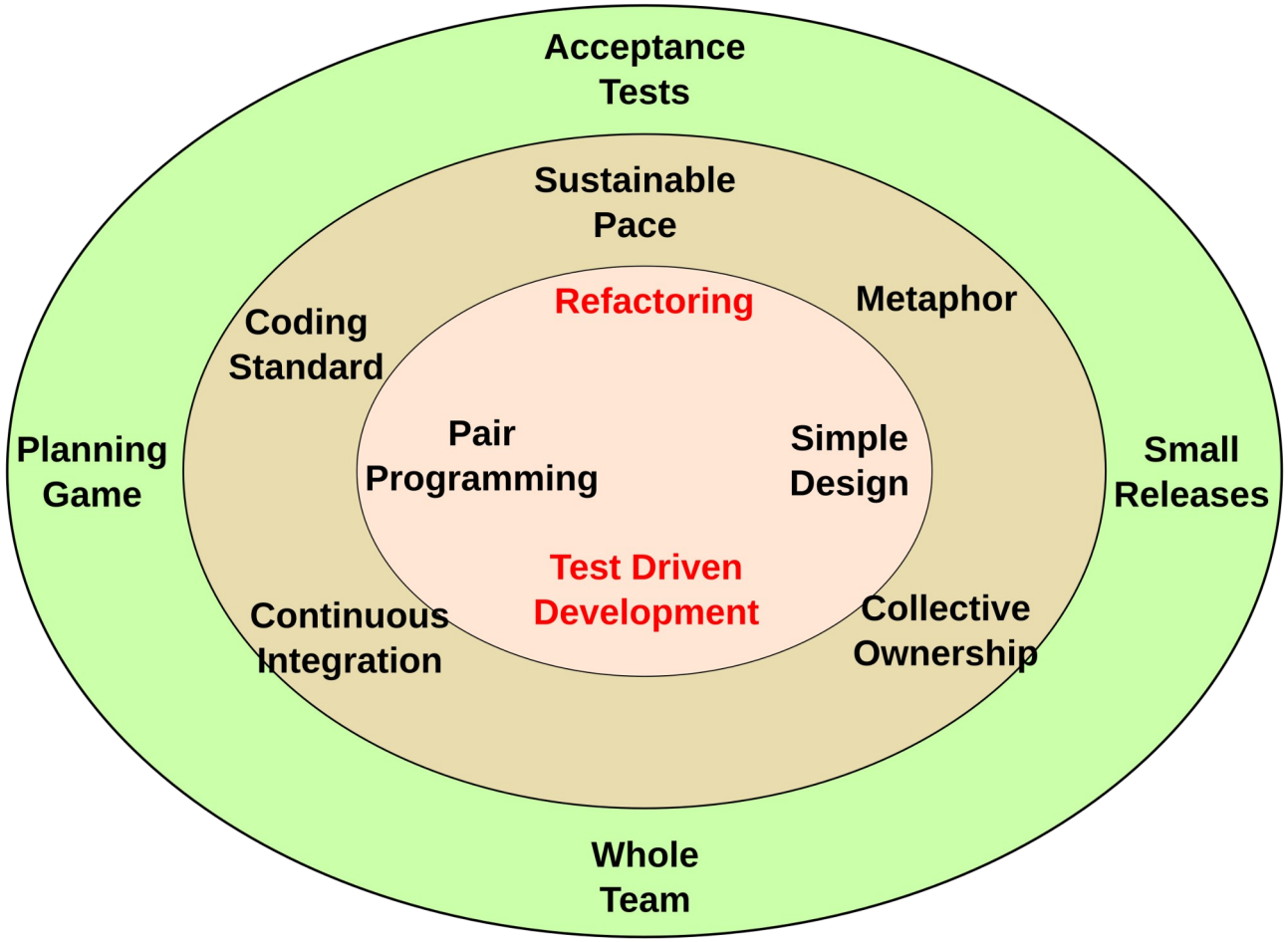
*Kent Beck describing his motivation for creating TDD*

*Kent Beck*

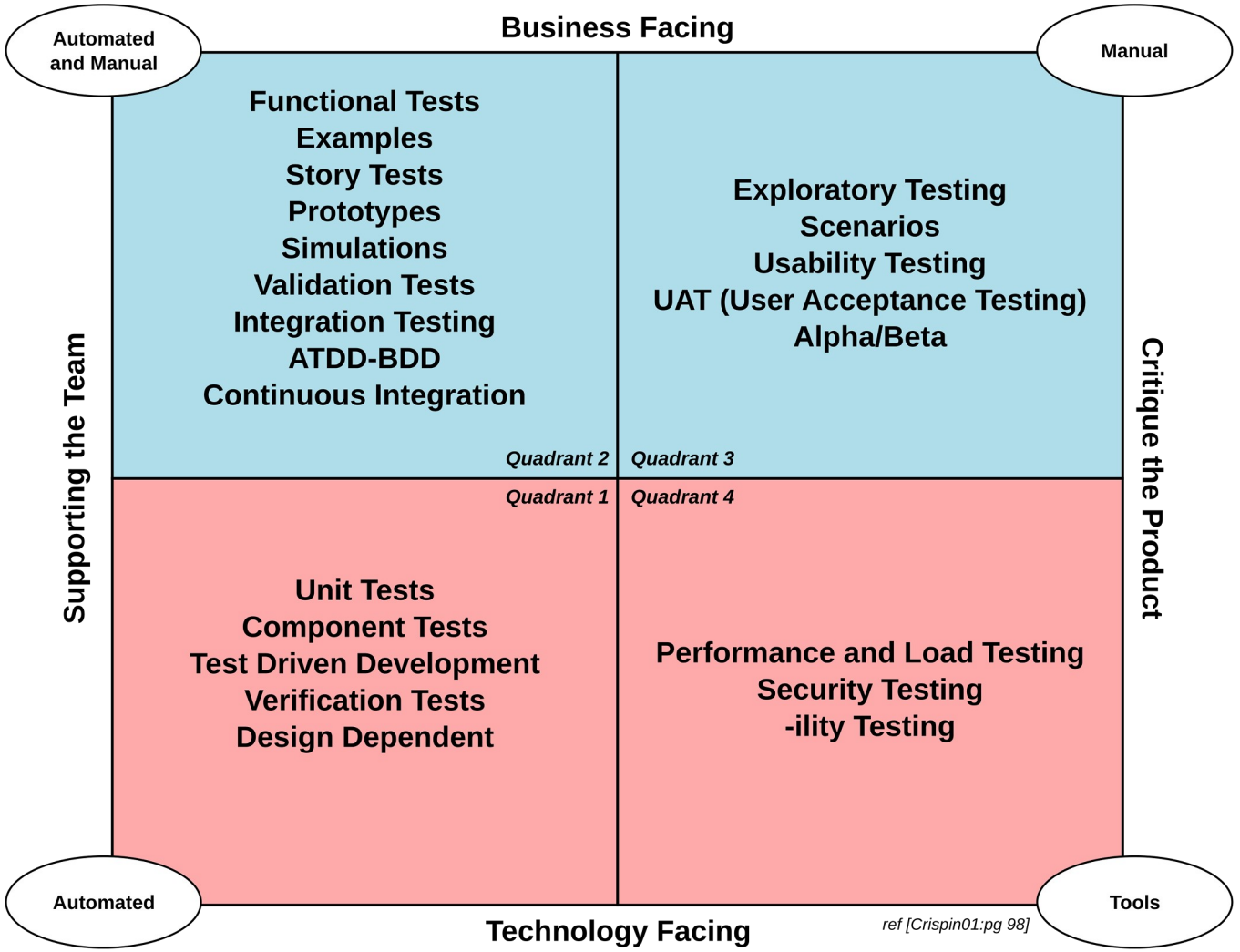
# Kent Beck and XP

- Kent Beck developed TDD as a way to write better code faster.
  - “I'm not a great programmer; I'm just a good programmer with great habits”
  - One of the founders of the Agile XP methodology
- Two of the core XP practices were TDD and Refactoring
- TDD and Refactoring were adopted by the larger Agile community
- TDD has now become recognized as a coding best practice

# The XP Onion



# TDD and Agile Testing




*ref [Crispin01:pg 98]*

# The State of TDD

- There is no TDD standard or canonical TDD process
  - Like a language, TDD has different "dialects“
  - All the dialects tend to share a common core of ideas
- TDD is often adapted to specific environments or methodologies
  - Many developers use "lazy" TDD
  - Practices are used in a casual fashion or inconsistently
  - Lazy TDD runs into problems when the projects scale up

# Empirical Evidence

- On average code produced with TDD:
  - Has a lower defect density
  - Has better coverage
- Shows mixed results as to increases in programmer productivity and decreased development time
  - Depends on whether it is applied in a strict or lazy fashion
- Often shows initial increase in development time but offset by “down the road” savings in development and application support
  - Is often more loosely coupled
  - Has higher client satisfaction measures on average

# Programmer Comments on TDD

- I’m writing code faster with less rework and less profanity”
- “It’s like the code is writing itself, I’m not doodling in code any more trying to figure out why I wrote that piece of code a month ago”
- “My code is simpler, cleaner and easier to understand”
- “Amount of rework is reduced since bugs are caught early, which means I go home on time”
- “Continuous regression testing means that adding new code doesn’t break my existing code”
- “Programming is fun again. I don’t spend all my time on bug hunts and I get to think instead about optimizing my design and implementation”

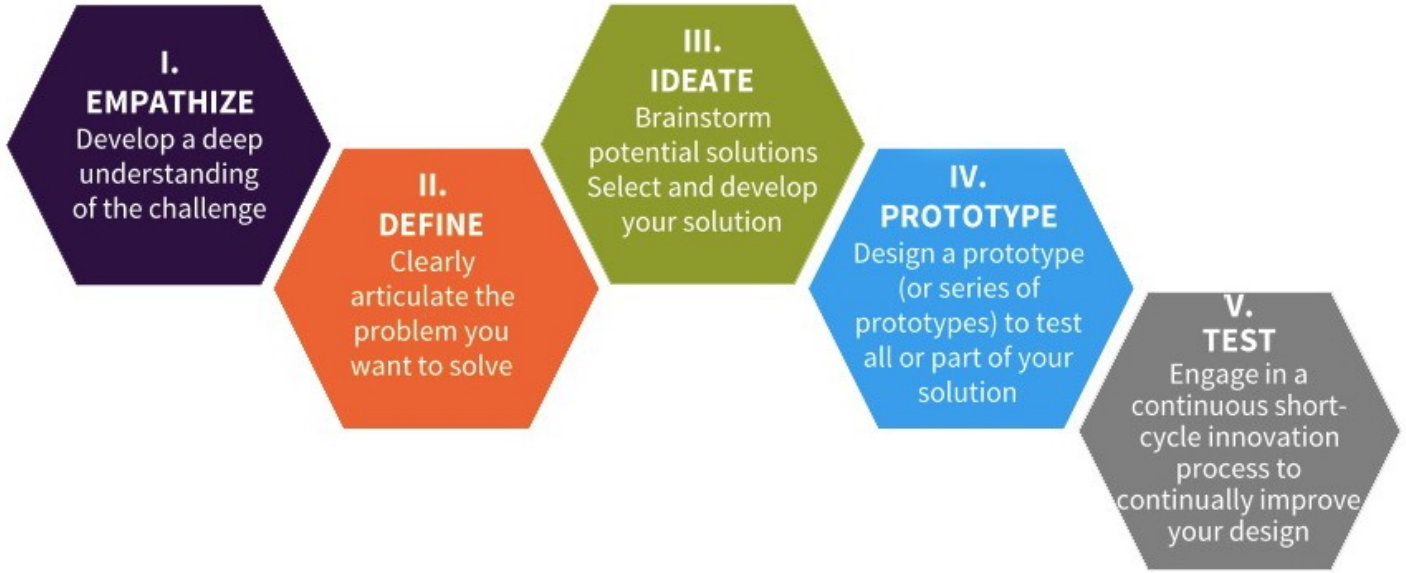
I've worked on three different development teams using TDD and on several more teams that didn't. I can tell you from first-hand experience that TDD produces code that has orders of magnitude fewer unit-level bugs, far fewer functional bugs, and an exponentially higher probability of meeting stakeholder expectations when compared to code produced by conventional programming techniques

*Lisa Crispin*

# Why Does TDD Improve Programming?

- TDD emphasized working through the tests first, then writing the code:
  - Forces programmers to immerse in the problem before writing any code
  - Forces analysis of the programmer understanding of what the software is supposed to do
- “If you can’t figure out what the right result of a test should be, then you don’t know what your code should do in that situation”
  - Forces a global view of the problem to be solved before crafting a solution

# The Stanford Design Process




***The Stanford Design Process***

The structure of the TDD process is remarkable like the Stanford Design Process which has been widely adopted as a methodology developing consistent creative and innovative design thinking.

From Wikipedia "Design thinking is a method for practical, creative resolution of problems and creation of solutions. It is a form of solution-based, or solution-focused thinking with the intent of producing a constructive future result. By considering both present and future conditions and parameters of the problem, several alternative solutions may be explored."

More than the act of testing, the act of designing tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding and the rest.

