Copyright Notice

Agile Software Development with TDD

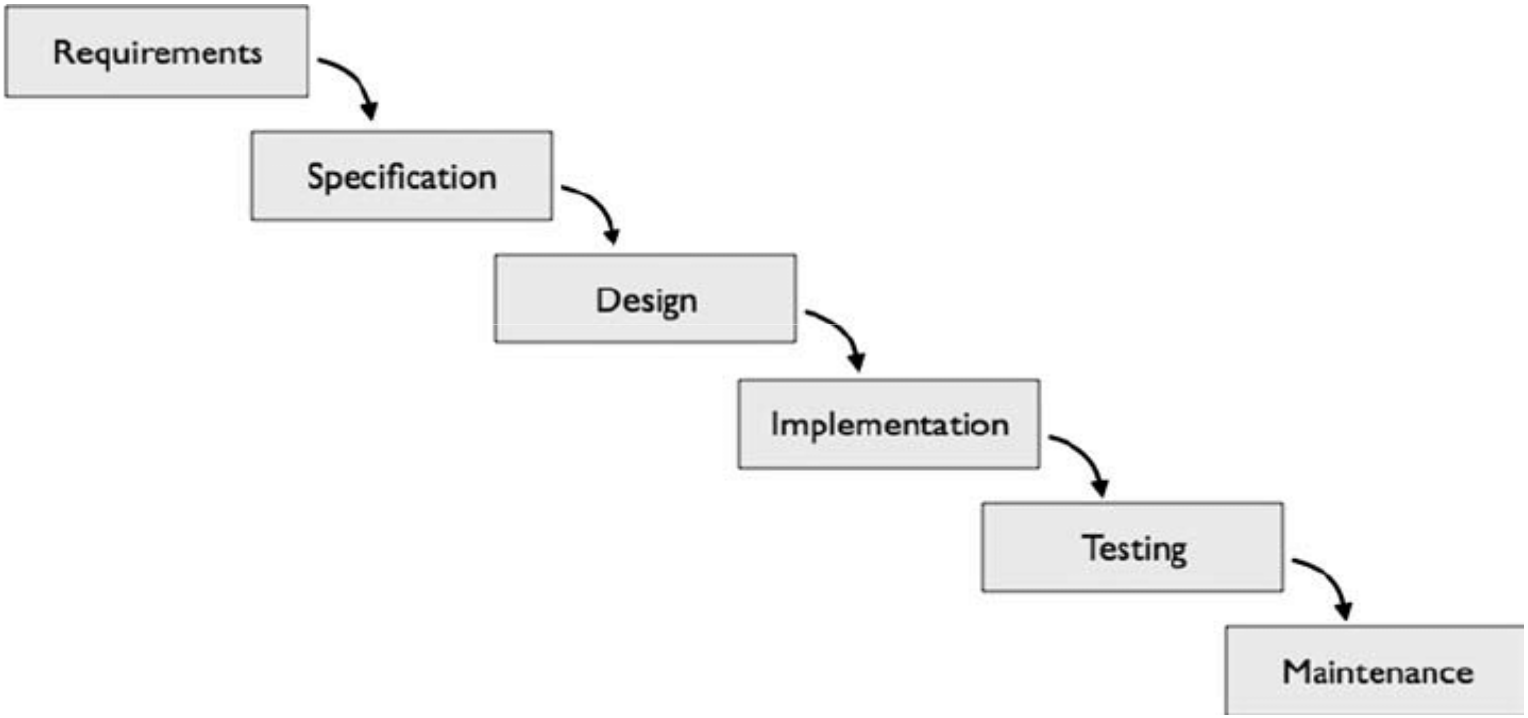
Prakash Badhe

Mail : prakash.badhe@vishwasoft.in

About Me: Prakash Badhe

- ✓Technologies Consultant-Trainer for 20+ years
- ✓Passion for technologies and frameworks
- ✓Promoting Agile Technical Practices
- ✓Skilled in application frameworks
- ✓Working with JavaEE, Spring,Hibernate,JUnit,EJB3.x components.
- ✓Developed applications in enterprise domain.
- ✓Proficient with web services,Html5,Ajax,XML standards
- ✓Working with client side Web Technologies

SDLC in Waterfall



Waterfall model Drawbacks

- Dependencies on sequential stages
- Entire process repeats for change management.
- Testing the application ONLY at the last stage
- In case of issues/bugs revert back the process to starting stage again.
- Tedious to accommodate change
- Islands (isolated)of information

Agile Model of Development

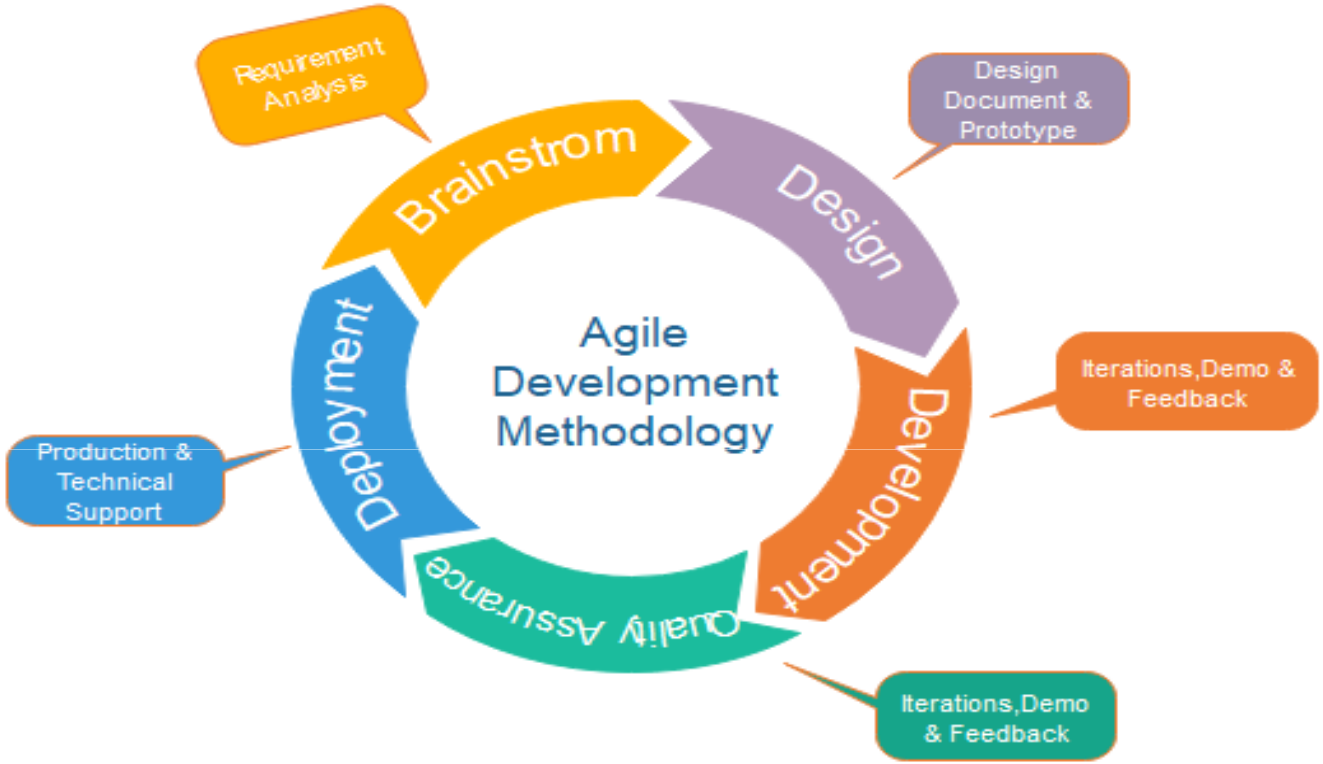


Fig. Agile Model

Agile means to be able to move quickly and easily.