**If you can't test it, don't build it.**
**If you don't test it, rip it out.**

*Boris Beizer*

# JUnit Assumptions

- Assumptions about the code to be developed:
  - Component is designed according to OOP/OOD best practices
  - Component functionality is clearly defined during design
- Assumptions about state of the development:
  - Architecture of the system is defined: dependencies between components are known
  - Acceptance tests exist at the system level so we know how the component under development should behave
  - The levels of acceptable risk and quality are defined

# The Outside-In Principle

- TDD assumes code development follows the outside-in principle:
  - Interfaces are designed earlier during the design phase.
  - Interfaces describe all the functionality of a component
  - Classes are written to implement interfaces
- Because interfaces are defined during design:
  - They remain stable and do not change during code development
  - Interfaces can change if requirements change
  - Interfaces can change if the application architecture changes
- All interactions take place through a component's interface:
  - Therefore, a component can be fully functionally tested through its interface methods

# Design by Contract

- Interfaces establish contracts between the component and clients that call the methods
- Each interface method has three constraints as part of its contract
  - **Preconditions**: conditions that must be true before a method can be allowed to execute
  - **Postconditions**: conditions that must be true after the method executes
  - **Invariants**: conditions that must not change as a result of executing the method
- Tests are easier to implement for code when these constraints are known

# Command Query Segregation

- Methods should be either a command or a query
  - Borrowed from the database world
- Queries do not alter the state or internal content of the object
  - The don’t “write” anything, they only read data
- Commands cause a change in the object or environment
  - The only return value should be information about the success of failure of the command
- Functions that combine both operations are;
  - Prone to synchronization errors when being executed by multiple clients
  - Hard to debug
  - Hard to test
- “Asking a question should not change the answer.”

# TDD Testing Assumptions

- TDD assumptions about how testing should be done:
  - Test execution should always be automated
  - There should be no test code inside the production code
  - Testing is not debugging: those tools already exist elsewhere
  - There should only be one copy of the application code, there should not be a “test” version of the production code
  - The presence of test code should not impact the design of the production code

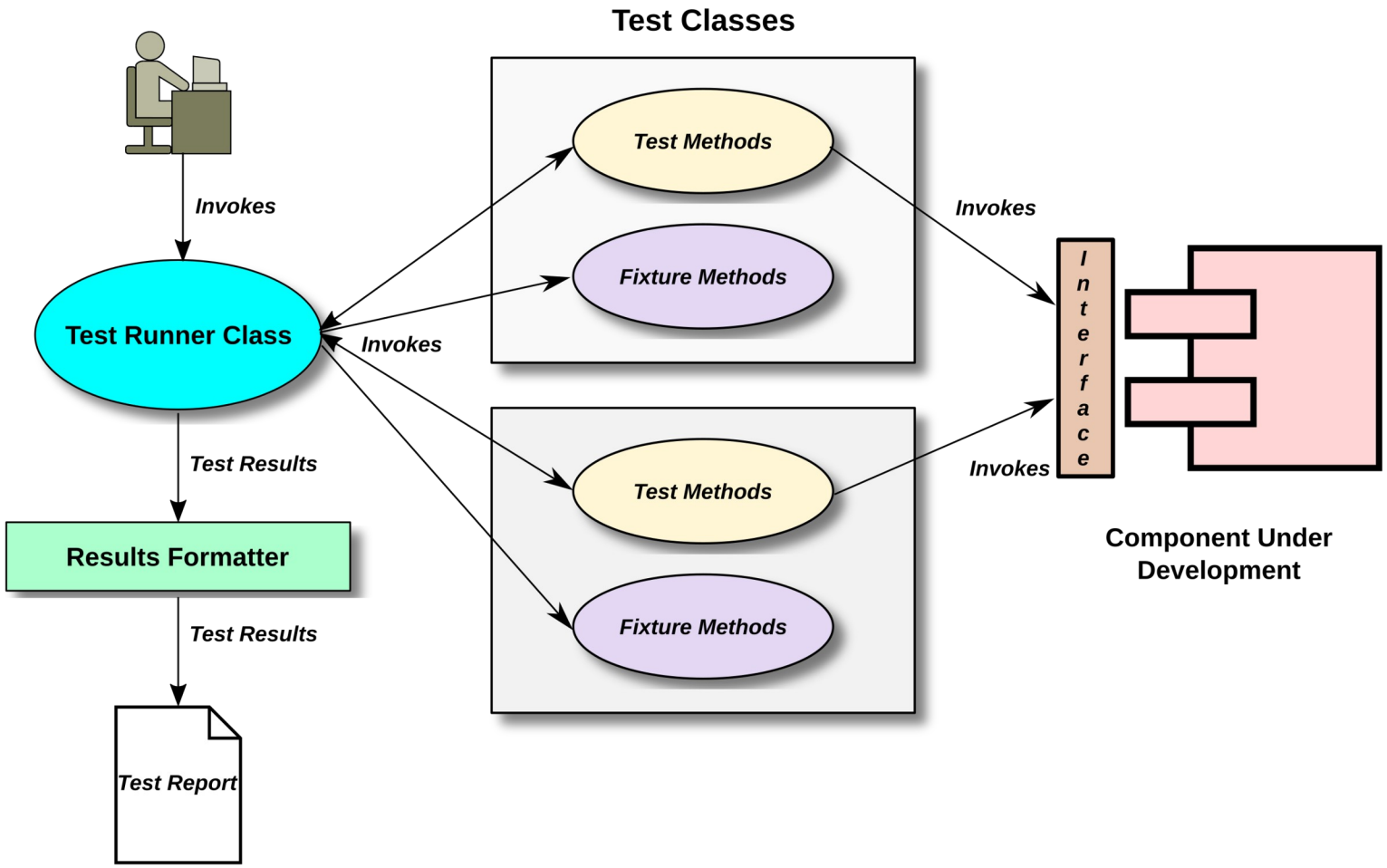
# Some JUnit Background

- Kent Beck originally developed SUnit – a testing framework for Smalltalk programmers in the mid 1990s
- Beck and Eric Gamma converted it to a Java framework and called it JUnit on a flight from Zurich to Atlanta in 1997
- The original architecture has come to be known as xUnit and is the basis for many language ports (CppUnit for C++) for example
- The xUnit family of tools shares a characteristic architectural pattern

# JUnit Architecture

- JUnit is made up of:
  - **Test Runner**: A class or mechanism that is responsible for executing the tests
  - **Test Class:** One or more test classes containing the tests for a component under development
  - **Test Method:** Each test is implemented by a test method in a test class
  - **Test Fixture:** The state the system to be in for a test to be run
  - **Test Suite**: A set of tests that all share the same test fixture
  - **Test Execution**: The running of the test case along with any fixture methods required to set up and tear down the test fixtures
  - **Test Result Formatter:** Responsible for reporting on the results of the tests in a usable format
  - **Assertion Set:** Functions that verify the results of a test

# JUnit Architecture




Setting Up JUnit
Demo


JUnit Setup
Lab Testing 1

# The Calculator Project

- To demonstrate the functionality of JUnit, we will implement a trivial example of a calculator that does basic arithmetic.
  - This is not Test Driven Development, this is just experimenting with Junit
  - We will be using JUnit (JUnit 5 can have some hiccups in eclipse we want to avoid) – the functionality is the same though
  - We will be working within Eclipse
- Initial steps:
  - Create the Java project
  - Define the Calculator interface
  - Create the implementing class CalcImp
  - Create the JUnit test class

# The Calculator Project

Calculator.java

```
1
2  public interface Calculator {
3      public int add(int a, int b);
4      public int sub(int a, int b);
5      public int mult(int a, int b);
6      public  int div(int a, int b);
7
8  }
```

Calculator.java | CalcImp.java

```
1
2  public class CalcImp implements Calculator {
3
4      public int add(int a, int b) {
5          return 0;
6      }
7
8      public int sub(int a, int b) {
9          return 0;
10     }
11
12     public int mult(int a, int b) {
13         return 0;
14     }
15
16     public int div(int a, int b) {
17         return 0;
18     }
```

***The Project Code***

The first image shows the Calculator interface which describes the calculator functionality – which is quite trivial. The second image shows the implementing class where we have to create the code that implements the functionality defined by the interface.

This is an example of "outside-in" development where the interface is defined before we start to write any code. This is the state we want our code in just before we start to do our Test Driven Development.

Also notice that these methods are all queries since they do not change anything inside a calculator object nor do they create any side effects.

# Fixture Methods

- Fixture Methods are run to set up and tear down tests
  - Set up methods prepare the test environment and system state
  - Tear down methods undo what set up methods do
- Fixture methods are identified by annotations
  - @BeforeClass are executed once before any tests are run
  - @AfterClass are executed once after all tests have run
  - @Before are all executed before each test is run
  - @After are all executed after each tests is run
- Fixture methods with the same annotation are not guaranteed to run in the order they appear in the code

# Adding the Test Class

- Eclipse is used to add a JUnit test class to the project using a builtin wizard, although we could hand code it
- The test code is in a separate class from the production code
- The test class will be created that creates test method “stubs” that we will use to implement the tests and fixture methods

# Fixture Method Example

Calculator.java | CalcImp.java | CalcImpTest.java

```
11    @Test
12    public void divTest001() {
13        Calculator c= new CalcImp();
14        System.out.println("       Div001");
15        assertEquals("Oops!",2,c.div(6,3));
16    }
17    @Test
18    public void divTest002() {
19        Calculator c= new CalcImp();
20        assertEquals("Oops!",-2,c.div(-6,3));
21        System.out.println("       Div002");
22    }
23
24    @BeforeClass
25    public static void setUpBeforeClass() throws Exception {
26        System.out.println("*** BeforeClass ");
27    }
28
29    @AfterClass
30    public static void tearDownAfterClass() throws Exception {
31        System.out.println("*** AfterClass ");
32    }
33
34    @Before
35    public void setUp() throws Exception {
36        System.out.println("  --- Before ");
37    }
38
39    @After
40    public void tearDown() throws Exception {
41        System.out.println("  --- After ");
42    }
43 }
```

***Fixture Methods***

In the code, we have added some dummy fixture methods and a second test method.

# Fixture Method Example


***Fixture Method Output***

Running the test and looking at the console output, the sequence of execution of the fixtures and the test methods is clearly seen.


Fixture Methods
Demo

# Fixture Methods

## Lab Testing 2

# The Test Method

- The JUnit runner uses the annotations to find the test methods
  - All methods with the @Test annotation are test methods
  - The autogenerated names on the methods should be changed to something that more meaningful
  - Test methods must return void and take no arguments
  - The other methods with other annotations (@Before, @After, etc) are fixture method stubs
- At this point we can run the test method using the built in Eclipse JUnit runner

# Adding the Test Class

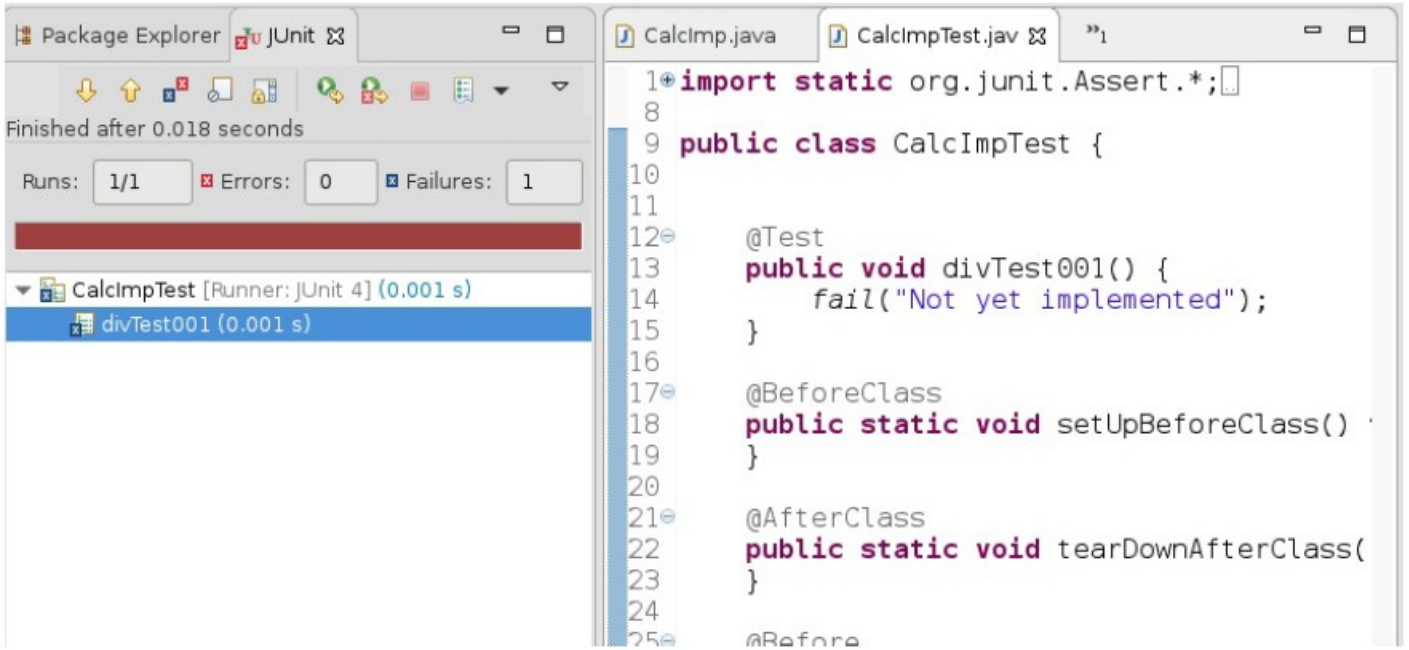
Calculator.java | CalcImp.java | CalcImpTest.java

```
1  import static org.junit.Assert.*;
8
9  public class CalcImpTest {
10
11
12     @Test
13     public void testDiv() {
14         fail("Not yet implemented");
15     }
16
17     @BeforeClass
18     public static void setUpBeforeClass() throws Exception {
19     }
20
21     @AfterClass
22     public static void tearDownAfterClass() throws Exception {
23     }
24
25     @Before
26     public void setUp() throws Exception {
27     }
28
29     @After
30     public void tearDown() throws Exception {
31     }
32 }
33
```

***The Generated Stubs***

This is what the final result of the using the wizard looks like, although the fixture methods have been moved to end of the file for readability.

# The Test Method




***Running the Tests***

By selecting the "Run as JUnit test" option, Eclipse calls the JUnit default runner class which then looks through the test class and runs all of the @Test methods.

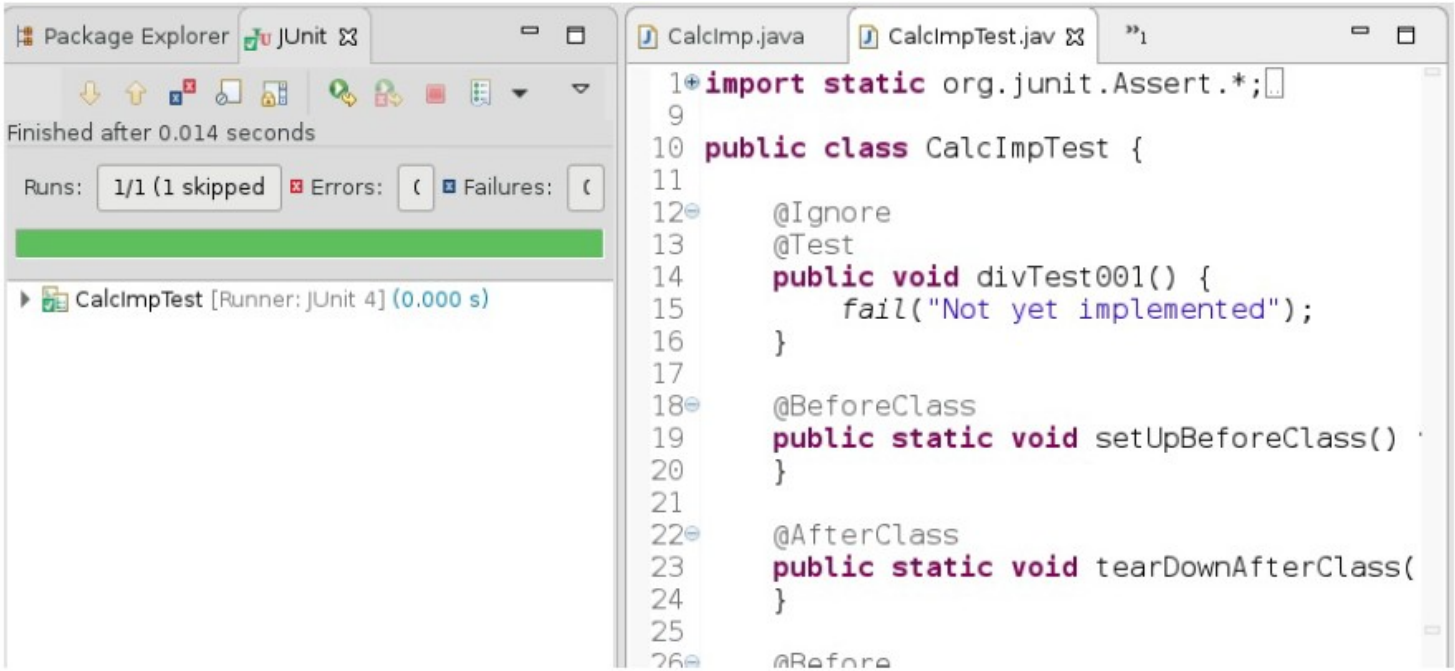
The Eclipse built in results formatter reports that 1 test was run and 1 test failed. The test failed because when the stub was generated, a fail assertion was placed in the body as a reminder to add the code to test method.

The test methods are not guaranteed to run in any particular order.

# The Test Results

- JUnit cannot understand what a test “means” so it relies on us to tell it whether a test has passed or failed
  - If an assertion exception is thrown, JUnit marks the test as failed
  - The fail() method throws an exception every time it executes
- Tests that don’t throw assertion exceptions are considered to have passed
  - Best practice: always have a fail() method in a test method until the test code is written

# Ignoring Tests

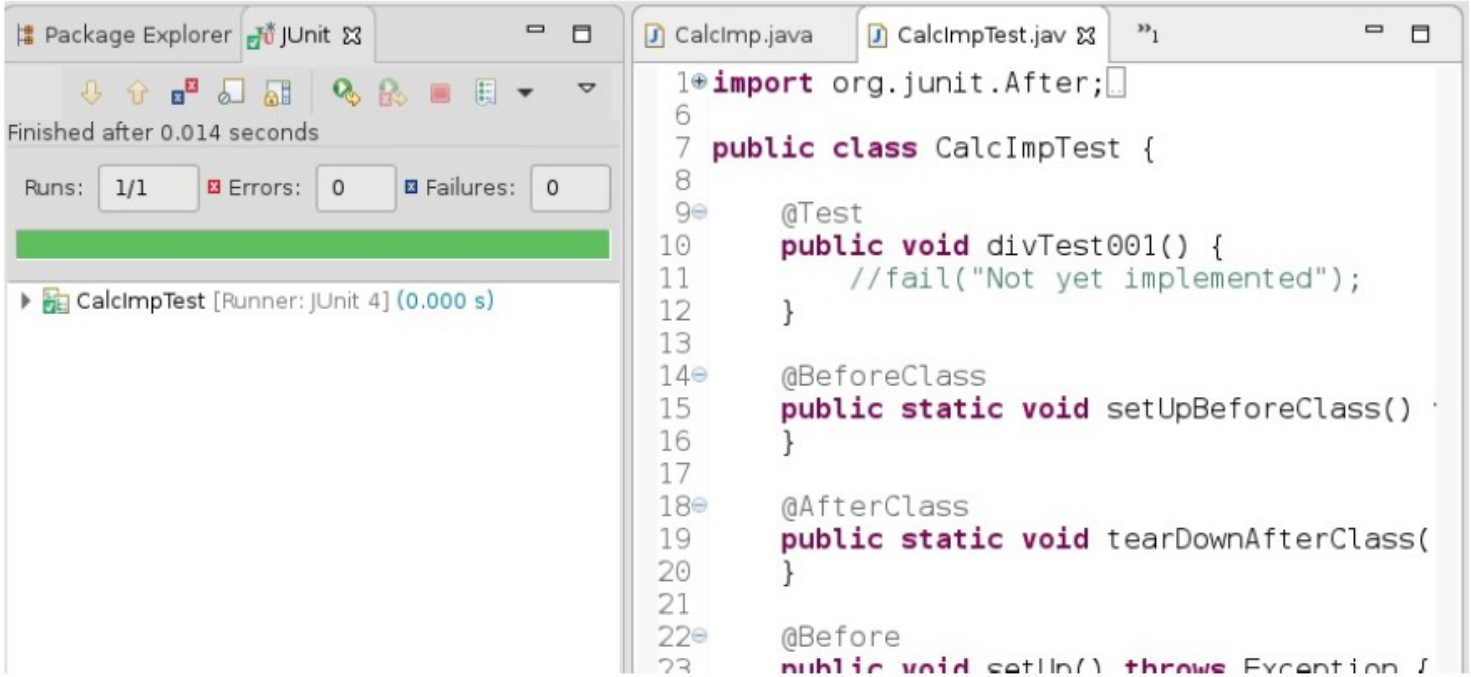



***Ignoring Tests***

The @Ignore annotation has been added to the test methods. This causes JUnit to skip the test and not count it as a pass or failure. The @Ignore annotation is used to suppress reporting about a test until we actually want to run it. Ignoring a test method is a lot safer than editing out the fail() annotation.

Because no test has failed, JUnit reports that all of the tests passed.

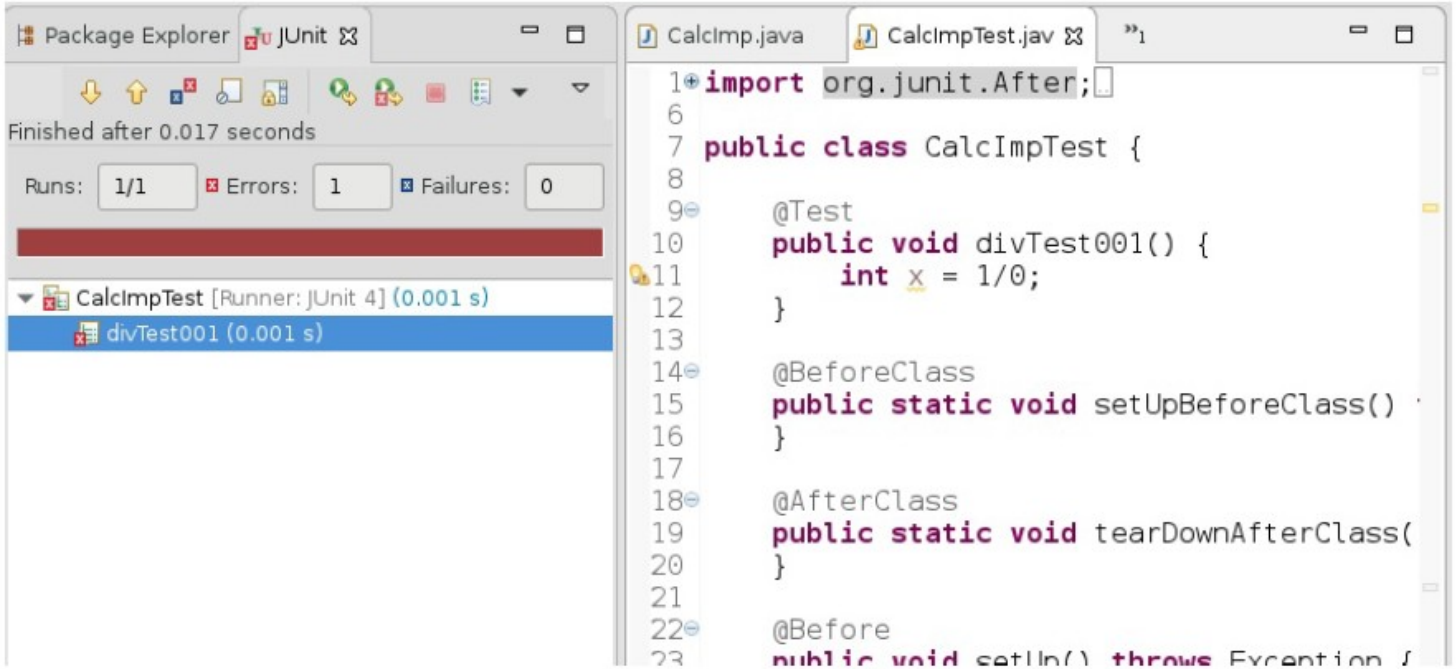
# Passing Tests




***Passing Tests***

In the example, the @Ignore annotation has been removed and the fail() assertion commented out. Now the test passes vacuously since it doesn't do anything that informs JUnit a failure has taken place. Notice the only different between this output and the previous slide is that in this case, no tests are reported as being skipped.

# Errors are not Failures




***Errors are NOT Failures***

In the example above, a Java error (divide by zero) has been added to the code. This caused an exception to be thrown that was not generated by an assertion statement. Because the divide by zero exception is not an assertion exception, this test has not failed or passed from a testing point of view because it could not be run.

Notice that this is reported in the results window as an error and not as a failure.