What is Agile ?

- Agile software development relates to process of project management in software development.
- Being Agile means the division of tasks into short phases of work and frequent re-assessment and adaptation of plans.
- Goal of agile is quick and frequent delivery of software applications.
- The agile process replace initial high-level design with frequent redesign in software application development which is required by change in requirements.
- Being agile improves software quality and responsiveness to changing customer requirements

How to be Agile ?

- Agile Manifesto by agile groups
- Extreme Programming Practices
- Small team size
- Pair Programming
- Continuous Testing
- Code Review and analysis
- Coding Practices and standards
- Coder disciplines

Agile Methodologies

- Scrum
- Kanban
- SAFe Agile
- Lean Agile
- DevOps tools and practices

Being Agile..?

- Small team (with people having different skills)
- Customer involvement
- Frequent delivery of M**inimum Viable Product** (MVP)
- *MVP is a product with just enough features to satisfy early customers, and to provide feedback for future product developments.*
- Coding in iteration and small steps
- Smaller increments in delivery
- **Continuous testing** and integration
- Automate the process as much as possible.
- **Everyone is aware and involved in the process**.

Benefits of being Agile

- Manage changes with less efforts
- Ability to quickly respond to changes.
- Reduced bugs in the product.
- Developer confidence
- Team involvement
- Customer focused approach
- Easier and flexible for maintenance and upgrade
- Smooth troubleshooting

MVP-Minimum Viable Product

- Think about incrementally developing the software only if we can break it into smallest possible, valuable pieces.
- The smallest possible increment to a product that can be implemented giving business value is called as Minimum Viable Product (MVP) e.g. Initial prototype and next on-going additions applied in delivery stages.

Traditional look at Design

- Understand the requirements or specifications as an input.
- Produce a abstract structure of the individual classes or components and/or their combinations.
- The above structure can be implemented using a programming language in an environment.

Incremental versus “All at once” development

- Incremental development allows us to remove the risks associated with

1. Changes in the requirements
2. Misunderstanding the expected behavior of the system

It is better to grow the system than planning for it upfront at all levels of details.

What is needed for Incremental Development?

- Definition of Architecture
- Definition of Minimum Product Features(prototype or current need of specifications)
- Environment for automated tests
- Environment for Continuous integration
- Development Practices like TDD.

Traditional look at Testing

- ✓ Exercising On the Test class to verify if it works as per its specification ***after*** the development.
- ✓ Testing to verify the **correctness** of the code!
- ✓ Testing to detect any side effects after the change-Regression testing.

Testing Process

- Writing test code is an art of programming!
- The testing application/class acts as a ‘vise’ for the object/function under test on which the test functions are invoked by simulating the inputs.
- The output is verified as per the expectations.
- If any output is not matching, it is fixed in the application code and re-testing done to verify the change.

Implement the Test code

- Create the object/s
- Invoke the functions
- Verify the results
- If the results are not matching ,modify the class code and again run the function.
- Proceed with more test functions with combinations of inputs and variables.

Testing Framework

The testing framework

- Allows to write test methods as test cases in standard formats.
- Executes the test cases
- Verify/Assert the test results
- Combine multiple test cases
- Generate the test results.
- Automates the testing process.

Test Runner

- Executing the TestRunner will run the tests.
- If all the tests pass, you'll get an informative message.
- If any fail, you'll get the following information:
- The name of the test case that failed
- The name of the source file that contains the test
- The line number where the failure occurred

Success, Errors and Failures

- The test results distinguishes between failures and errors.
- A failure is detected as a result of comparison made on test results.
- Errors are unanticipated problems,like compiler error,division by zero and other exceptions thrown by the runtime or the code.
- In case of no errors or failure ,the test is considered as success.

Testing phases

- Unit testing : white box testing
- Integration testing : black box testing
- Acceptance testing : end to end product specification testing(verification).

Unit testing

- In procedural codes: units are functions
- In OO code: units are classes.
- Example : Testing the Java code of class.

Integration Testing

- The application release is tested in integration phase where more number of units developed by different teams are combined and integrated together.
- Example : Test the interaction between two or more classes/units.
- During integration testing compatibility, interfacing ,connection problems are identified and to be resolved at each module stage and then integrated and tested again.

Acceptance Testing

- Testing conducted on the final product released by the development team in production environment.
- Example : Web application testing with Selenium
- Requirements matching, customer's understanding about the product is verified here and in in case of issues it is resolved at integration as well as module stages and tested again after the build stage in deployment.

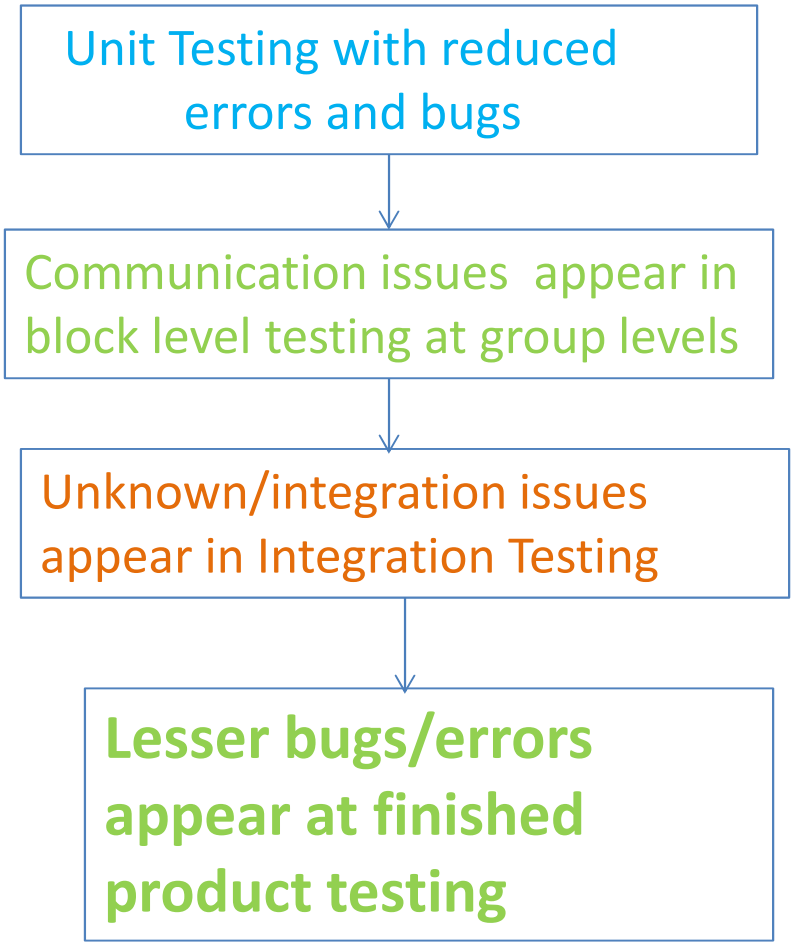
How to reduce the bugs..?

- Follow proper coding guidelines
- Code review
- Testing at every stage
- Regression testing
- **Ensure that the product follows the requirements..**

Product Testing

- Finished product testing : More severe bugs
- Testing during integration : communication/understanding/unknown/integration issues.
- Testing of individual blocks : bugs easily controlled at group/tester level.
- **Unit Testing**
  - **The bugs easily detected and controlled at developer level.**
  - **Bugs mainly because of requirements understanding.**

Effect of Unit Testing



How to unit test the code..