# Writing Test Methods

- Each test method is a single test case consisting of
  - A test input
  - A description of the system state required for test execution
  - The expected correct output
- The test method:
  - Acquires an instance of the component under development
  - Invokes the method being tested using the test input
  - Compares the actual value returned with the expected value
  - If the two values match then the test passes, otherwise it fails
- Putting the system into the required test state is done with the fixture methods

# Implementing Test Case

CalcImp.java | CalcImpTest.jav

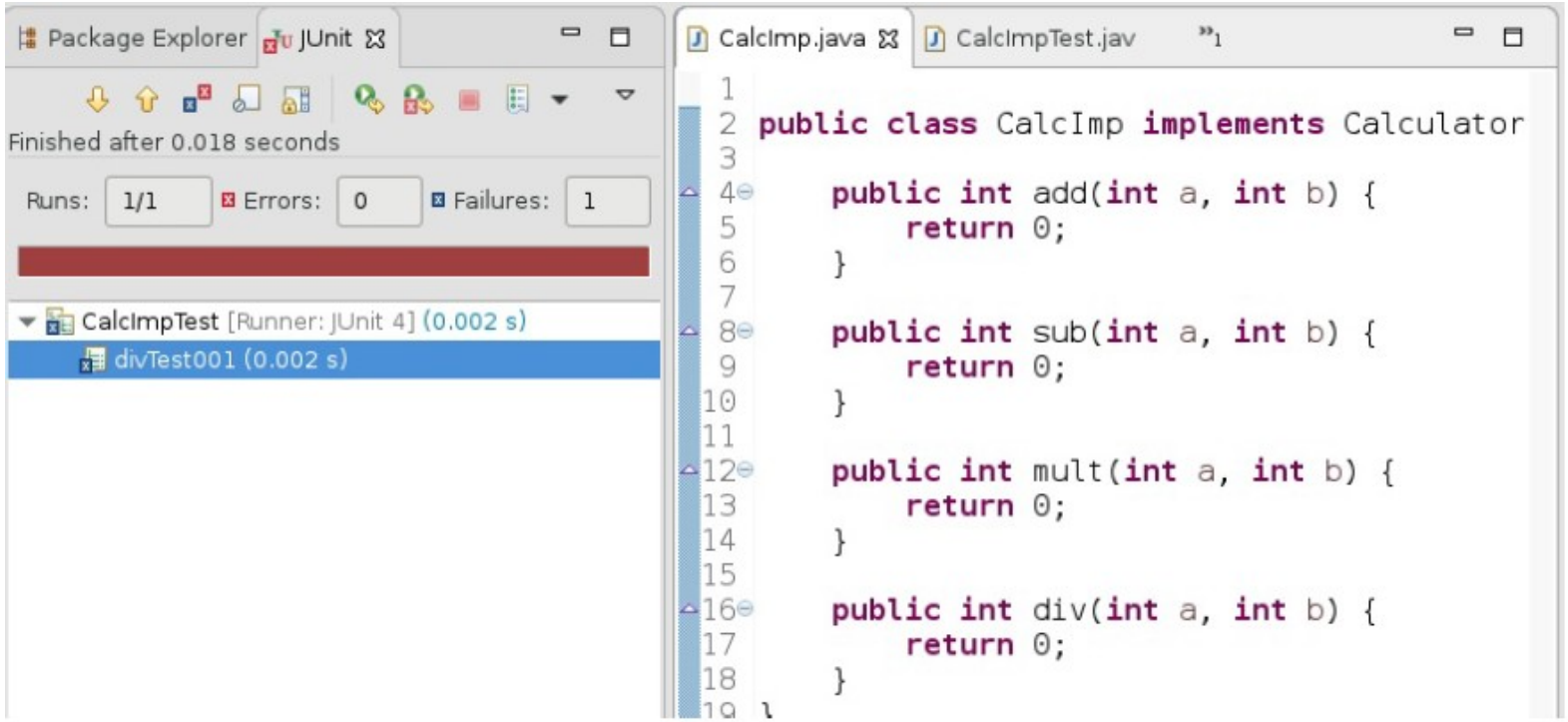
```
1  import static org.junit.Assert.*;
8
9  public class CalcImpTest {
10
11     @Test
12     public void divTest001() {
13         Calculator c= new CalcImp();
14         int retVal = c.div(6,3);
15         if (retVal != 2) {
16             fail("divTest001 failed");
17         }
18     }
19
20     @BeforeClass
21     public static void setUpBeforeClass()
22     }
23
```

***Implementing a Test Case***

Consider test case divTest001() for the Calculator div() method. The test case specifies inputs of 6 and 3 and an expected value of 2. An instance “c” of the calculator object is created and the result of invoking c.div(6,3) is stored in retVal

If retVal is not 2, a fail() exception is thrown and the test fails

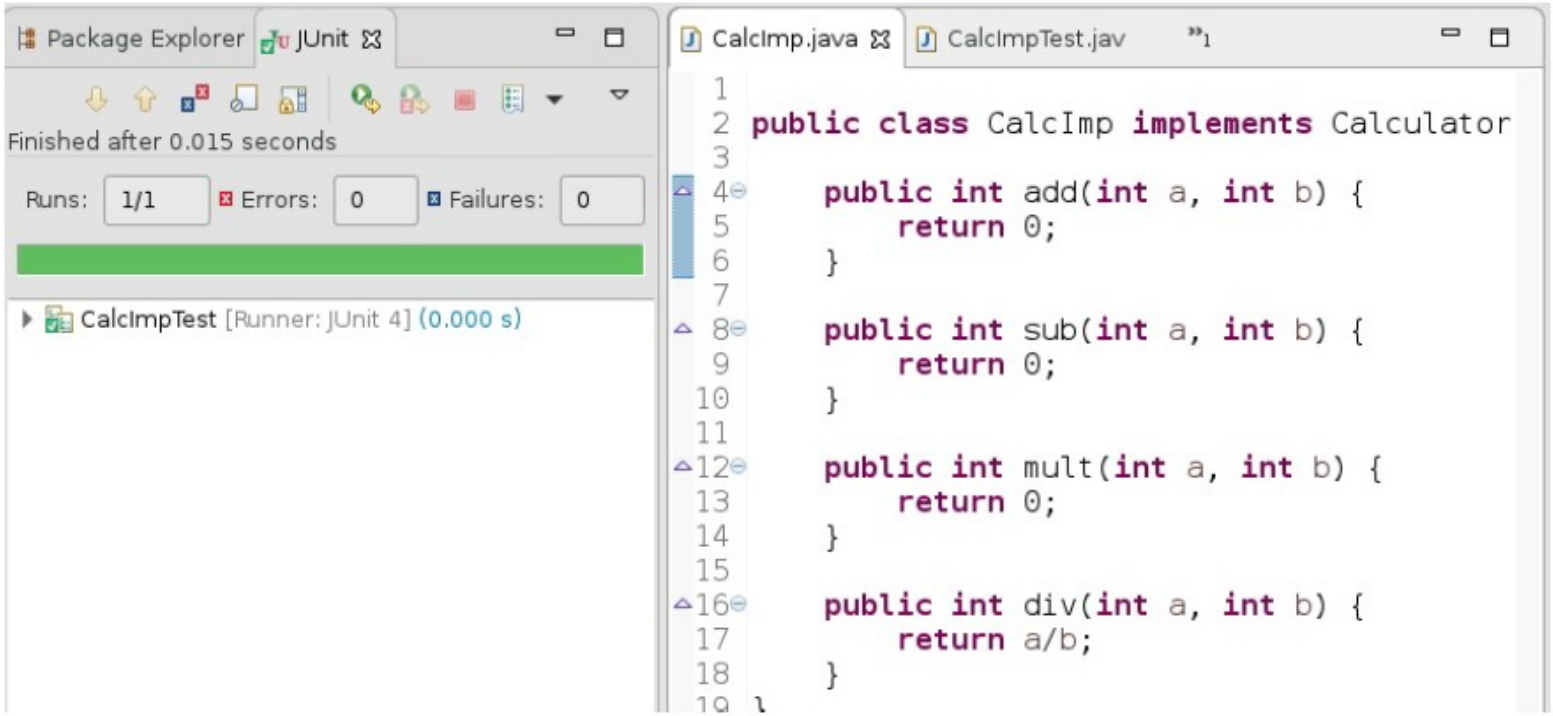
# Failing Test Case




***Test Case Failure***

Running the tests now reports a failure because we haven't yet implemented the production code that makes the test pass.

# Passing Test Case



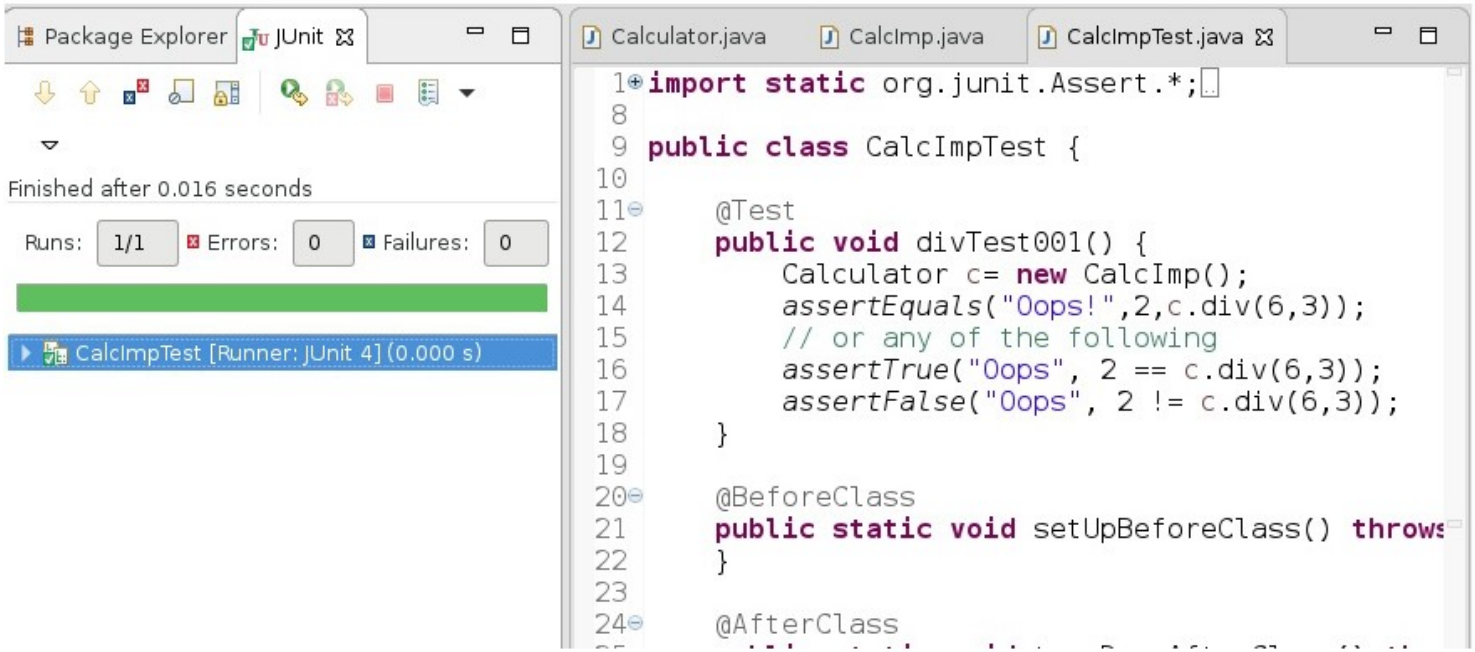

***Passing the Test***

The production code has been added to the div() method and the test now pases.

# JUnit Assertions

- Assertions are statements that evaluate to true or false
  - If an assertion is false then an exception is thrown, otherwise no action is taken
- Assertions express test conditions in a readable form, for example:
  - assertEquals([msg],expected, actual) compares two values, if theyare not equal, the assertion fails (“msg” is an optional message to be printed if the assertion fails)
  - assertEquals(expected, actual, delta) is a form used for floating point numbers where two values are “equal” if |expectedactual|< delta
  - assertTrue(val) where val is a boolean predicate

# Adding Assertions




***Readable Forms of Assertions***

The use of the different forms of assertions allows us to state the test conditions in a much more natural and readable manner. JUnit does not care what form of the assertion is used which means that we choose the one that reads most naturally.

For example, there are three forms used in the test method and JUnit will accept any one of them, but the first assertion used would be preferable because it communicates more clearly what the method is testing to those reading or maintaining the code.

The use of the different forms allows us to write more streamlined and compact code as well.


Test Methods
Demo

# Test Methods

## Lab Testing 3

# Testing for Exceptions

- Sometimes a test expects an exception to be thrown in order to pass
  - The standard assertions do not allow us to check this case
  - Instead we identify the exception to be thrown in the annotation
- The assertion syntax is:
  - @Test(expected = <java exception class> )
- The test will pass only if the specified exception is thrown
  - The test will fail if no exception is thrown
  - The test will fail if any exception other than the specifiedone is throwne

# Testing for Exceptions

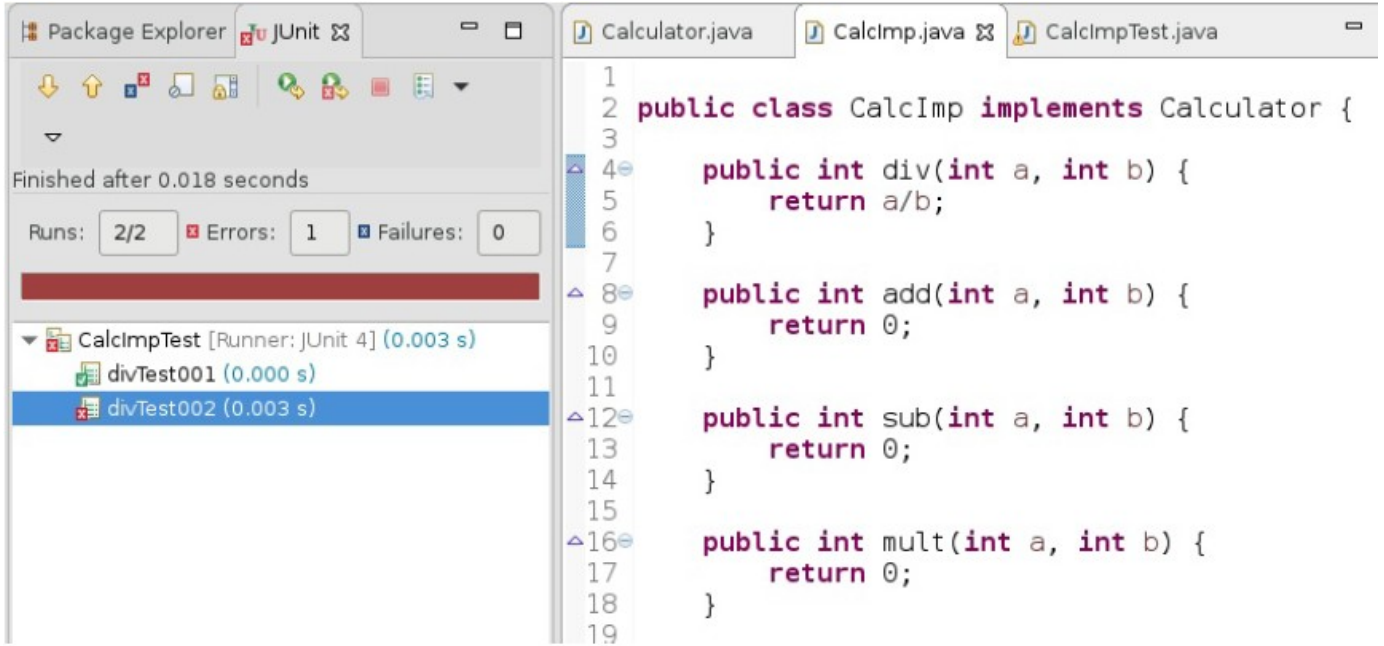
Calculator.java | *CalcImp.java | CalcImpTest.java

```
 1 import static org.junit.Assert.*;
 8
 9 public class CalcImpTest {
10
11     @Test
12     public void divTest001() {
13         Calculator c= new CalcImp();
14         assertEquals("Oops!",2,c.div(6,3));
15     }
16     @Test (expected = java.lang.IllegalArgumentException.class)
17     public void divTest002() {
18         Calculator c= new CalcImp();
19         c.div(2,0);
20     }
21
22 }
```

***The Exception Test Method***

In the second test method, we are checking to see if the production code throws an IllegalArgumentException when the divide method divides by zero.

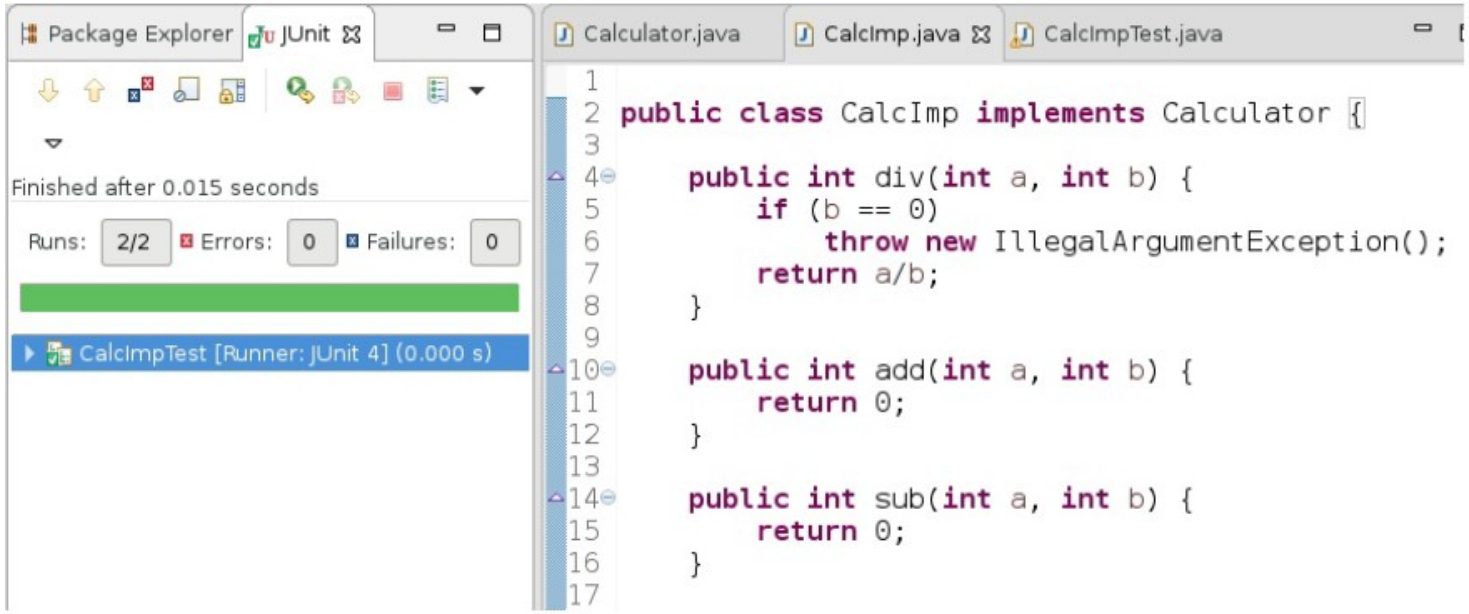
# Testing for Exceptions




***Running the Exception Test***

Since the code to check for a division by zero does not yet exist, the test fails. Notice that this is one time when an error and failure are the same thing. The error occurred because Java threw an ArithmeticException however the test failed because it was expecting an exception but the wrong type of exception was thrown.

# Testing for Exceptions




***Running the Exception Test***

Once the production code is added to perform the check for zero and throw the correct exception, the tests pass.

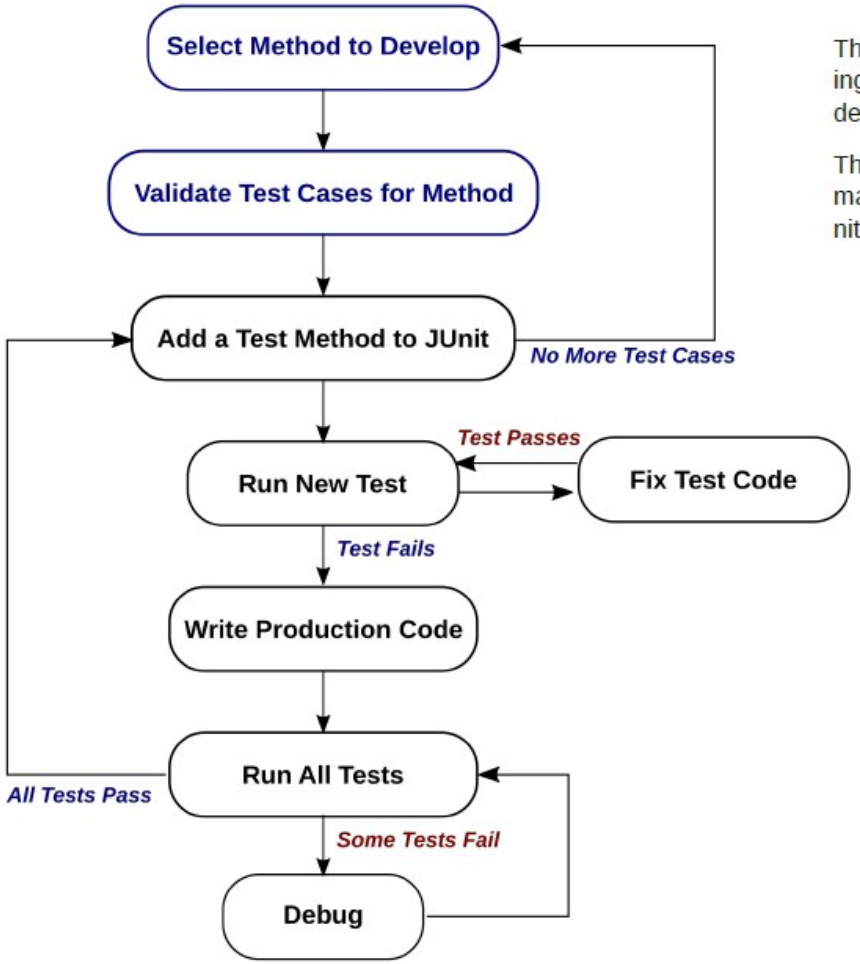

Testing Exceptions
Demo

# Testing Exceptions

## Lab Testing 4


Questions?

# TDD Process




***The TDD Process***

This is general flow of the TDD process for writing code. There are variations of the process described by different authors.

This is also called the "red-green-refactor" TDD mantra because of the color coding used in JUnit.

# Choosing a Method

- Choosing methods to implement in the right order makes development easier
  - Develop the query methods first, then the command methods
  - Query methods do not change the state (internal data) of an object
  - This means we only need to test the return value
  - We may decide to also test invariants to make sure nothing changed as a result of the query
- The command methods usually change object state (internal data)
  - Having fully tested query methods simplifies writing command Junit tests
  - Fully tested query methods can be used to test the object state

# Validate Unit Tests

- Since tests guide development, bad tests results in bad code
- We do not makeup unit test cases “on the fly”
- The set of unit tests for the chosen method already exist
  - They are consistent with the system acceptance tests
  - They are consistent with the design architectural constraints
  - The set of unit tests for the method have been validated
- For this module, we assume that all the tests have been validated

# Adding a Test Case to JUnit

- Adding a test case means writing a test method in the Junit test class to implement a chosen test case from the set of test cases
- The valid or positive test cases are all added before the invalid or negative test cases
  - Adding the positive test cases first creates better code structure than adding them in arbitrary order
  - It is more natural to have the main flow of the code handle all the logic for valid inputs
  - Then the invalid cases are implemented as alternate paths in the flow of the main code
- Adding negative cases too soon tends to produce more convoluted code

# Run All the Tests

- After a test is added all of the tests are run
- This is continuous regression testing
- Ensures that code just added does not break existing code

# Write the Production Code

- The test case represents a class of inputs
  - The added code should work for the class of inputs represented by the test cases
  - In the multiplication test cases below, we write code to satisfy the test classes that are represented by the test cases
  - The specific test cases then test to see that our logic works for that class of inputs
  - We should be able to use different test values from the same test classes and still have the tests pass

| Test Case | Input | Expected Output | Test Class |
|---|---|---|---|
| mult001 | (2,3) | 6 | multiplying two positive numbers |
| mult002 | (2,-3) | -6 | multiplying a positive number and a negative number |
| mult003 | (-2,3) | -6 | multiplying a positive number and a negative number |
| mult004 | (-2,-3) | 6 | multiplying two negative numbers |
| mult005 | (-2,0) | 0 | multiplying a negative number by zero |
| mult006 | (0,3) | 0 | multiplying a positive number by zero |

# Run All the Tests

- The production codes should pass the new test and all previous tests should pass
- If any of the tests fail, debug the code and run all the tests until they all pass

# Problem Spec

***Specification for the Bank Account class***

1. The BankAccount interface provided is to be implemented as a Java class. The BankAccount object is a temporary in memory object created during an interactive session with a user. Whenever a user needs to interact with their account, a BankAccount object will be created for that purpose..

2. When instantiated, the BankAccount object will  populate itself with account data from the bank mainframe database via the BankDB interface. Once the session is completed, the BankAccount object will update the database, if necessary, before the BankAccount object is destroyed.

3. Users can make deposits and withdrawals and can query their balance and available balance.

4. Each bank account has a status code associated with it which is either a 0 if the account is unencumbered or a non-zero number which indicates the account is suspended or partially suspended for some reason. No operations, excluding queries, are permitted on an account with a non-zero status.

5. Each bank account has a transaction limit and a session limit. The transaction limit is the maximum amount the user may request on a single withdrawal. The session limit is the maximum cumulative amount allowed for all withdrawals within a session (i.e. from the time the BankAccount is created until it is destroyed).

6. Users may not overdraw their accounts or exceed any limits associated with the accounts.

7. All amounts deposited or withdrawn must be greater than 0.

# The BankAccount Interface

BankAccount.java

```
1  package bank;
2
3  public interface BankAccount {
4      int getBalance();
5      int getAvailBalance();
6      boolean deposit(int amt);
7      boolean withdraw(int amt);
8
9  }
10
```

***The BankAccount Interface***

1. The BankAccount interface has two query methods and two command methods
2. The query methods return a value but do not change any data in the account.
3. The command methods alter the internal data and return a boolean to indicate whether the command succeeded.

# The BankDB Interface

BankAccount.java | BankDB.java

```
1  package bank;
2
3  public interface BankDB {
4      public int[] getData(int actnum);
5      public void putData(int actnum, int[] data);
6
7  }
8
```

***The BankDB Interface***

Given an account number, the getData() method returns an array of integers of length 5

data[0] contains a 0 if the account is active, or an inactive code

data[1] contains the current balance

data[2] contains the available balance

data[3] contains the per transaction limit

data[4] contains the per session limit

The putData() method takes an integer array of length 3 where

data[0] contains the account number

data[1] contains the current balance

data[2] contains the available balance

# Test Data

- In order to populate our test objects, we need some test data
- The spreadsheet below represents some constructed test data

| Account ID | Status | Balance | Available Balance | Transaction Limit | Session Limit |
|---|---|---|---|---|---|
| | | | | | |
| 1111 | 0 | $1,000.00 | $1,000.00 | $100.00 | $500.00 |
| 2222 | 1 | $587.00 | $346.00 | $100.00 | $800.00 |
| 3333 | 0 | $897.00 | $239.00 | $1,000.00 | $10,000.00 |
| 4444 | 0 | $397.00 | $0.00 | $300.00 | $1,000.00 |
| 5555 | 0 | $0.00 | $0.00 | $100.00 | $500.00 |
| | | | | | |

# Query First

- The first two methods to be implemented are the queries
  - The queryBalance() and the queryAvailBalance()
- Examining the programming contract of the queries:
  - There are no postconditions because the queries do not change any data in the account object.
  - There are no preconditions since the queries always run no matter what the status of the account is.
  - There is an invariant: no the internal data should be changed
- Do we write tests for invariants?
  - It depends on what we call our Good Enough Quality level

# Validate Test Cases

- For now, we use the test cases below assuming they have been validated

| Test Case ID | Account | Expect | Balance | Available Balance | Transaction Limit | Session Limit |
|---|---|---|---|---|---|---|
| BAL001 | 5555 | $0.00 | $0.00 | $0.00 | $100.00 | $500.00 |
| After test | | | $0.00 | $0.00 | $100.00 | $500.00 |
| | | | | | | |
| BAL002 | 2222 | $587.00 | $587.00 | $346.00 | $100.00 | $800.00 |
| After test | | | $587.00 | $346.00 | $100.00 | $800.00 |

# Implementation Class

BankAccount.java | BankDB.java | MyAcct.java

```
1  package bank;
2
3  public class MyAcct implements BankAccount {
4
5      private boolean status;
6      private int balance;
7      private int available_balance;
8      private int transaction_limit;
9      private int session_limit;
10     private int total_this_session;
11
12     public int getBalance() {
13         return 0;
14     }
15
16     public int getAvailBalance() {
17         return 0;
18     }
19
20     public boolean deposit(int amt) {
21         return false;
22     }
23
24     public boolean withdraw(int amt) {
25         return false;
26     }
27
28 }
```

***Creating an Implementation Class***

1. A class is defined called MyAct that implements the interface. In addition to the defined methods, the class also has instance variables to hold the data from the bank database.