- Units are functions, classes,files,blocks etc.
- How to test the units…
  - Function testing in procedural programming
  - Class testing in object oriented programming
- Should we test the class after it is developed completely…
- Should we test it one by one functions of the class after it is written…

Side Effects of Development

- **Some units developed may not be used in actual requirements.**
- ***How to ensure that the units follow the requirements and nothing more and nothing less..?***

Requirements based testing

- Understand the requirements in steps and not in complete.
- Focus on developing one small part of the requirements.
- **Understand the requirement clearly and the code has to match it.**

Match the requirement to Units

- Unit test ensures that the requirement has been fulfilled by the unit.
- Who should write the unit test..?
  - Tester/Developer
- When should the unit test be written..?

Discussion point: Testing Approach

- Are there any problems with Design First approach?
- Are there any advantages of Test-First Approach?
- Design-by evolution or by creation?

First Chicken or First Egg..?

- Which should be first..
- Code or Test..?

Code first and then test it…

- **Test after the unit has been designed/coded.**
- Tests tend to verify implemented features in the code and NOT what is required in the code.
  - The designed code may have more features than required to pass Unit testing and these features may not get exposed/tested/utilized.
  - The designed code may not match the requirements .

Side-effects of after testing

- The test may not cover all the requirements.
- Difficult to diagnose the defects/side effects in the designed code even after successful testing!.
- Testing the code after the designing phase is more towards waterfall model and difficult to manage change in the designed code, once it is tested.

Test before the coding ..?

- **Unit test written before application coding**
- Unit test is written to verify the requirements and NOT the code features.
- Which code this unit test will verify ..?

Code should follow the test..

- **The code should be written just to make the test success and nothing more or nothing less.**
- **The code indirectly and exactly matches the requirements.**
- **The designed code is just to make the test succeed will NOT have more features than required to pass Unit testing .**

Advantages of Test first approach

- **The test exactly verifies the requirements.**
- **The code follows test is Agile model and easier to manage change in the requirements, at any stage.**
- **Easier to diagnose the defects/side effects in the designed code if any, during design and even after the development is over!.**

Driving the Code=>

- Which factor/item should drive the code development ..?
- Requirements
- Developer skills
- Programming Platform
- **Requirements get mapped to test cases.**
- **The implementation code is written to satisfy the test cases.**
- **Tests Drive the Code.**

Test Driven Development

- NOT a build and test approach..
- We don't start to write the implementation code initially.
- Write test first approach…
- The test first approach
  - The tests are governed by the requirements.
  - The tests govern the implementation code.
- Write the only needed code to succeed the test,no more..no less..

TDD Goals

- The primary goal of TDD is not testing in itself, but rather code design and evolution.
- TDD is an iterative process where iterations are typically short and always start by writing a unit test which describes some specific behavior.

Focus in TDD

- By starting with the test (or "spec"), the programmer is encouraged to focus on the behavior of the system rather than the implementation.
- The implementation comes second, and is typically changed frequently as the system grows through refactoring.
- Because the system is always approached through unit tests, the resulting code is inherently testable and usually loosely coupled, which can be a hard goal to reach when developing software the traditional way.

TDD : A style

- TDD is a programming strategy.
- TDD it does not make a whole lot of sense for testers to be doing it unless they are actually writing code.

Steps in Unit-testing

The steps involved in unit-testing

- Create the object under test
- Invoke the method
- Verify the results/return of method
- If the results/output are not matching ,test fails;
  →modify the code and again test the method.
- Proceed with more test methods

Unit-Test Framework

The unit-testing framework automates the testing process.

- Allows to write test methods as test cases
- By writing test methods
- Marked with annotations
- Executes the test cases
- Verify/Assert the test results
- Combine multiple test cases
- Generate the test results.

Unit testing frameworks

- Junit : Java code testing
- CppUnit: C++ code testing
- Nunit : MS.Net code testing
- TestNG: Extension of JUnit

The xUnit Standard

- The frameworks based on a design by Kent Beck.
- The **xUnit** frameworks allow testing of different elements (units) of software, such as functions and classes.
- provide an automated solution to execute the same tests many times, and no need to remember what should be the result of each test.

JUnit

- JUnit originally written by Erich Gamma and Kent Beck.(junit.org)
- JUnit is an open source Java testing framework used to write and run repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks.

CppUnit

- The CppUnit is porting of JUnit for the C++ platform.
- The CppUnit is also open source and freely available as downloadable source code from sourceforge.net web site.

NUnit

- Nunit is the test framework for MS.Net platforms.
- The Nunit is also open source and freely available as downloadable source code as well as installable from nunit.org web site.
- NUnit is directly usable inside Visual studio as well as integrated with Visual studio add on as TestDriven.Net

JUnit as xUnit Framework

- JUnit originally written by Erich Gamma and Kent Beck.(junit.org)
- JUnit is an open source unit-testing framework for java used to write and run repeatable tests.
- It follows the xUnit architecture for unit testing frameworks.

JUnit Features

The JUnit features include:

- Define the test cases by extending the TestCase class or decorating test methods with annotations
- Provide test fixture to execute the test cases
- The Test fixtures for sharing common test data
- Supports the assertions for testing/verifying expected results
- Test suites for easily organizing and running tests
- Graphical and textual **Test Runners**

Test Fixture/Test class

- A fixture is a known set of methods that serves as a base for a set of test cases.
- Fixtures come in very handy when you are testing as you develop.
- Fixture allows to share and reuse common values across the tests.
- TestFixture::setUp()
- -- to initialize the test variables/data
- TestFixture::tearDown()
- -- to release any permanent resources allocated in the above setUp()

Test Case in JUnit4.0 onwards

- The JUnit4 recognizes any test method decorated with '@Test' annotation as test case in any class.
- You can write multiple test methods in single test class.
- Inside each test method the test case result can be verified with assertions supported by JUnit.

Annotated test-case

```
public class AccountTest {

  @Test
  public void testAccountDeposit()
  {
    Account ac= new Account(12000);
    ac.deposit(1000);
    Assert.assertEquals(11000, ac.getBalance());
  }
```

If the assert method throws exception it is treated as test failure otherwise success.

Test Success or Failure

- The success or failure of a test method is defined by the outcome of a comparison method that compares(asserts) the results.
- If the comparison (assertion) returns false , an **AssertionFailedError** exception is thrown which indicates the failure of the test.
- If no exception occurs in test method it is treated as success.

A set of assertions

- The comparison of test results is done by a set of assert methods that compare the results and determine success or failure of test case.
- The class **Assert** provides **a** set of static methods to assert the values of different types.
- Messages are displayed only when an assert fails or produces an error.

Test Result verifications

The results of the test execution are verified by Assert set of functions in the Nunit framework.

- ✓Assert. AreEqual
- ✓Assert.AreNotEqual
- ✓Assert.AreNotSame
- ✓Assert.AreSame
- ✓Assert.Fail
- ✓Assert.Greater
- ✓Assert.Ignore
- ✓Assert.IsEmpty
- ✓Assert.IsFalse
- ✓Assert.IsTrue
- ✓Assert.NotNull
- ✓Assert.IsNull

If any of those Assert statements return false, the test failure is detected.

Assert methods

- **assertEquals**(boolean expected, boolean actual)
- **assertEquals**(java.lang.String message, byte expected, byte actual)
- **assertNotNull**(java.lang.Object object)
- **assertSame**(java.lang.Object expected, java.lang.Object actual)
- **assertNotSame**(java.lang.Object expected, java.lang.Object actual)

Test objects sharing

- For all the test methods in a single test class i.e. testFixture) a single instance of the Test class is created and shared by the framework for every test method.
- The initialization and closure methods are invoked as part of every test method life cycle.