2. However the problem is how we get the data into the class. We need to add a constructor that fetches the data from the database when given an account number.

# Constructor Implementation

BankAccount.java | BankDB.java | MyAcct.java

```
1  package bank;
2
3  public class MyAcct implements BankAccount {
4
5      public MyAcct(BankDB b, int actnum) {
6          BankDB myBank = b;
7          int[] data = null;
8          data = myBank.getData(actnum);
9          status = (data[0] == 0);
10         balance = data[1];
11         available_balance = data[2];
12         transaction_limit = data[3];
13         session_limit = data[4];
14         total_this_session = 0;
15
16     }
17
```

***Implementing a Constructor***

1. The constructor as implemented will now populate the account object when it is created. A dependency injection is used to provide the account with a reference to the bank database

2. The problem is that we cannot access the bank database nor should we in a development environment. What we need is an object we can use as if it were the bank database. This sort of stand-in object is called a mock object.

# Mock Objects

- A mock object is an object we use in a test environment that appears to be real object but is just a facade
  - A mock object implements the same interface as the real object, but the implementation just pumps out the right replies to the test case
  - Because the interface is the same as the real object, there is no way for our code to be able to tell the difference between them

  Mocking is a standard technique in TDD
  - We will look at the Mockito mocking library later on

# Hand Crafted Bank Mock

MockDB.java

```
1  package bank;
2
3  public class MockDB implements BankDB {
4
5          public int [] getData (int accountNumber) {
6              int[] data = null;
7              if (accountNumber == 5555) {
8                  data = new int[5];
9                  data[0] = 0;
10                 data[1] = 0;
11                 data[2] = 0;
12                 data[3] = 100;
13                 data[4] = 500;
14             }
15             else if (accountNumber == 2222) {
16                 data = new int[5];
17                 data[0] = 0;
18                 data[1] = 587;
19                 data[2] = 346;
20                 data[3] = 100;
21                 data[4] = 800;
22             }
23             return data;
24     }
25     public void putData(int actnum, int[] data) {
26         return;
27     }
```

***Implementing a Mock Bank Database***

1. The mock database just returns the test data for the account used in the first test case. The putData() method doesn't do anything since we do not use it in our testing.

2. At this point we have added code without testing it. It is a simple matter to write a test script to see that the mock is working correctly. Whether or not we would do this as opposed to just a visual inspection is again a quality decision and is discussed in the student manual

# Using the Mock

MockDB.java | MyAcct.java | MyAcctTest.java

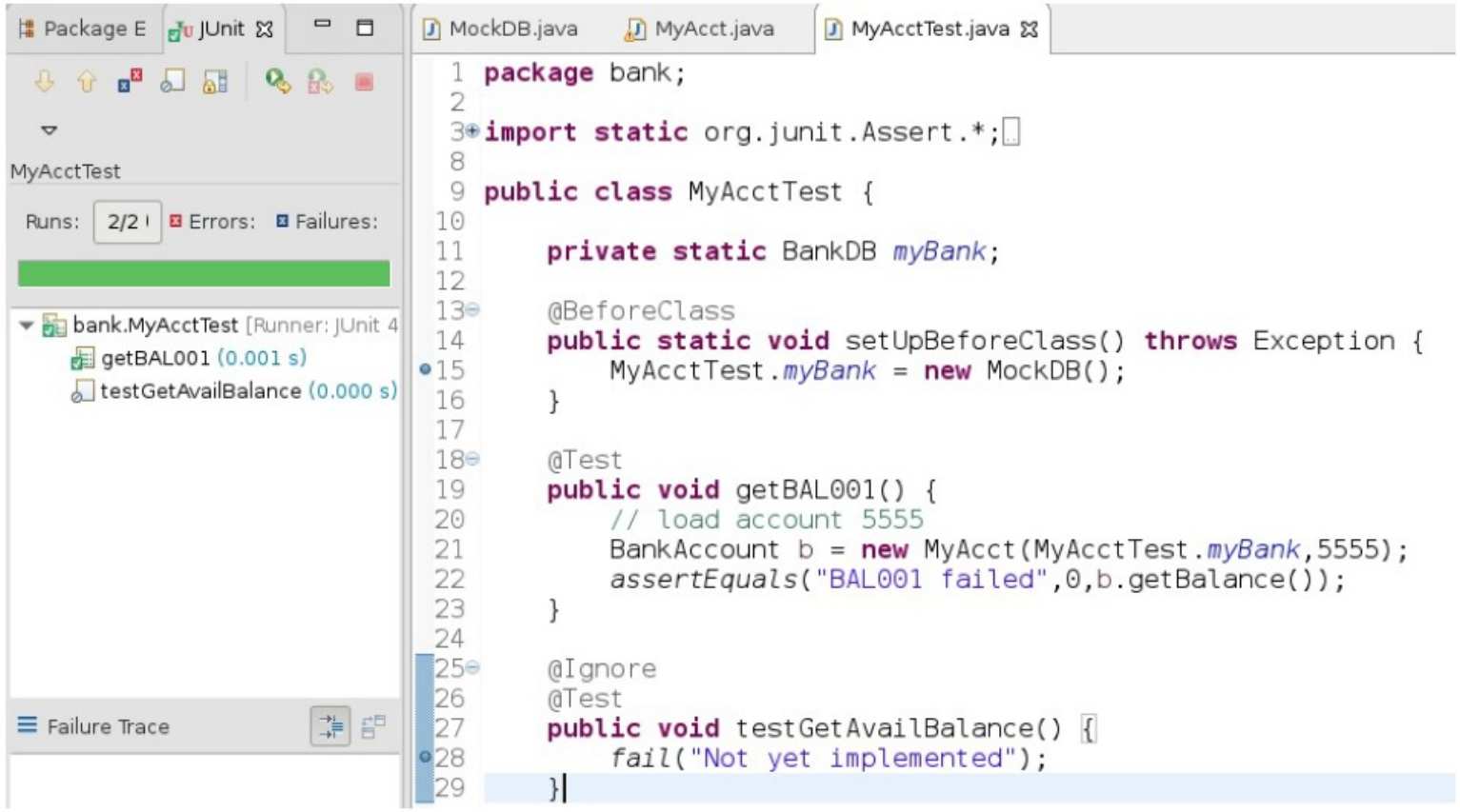
```
1  package bank;
2
3  import static org.junit.Assert.*;
8
9  public class MyAcctTest {
10
11     private static BankDB myBank;
12
13     @BeforeClass
14     public static void setUpBeforeClass() throws Exception {
15         MyAcctTest.myBank = new MockDB();
16     }
17 }
18
```

***Using the Mock Database***

1. After generating the test class, we can use one of the fixture methods to create an instance of the mock bank database before any of the tests are run. Since the mock database will just get garbage collected after the test runner exits, we do not need any tear down methods.

2. Because all of the tests will use the same mock object, we only need to create it once. This object is what will be passed in the dependency injection in the constructor to our account class.

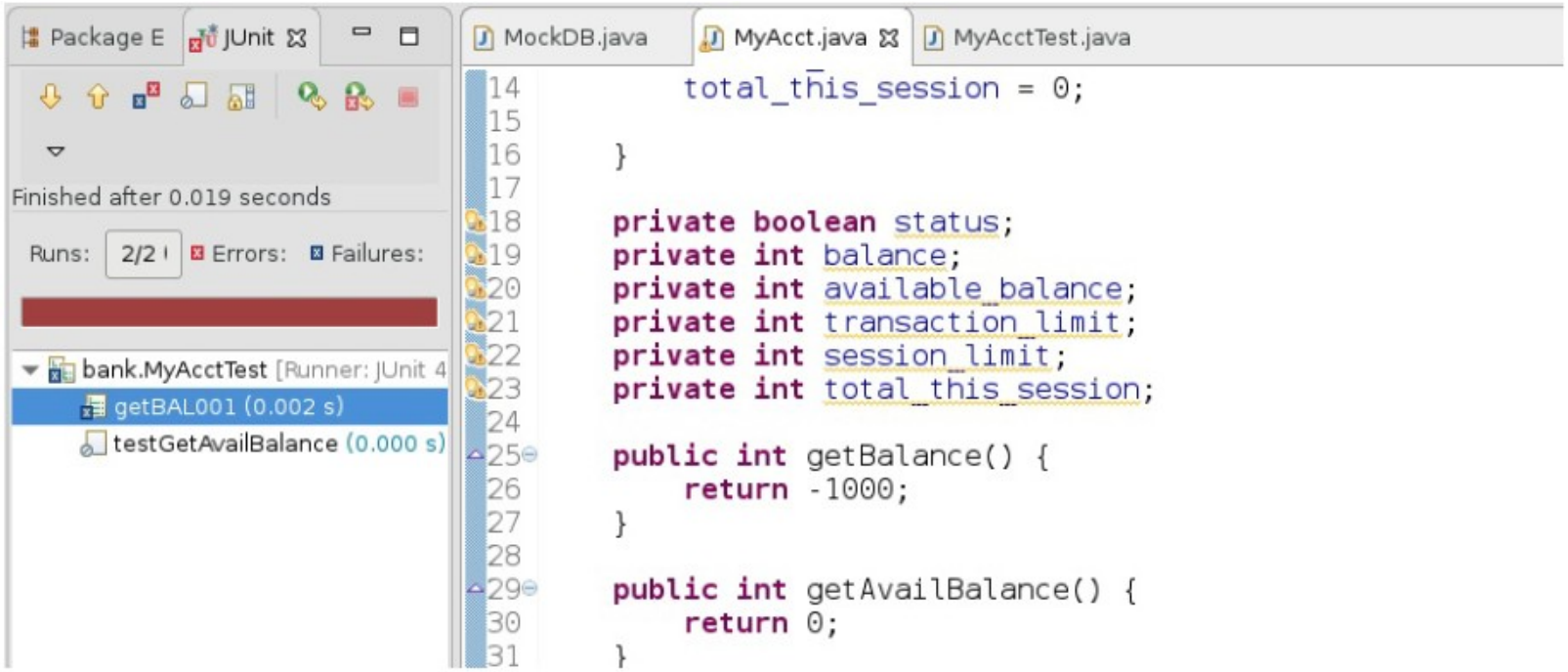
# Adding a Test Case




***Adding Test BAL001***

The test case is for BAL001 is added using the technique we saw for writing JUnit tests in the previous module. However when we run the test, it passes. According to our TDD protocol, it should fail.