Fixtures in the test case

- Test Setup:
  - Use the **`@Before`** annotation on a method containing code to run before each test case.
- Test Teardown:
  - Use the **`@After`** annotation on a method containing code to run after each test case.
  - These methods will run even if exceptions are thrown in the test case or an assertion fails.
- It is allowed to have any number of these annotations.
  - All methods annotated with **`@Before`** will be run before each test case, but they may be run in any order.

Static Fixtures at class level

- Test Class Setup:
  - Use the **@BeforeClass** annotation on a method containing code to run once before when the test class starts running the tests.
- Test Class Teardown:
  - Use the **@ AfterClass** annotation on a method containing code to run after all the test cases have been finished.

TestSuite class

- The TestSuite is a container of TestCase methods and TestCase objects and other testSuite instances.
- It runs a collection of test cases.
- The run method is invoked by the framework

Test Suite annotation


```
@RunWith(Suite.class)
//Add multiple test implementations here to run
@SuiteClasses({com.data.test.TestBackupFirst.class,com.data.test.UserTest.class,c
    om.data.test.TestBackupSecond.class})

public class RunAllTests {

public static Test suite() {
 TestSuite suite = new TestSuite("Test for com.data.test AllTests");
 return suite;
}
```

Test Runner class

- The test runner classes are provided in framework to invoke the test methods via run method in any Test implementations.(i.e. Test,TestCase or TestSuite).
- Two test runners are supported
- The junit.textui.TestRunner to run and produce only the test result.
- The junit.swingui.TestRunner class to run tests in GUI environmnet and produce graphical report of test executions.

Test Configuration

- To specify the test timeout specify with the annotation as

```
@Test(timeout=1000)
public void runLongTest() {….}
```

- To expect an exception in the test method

```
@Test(expected=ArrayIndexOutOfBoundsException.class)
public void testBounds() {
new ArrayList<Object>().get(1);
}
```

- If the method doesn't throw an exception of this type or if it throws a different exception than the one declared, the test is treated as failure.

Ignore the tests

- To temporarily disable a test or a group of tests use ignore annotation.
- Methods annotated with Test and @Ignore will not be executed as tests.
- A class containing test methods can also be annotate d with @Ignore and none of the containing tests will be executed.
- @Ignore @Test public void notYetRun() { ...}
- @Ignore public class TryNotMe {..}

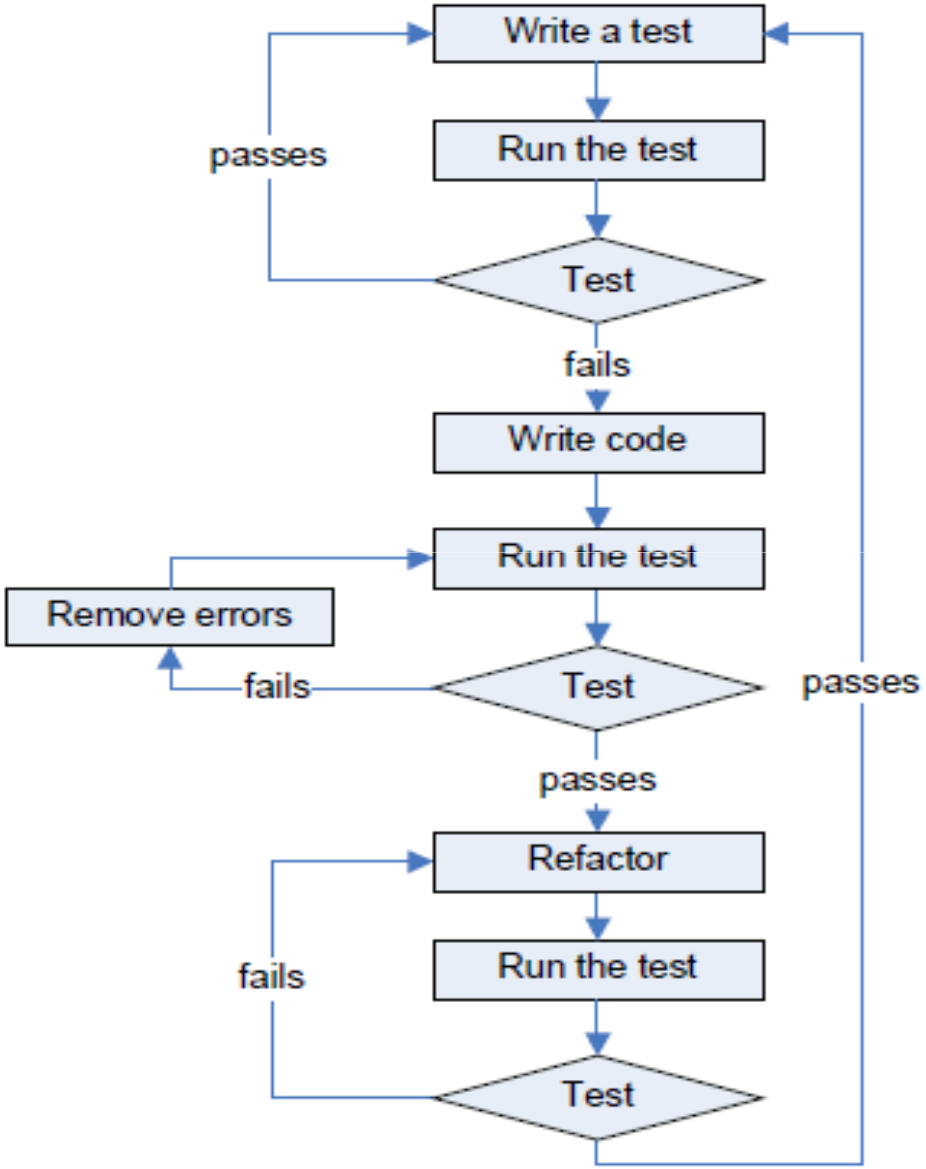
Explicitly fail the test

- Static method : class org.junit.Assert.fail(): Fails a test with no message.
- Static method : class org.junit.Assert.fail(String errorMessage): Fails a test with given message.

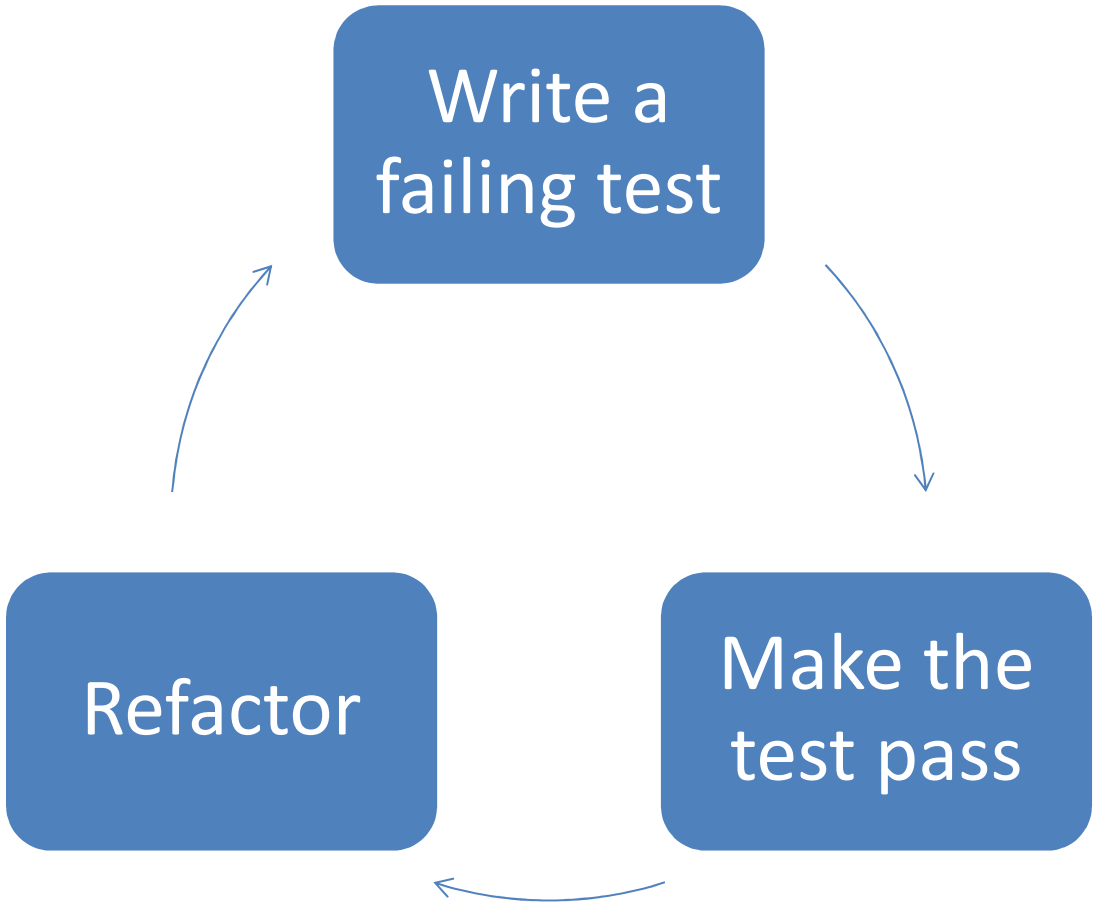
Golden Rule of TDD

Never write new functionality without a failing test.

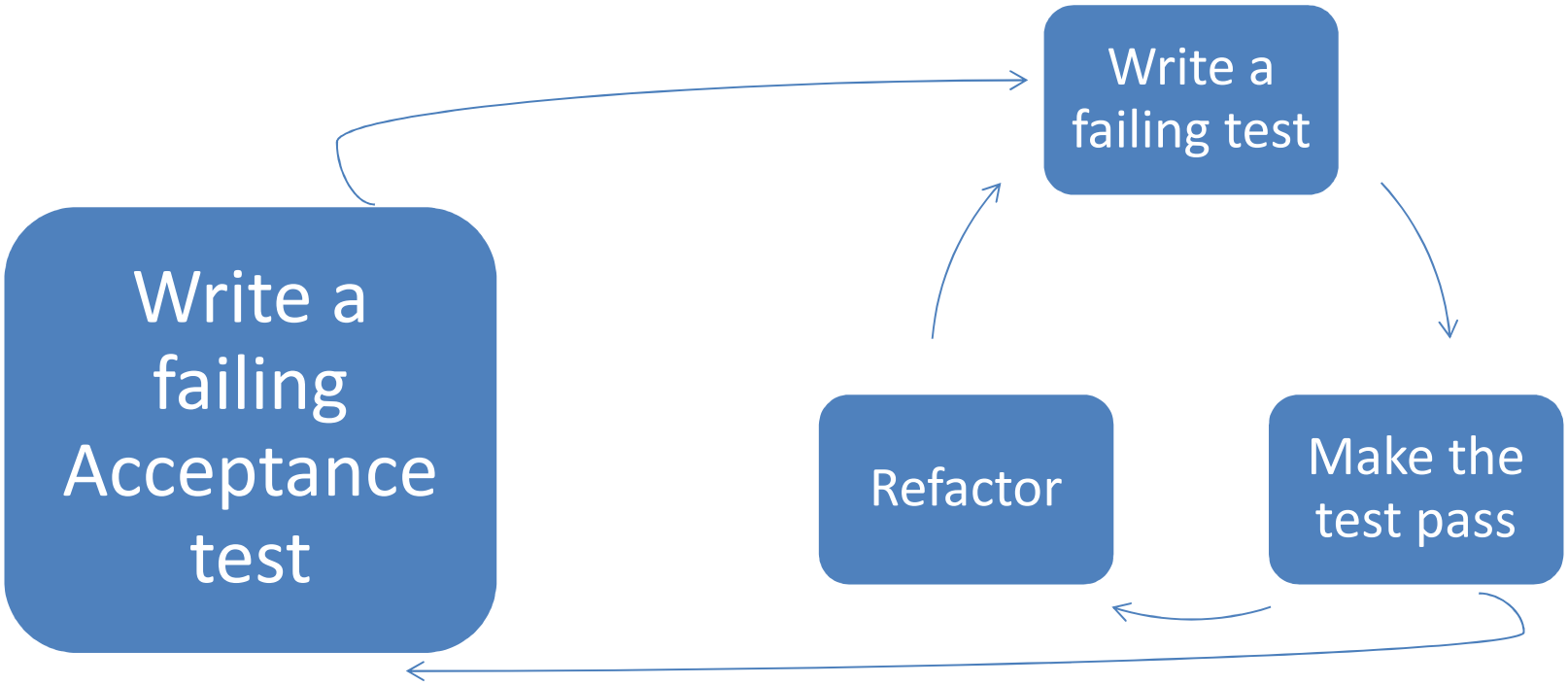
TDD cycle


Write a test
Run the test
passes
Test
fails
Write code
Run the test
Remove errors
fails
Test
passes
passes
Refactor
Run the test
fails
Test

Fundamental TDD Cycle-Red, Green Refactor



Inner and Outer feedback loops in TDD



User Story for Saving Account

As a user I should be able to

- Deposit amount in SavingAccount
- Verify the amount with balance
- Withdraw amount from the SavingAccount
- Verify the amount with balance

To Do

SavingAccount – deposit amount and verify the balance

Getting Started..

Create new Java project ‘BankApp’ with Eclipse Photon →Project Wizard support

Simple Design Principles

- Any code, including unit test code, should adhere to Four Principles of Simple Design.
  - Runs all tests
  - Contains no duplication
  - Expresses intent of programmers
  - Minimizes number of classes and methods