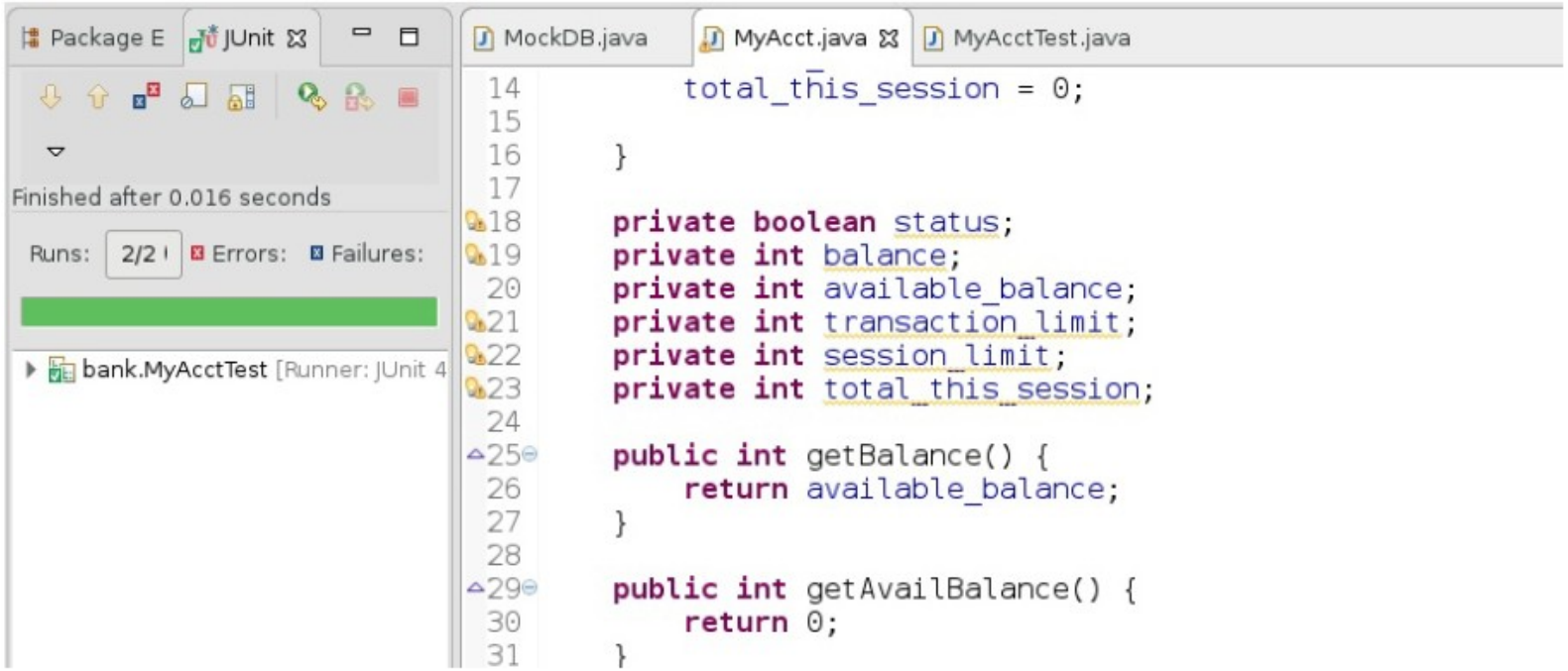
# Correcting the Test Case




***Correcting Test BAL001***

The test passed because the default return value for getBalance() was 0, which as also the expected value for test. Changing the default value of the production method stub produces the required failure. While this is a contrived example, the underlying principle is to always "test the test" since there are a number of places where we could have made errors.

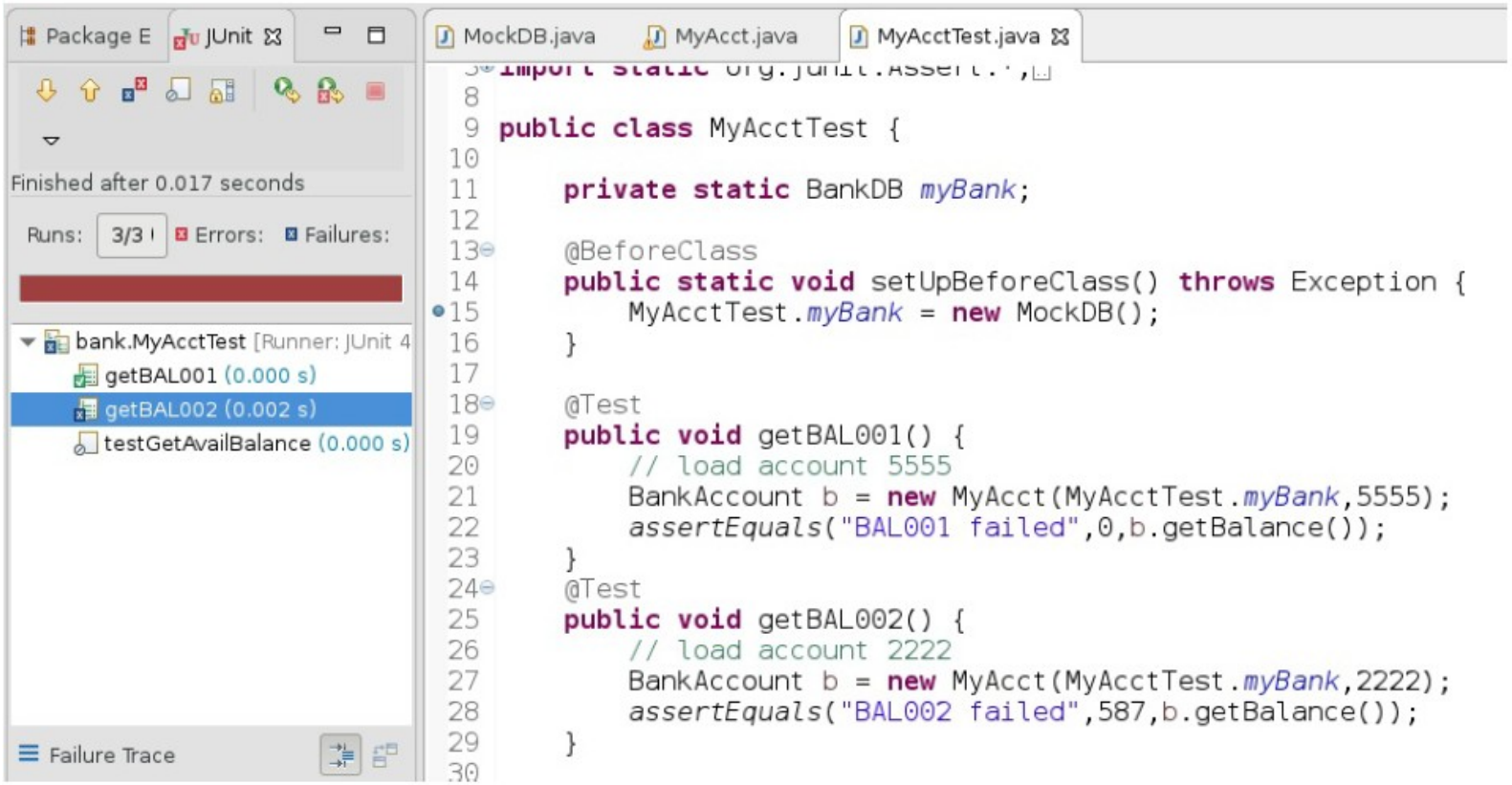
# Write Production Code




***Writing code***

The code is written to make the test pass. However notice that the code is wrong and the test still passes. It just happens that this specific test case will not pick up the programming error. This is called coincidental correctness and an example of why do do not rely on a single test case to see if our code is correct.

# New Test = Regression Test




***Adding the New Test Case***

Adding the new test case and running it picks up the error. If we had not had a robust set of test cases and just assumed that one test would be sufficient, the error would have not been caught. This is why it is important to have a robust set of unit tests to work from.

# Fix the Code


***Fixing the Code***

The programming error is fixed and all of the tests now pass. This example has been made simple enough that the sort of issues that we have to be aware of in the TDD process are obvious. However, in more realistic and complex coding projects, these sort of errors are not obvious on inspection. Following the TDD process has a self correcting quality that allows us to correct these errors as they occur.

# Implementing Commands

- Implementing Commands with TDD
  - Commands change the internal data of the object
  - Testing command return values is often trivial
- Four distinct kinds of tests:
  - Tests to ensure our code does not violate preconditions
  - Tests to ensure our code satisfies postconditions
  - Tests to ensure invariants are preserved
  - Tests to validate side effects

# The deposit() Contract

- Deposit preconditions
  - Account status must be 0
  - Deposit amount must be > 0
- Deposit postconditions:
  - Balance should increase by amount
- Deposit invariants
  - Available balance remains unchanged??
  - The spec does not describe the effect of a deposit on the available balance!
  - All limits remain unchanged

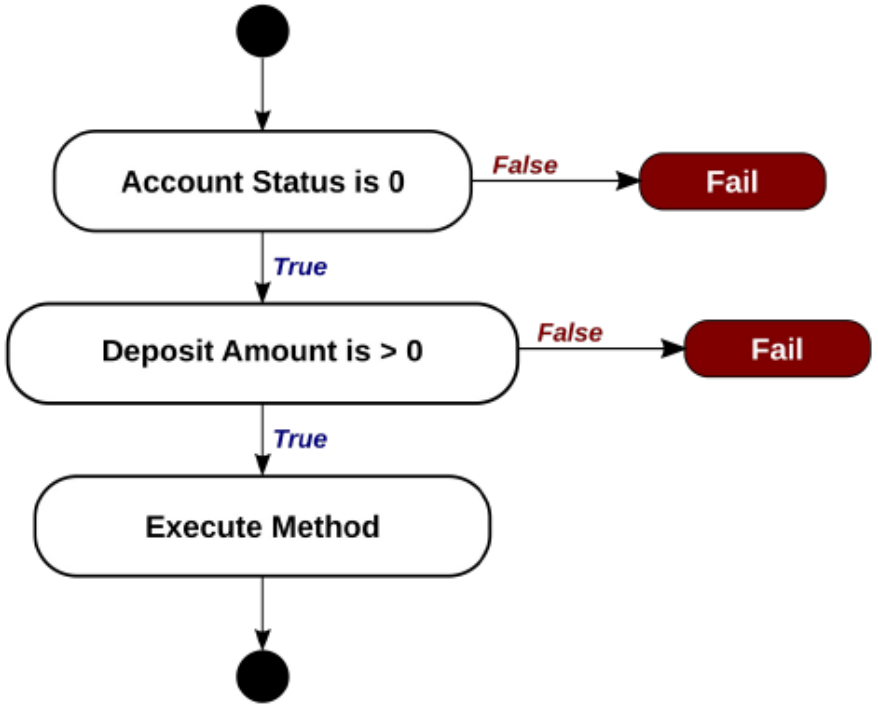
# Clarifying the Contract

- We have two alternatives
  - Deposits increase the available balance
  - Deposits do not increase the available balance
- Both alternative are reasonable but:
  - Making an assumption may result in an error
  - The only way to resolve it is to ask
- After asking the product owner we are told:
  - For fraud prevention requirements, the available balance cannot increase until the deposit is vetted overnight
  - If we had made a wrong assumption: Our tests would be wrong and our code would be wrong

# Testing Completeness for Pre-conditions

- In order to ensure we get a good set of tests:
  - We use structural testing to ensure we cover all the cases
  - We diagram the logic of all the preconditions
  - We ensure each edge is traversed by at least one test
- This is a completeness strategy:
  - It shows us the minimum number of tests needed to satisfy all the preconditions
  - For each test case postconditions and invariants still need to be checked
  - Generally, production code is written to check preconditions but rarely to validate postconditions

# The Structural Test Model




***The Test Logic***

This is very simple since we only have two pre-conditions to worry about but it does show us that we need three tests to ensure we have considered all possible preconditions. While a model this simple could have been done on the fly, when we start to get more complex, this becomes an excellent tool for ensuring nothing has slipped through the cracks.

# The Test Cases

| Test Case ID | Account | Status | Amount | Expect | Balance | Available Balance | Transaction Limit | Session Limit |
|---|---|---|---|---|---|---|---|---|
| | | | | | | | | |
| DEP001 | 3333 | 0 | $1.00 | TRUE | $897.00 | $239.00 | $1,000.00 | $10,000.00 |
| After test | | | | | $898.00 | $239.00 | $1,000.00 | $10,000.00 |
| | | | | | | | | |
| DEP002 | 2222 | 1 | $1.00 | FALSE | $587.00 | $346.00 | $100.00 | $800.00 |
| After test | | | | | $587.00 | $346.00 | $100.00 | $800.00 |
| | | | | | | | | |
| DEP003 | 1111 | 0 | -$1.00 | FALSE | $1,000.00 | $1,000.00 | $100.00 | $500.00 |
| After test | | | | | $1,000.00 | $1,000.00 | $100.00 | $500.00 |

***The Test Cases***

To fully test the pre-conditions, we can choose the three cases listed above to ensure coverage of the pre-conditions. In terms of order of implementation, we have one valid case (the one that returns true) and two invalid cases (the ones that return false) so we add the valid case first.