Test Case and Test class naming conventions

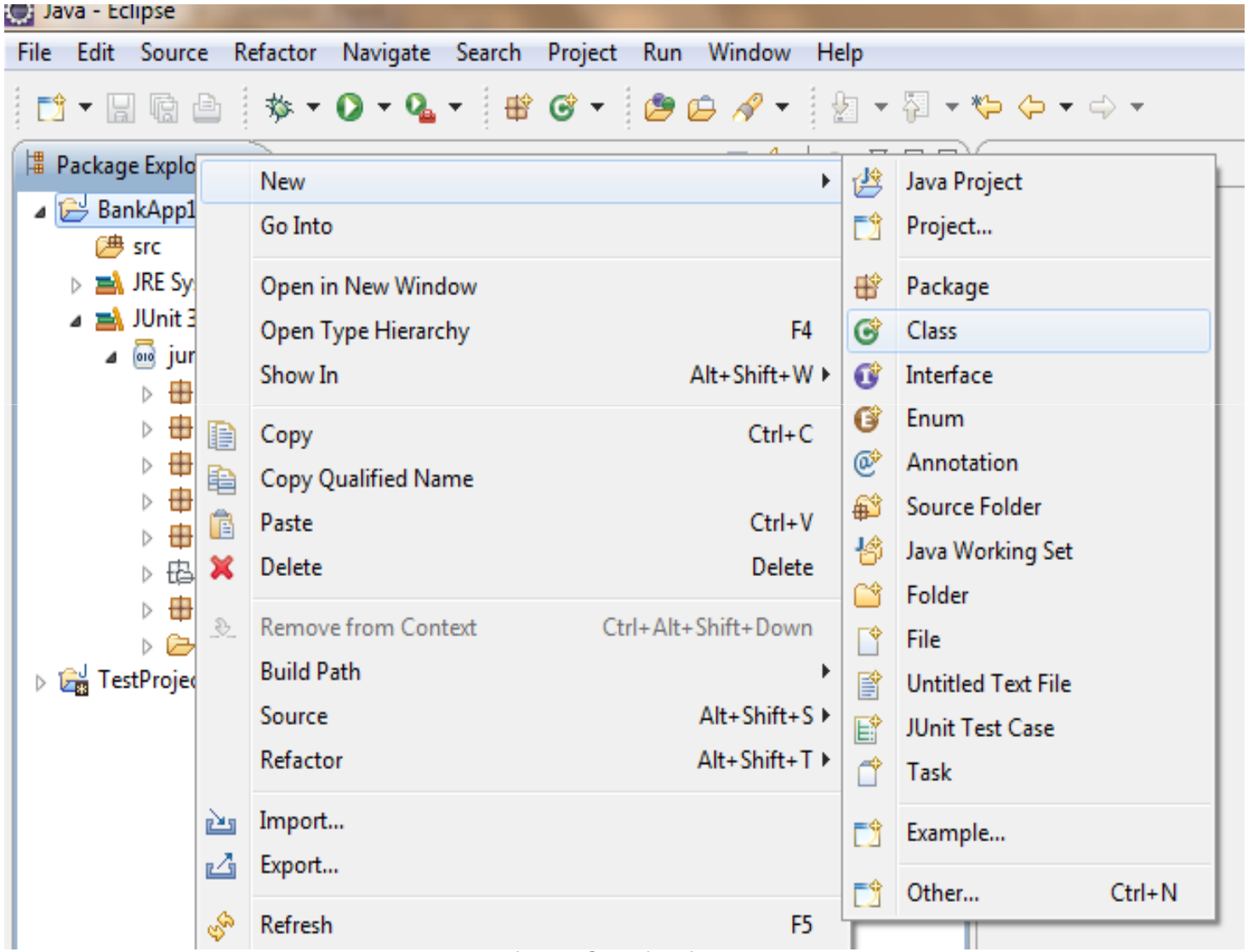
- The naming should follow standard naming conventions and guidelines.
- The name should indicate the purpose of testing.
- Unit Tests Should Be Maintainable and Readable.

Project Reference –JUnit library

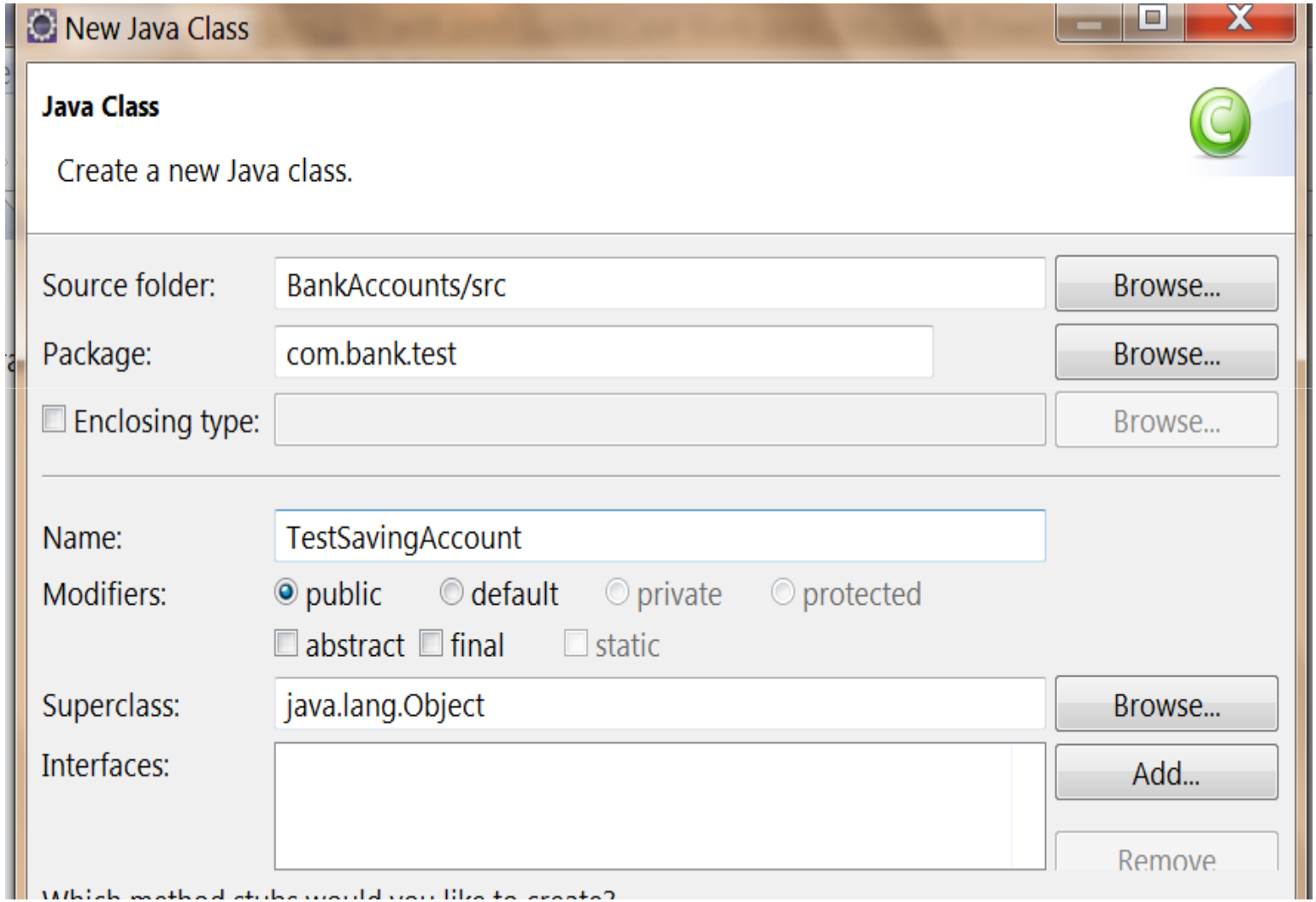
Add to the project the reference of the JUnit ver5.0 library with the project class-path settings.

Project Explorer view ,Right click on the project and select
BuildPath ->Configiure Build Path

Add a new Java Test Class



Java Test class ‘TestSavingAccount’

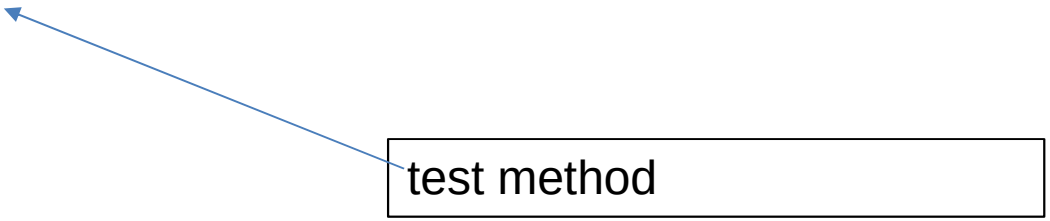


The Test case class : The Test Fixture


```
package com.server.test;


public class TestSavingAccount  {

}
```

Add the first Test method to Test class


```
package com.server.test;

import org.junit.Test;

public class TestSavingAccount {

@Test
public void checkSavingAccountDeposit()
{

}

}
```



Add the test code


```
package com.server.test;

import junit.framework.Assert;
import org.junit.Test;
public class TestSavingAccount {
 @Test
public void checkSavingAccountDeposit()
{
//Compiler errors..fix it
 SavingAccount obj= new SavingAccount();
 obj.deposit(1000);
 int amount = obj.balance;
 Assert.assertEquals(1000, amount);
}
}
```

Verify that the obj balance is equal to amount

Define the domain class SavingAccount


```
package com.server.bank;

public class SavingAccount
{

}
```

Add deposit method in SavingAccount

Write the minimum required code …

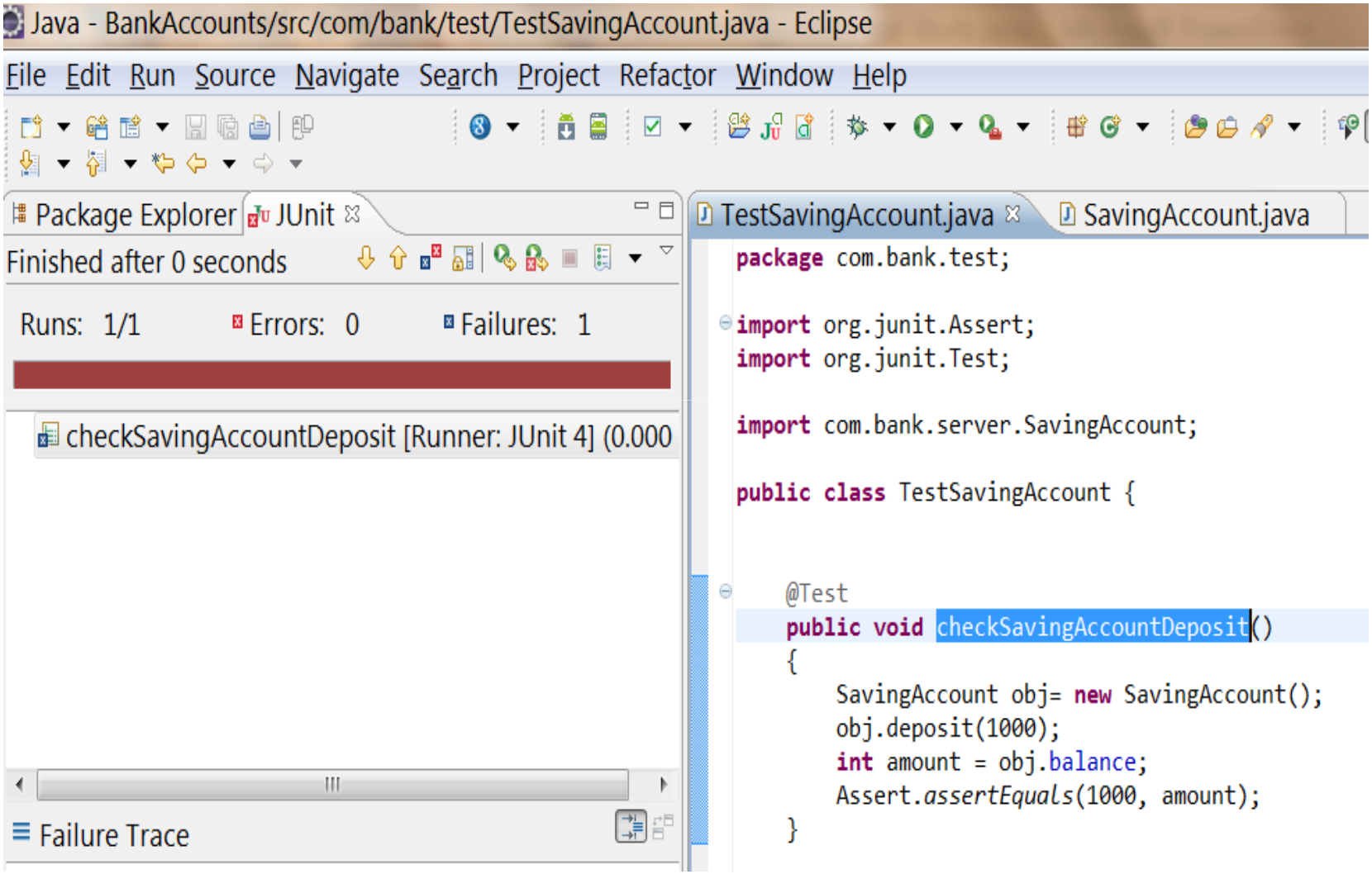
```
package com.server.bank;

public class SavingAccount {

public void deposit(int amount) {

  }
}
```

Add the balance variable in SavingAccount


```
package com.server.bank;

public class SavingAccount {

public int balance  ;

public void deposit(int amount) {

}
```

Run the Test

Congrats! Your First Test FAILED!



Test Failure description


```
java.lang.AssertionError: expected:<1000> but was:<0>
```

The first test as failure and NOT Success!

- The failure is most easy to implement!
  - Do Nothing!!
- We know the cause of failure.
- Incase if the test fails in a way not as we expected, something is wrong..fix it and correct it..helps to correct the errors easily..
- Helps to isolate the problems at actual failures .

An Unstable System

The Symptom of an Unstable System is that there is no single obvious place to look at when there is failure in the system.