# Implementing A Test Case

```
 9 public class MyAcctTest {
10
11     private static BankDB myBank;
12
13     @BeforeClass
14     public static void setUpBeforeClass() throws Exception {
15         MyAcctTest.myBank = new MockDB();
16     }
17
18     @Test
19     public void depositDEP001() {
20         // load account 3333
21         BankAccount b = new MyAcct(MyAcctTest.myBank,3333);
22         // transaction accepted
23         assertTrue("DEP001 failed",b.deposit(1));
24         // post-conditions and invariants
25         assertEquals("DEP001 wrong balance",898,b.getBalance());
26         assertEquals("DEP001 wrong avail bal",239,b.getAvailBalance());
27
28     }
```

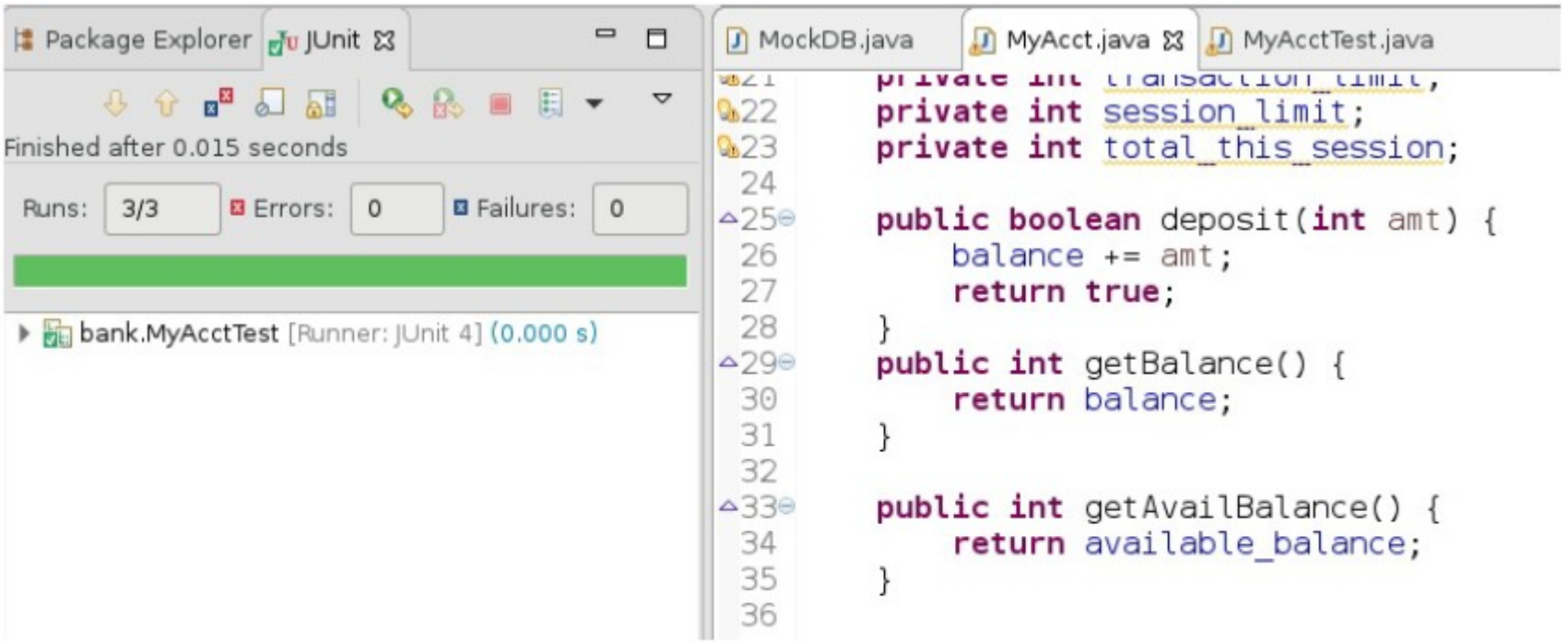
***Implementing a Test Case***

Notice the order of the assertions. We want to test first that the method executes so the first assertion checks the return value. Then we check the post-conditions and invariants that have to be true after the method executes.

# Using Multiple Assertions

- In the previous example, multiple assertions were used
- We can place assertions anywhere in a test method we want
- All of the assertions have to pass for the test method to pass
- As soon as one assertion fails, the test is marked as a failure and the test method aborts

# Writing the Production Code




***Writing Production Code***

After confirming that the test fails, production code is added to make the test pass.

# Adding the Other Tests

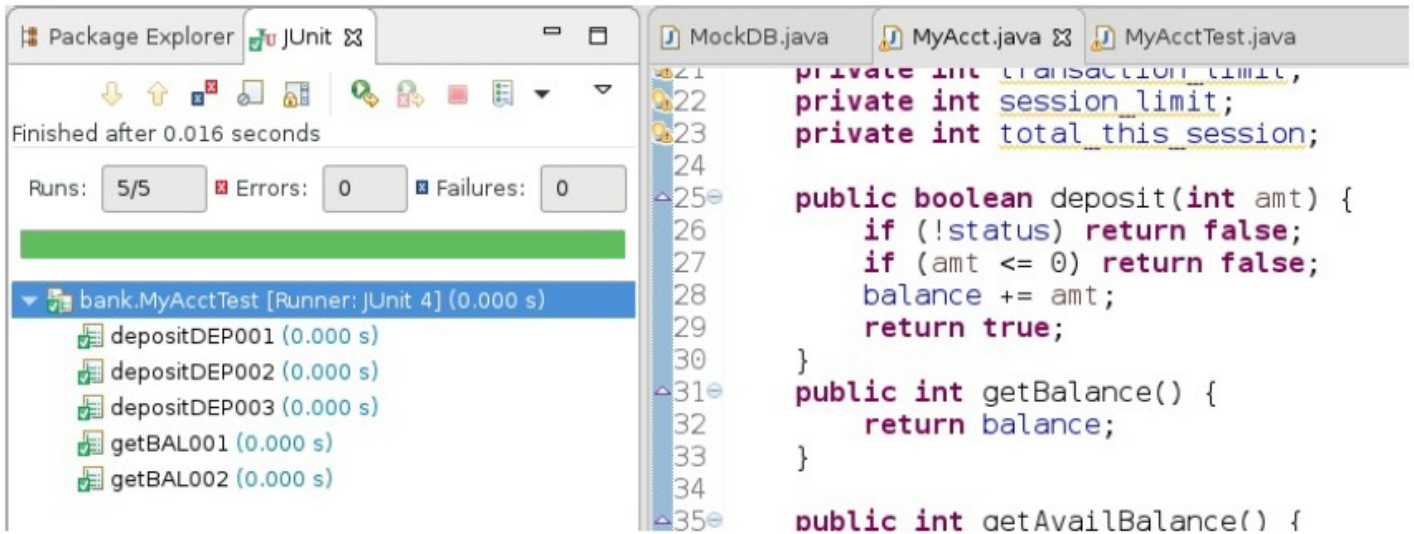
MockDB.java | MyAcct.java | MyAcctTest.java

```
26          assertEquals("DEP001 wrong avail bal",239,b.getAvailBalance());
27      }
28      @Test
29      public void depositDEP002() {
30          // load account 2222
31          BankAccount b = new MyAcct(MyAcctTest.myBank,2222);
32          // transaction accepted
33          assertFalse("DEP002 failed",b.deposit(1));
34          // post-conditions and invariants
35          assertEquals("DEP002 wrong balance",587,b.getBalance());
36          assertEquals("DEP002 wrong avail bal",346,b.getAvailBalance());
37      }
38      @Test
39      public void depositDEP003() {
40          // load account 1111
41          BankAccount b = new MyAcct(MyAcctTest.myBank,1111);
42          // transaction accepted
43          assertFalse("DEP003 failed",b.deposit(-1));
44          // post-conditions and invariants
45          assertEquals("DEP003 wrong balance",1000,b.getBalance());
46          assertEquals("DEP003 wrong avail bal",1000,b.getAvailBalance());
47      }
48
```

***Writing Production Code***

After confirming that the test fails, production code is added to make the test pass.

# Running the Tests




***Writing Production Code***

After confirming that the test fails, production code is added to make the test pass.

# Constructors

- Constructors create objects
  - Constructors must ensure the object is in a valid initial state
  - We always test our constructors
  - But we need a specification for what a valid initial state means and how to assess it
- General OOP rule:
  - If a constructor cannot create a valid object, it throws an exception
  - For complex objects, most OOP languages will undo any results of code executed in the constructor prior to throwing the exception

# Constructor Spec for Bank Account

***Constructor Specification***

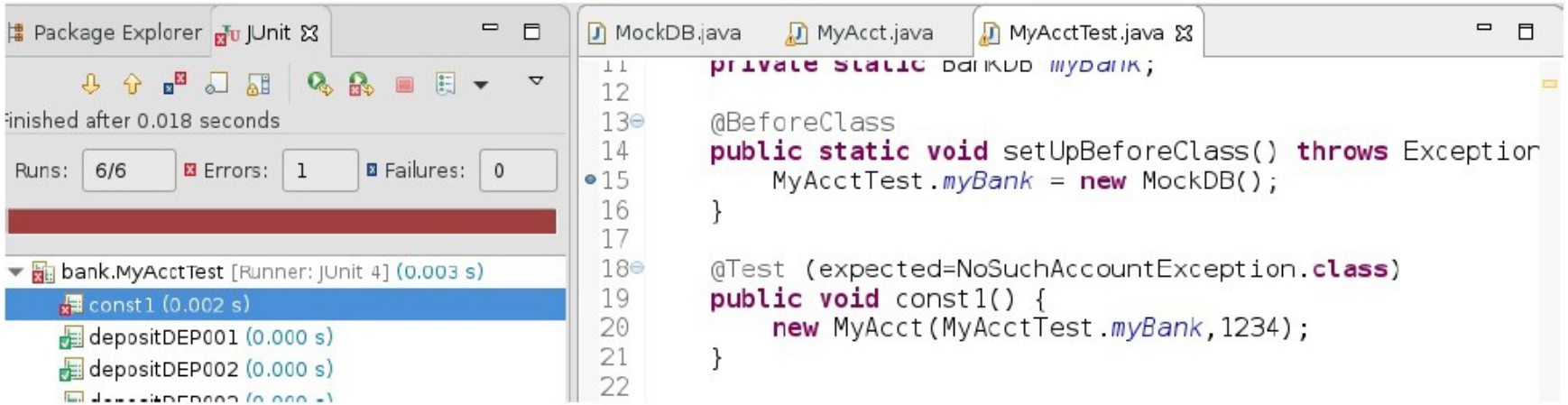
For the bank account to be valid,the following must be true:

1. The bank account must exist in the bank data base
2. All data values are >= 0
3. The available balance <= balance
4. The transaction limit <= session limit
5. The state is an integer from 0 to 10

Two unchecked exceptions will be used. The NoSuchAccountException will be thrown if the account does not exist in the bank database, and the AccountDataException will be thrown if any of the other rules are violated.

```
47 class NoSuchAccountException extends RuntimeException {}
48 class AccountDataException extends RuntimeException {}
49
```

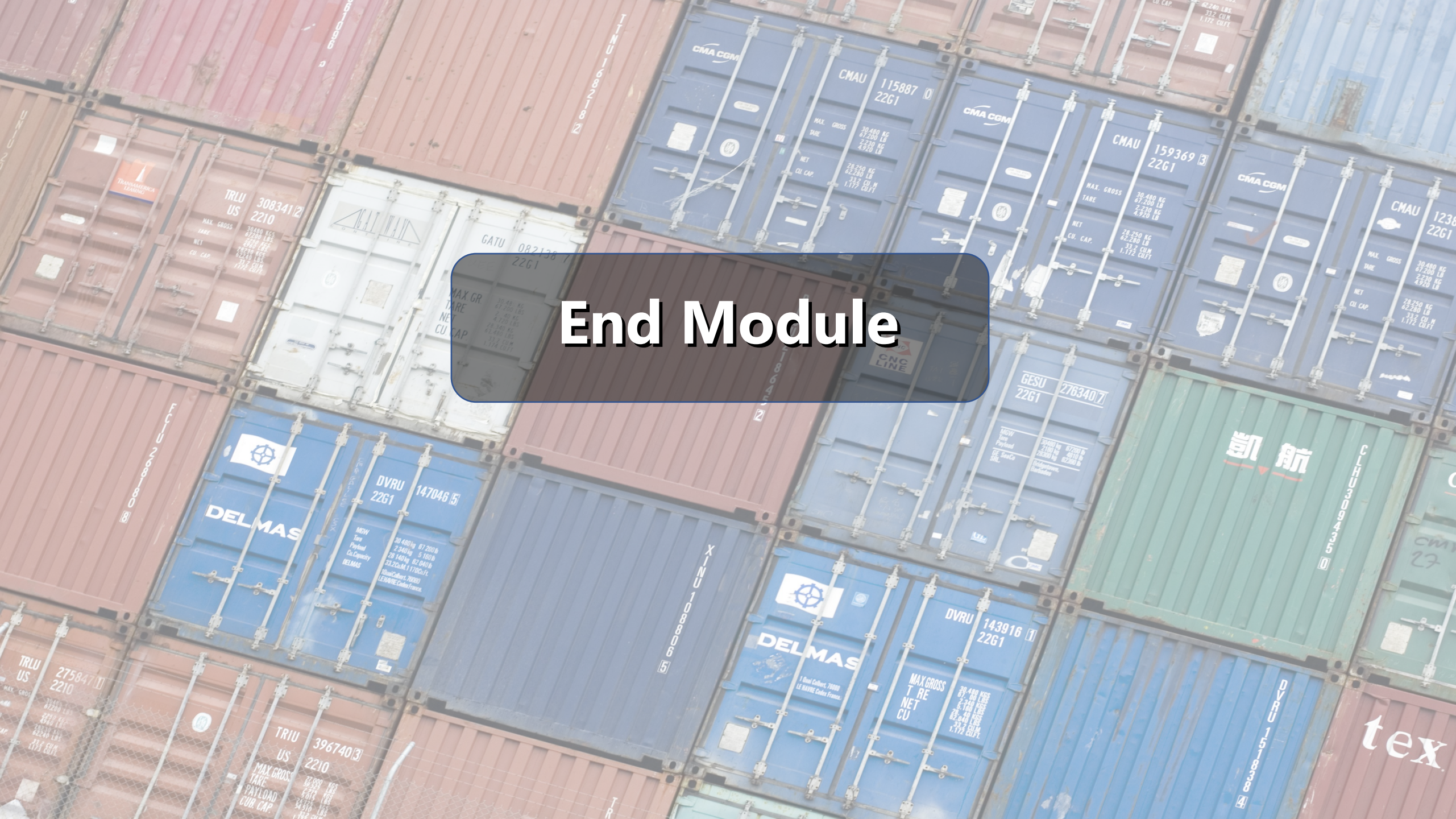
# Adding the Constructor Test




***Constructor Test***

The test for the NoSuchAccountException is added. Note that when we run it, the fact that we have a null pointer exception means that the test ends in an error rather than a failure. This often happens when we are doing exception tests because failing to throw an exception often results is some other error occuring.

# Class Project Discussion


End Module