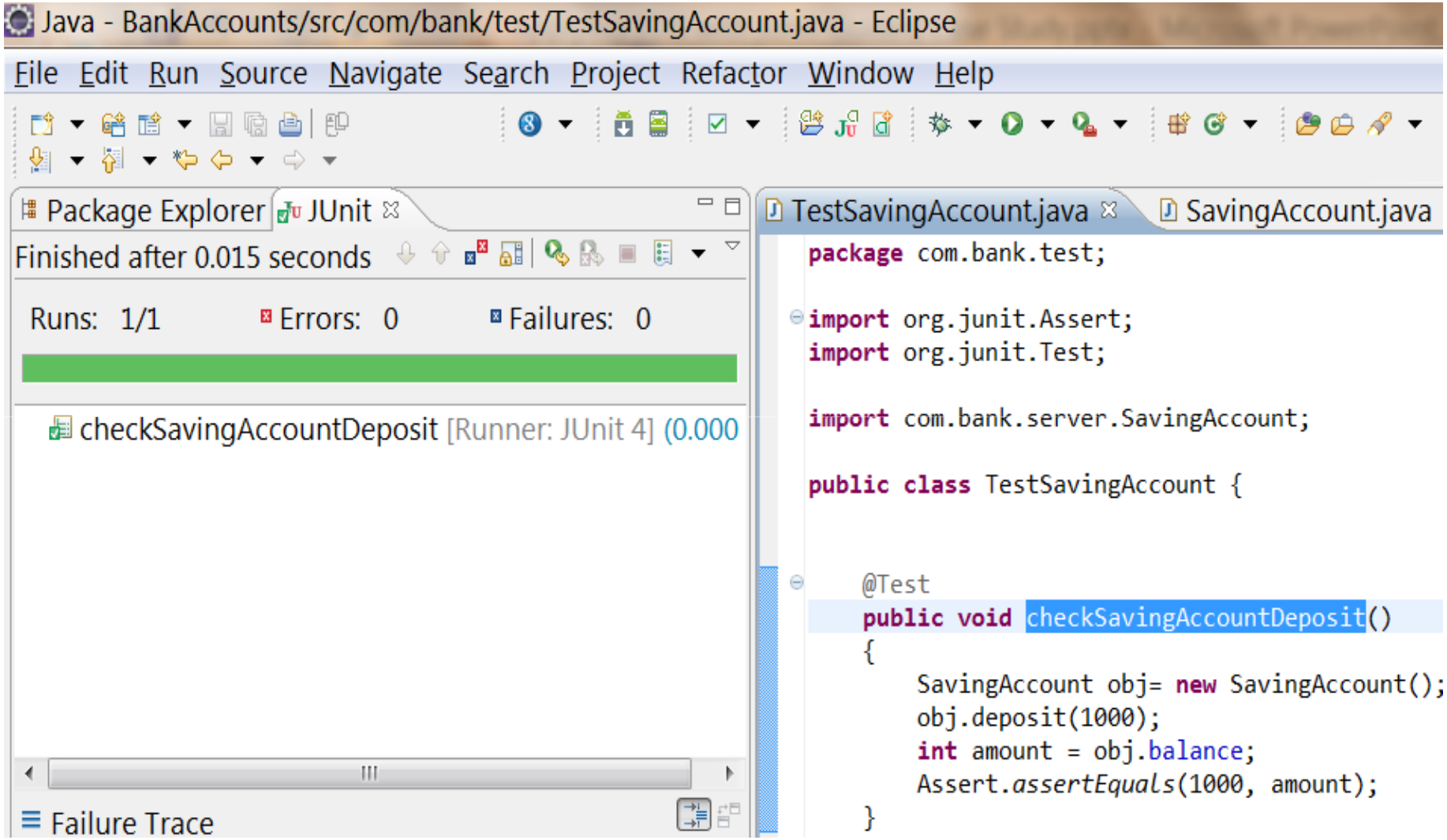
To make the Test succeed..

- In SavingAccount class initialize the balance variable value to 1000 .
- Run the Test..

Modified SavingAccount

```
package com.bank.server;

public class SavingAccount {

public int balance =1000;

public void deposit(int i) {

  }
}
```

Your First GREEN Bar of SUCCESS!



Although the result was Faked..

How Big the Test should be ?

1. Error localization: work involved to locate errors is large in case of large tests, in case of unit tests it is smaller.
2. Execution Time: More for larger tests
3. Coverage: Hard to determine coverage when new code is added for larges tests.
4. **The test should be ideally small and should test only one feature of the unit under test**.

Qualities of good unit test

- The test must fail if the code is broken and only if the code is broken.
- It runs fast.
- It helps to localize problems quickly.
- It tests only one feature.
- It is NOT dependent on other test cases.
- Any external dependencies needed are self contained in side the test.
- A unit test that takes less than $1/10^{th}$ of a second to run.

Good Unit Test ..

- Every Unit test should run quickly.
- To maximize benefits, tests should be run as frequently as possible.
- If unit tests are slow, they should be re-factored further to smaller unit tests.
- The test case should be time-independent.

Failure/Error reporting in test case

- The test case should have proper exception reporting and cleaning logic.
- The test should NOT have exception handling within a test case or if handled inside the test case it should indicate to test environment about the exceptions.

Test Case setup

- Do not use the test-case class constructor to set up a test case related parameters since the test class constructor is called for every new test case initialization.
- Instead prepare the setup before every test case execution.
- Externalize test input data wherever possible

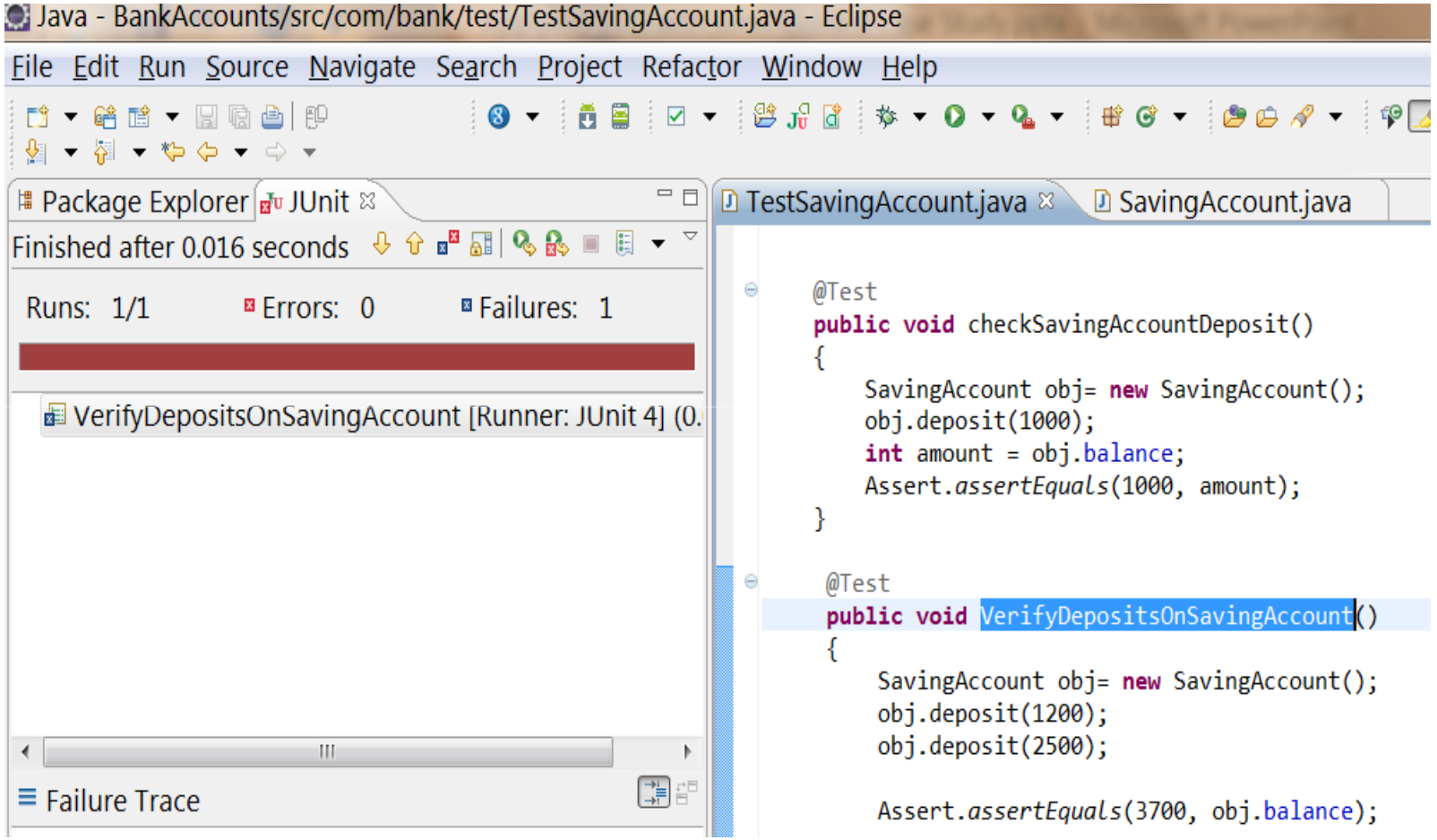
Test for the amount deposited..

- How to verify the amount that is deposited number of times in SavingAccount ?
- Add another test function ‘VerifyDepositsOnSavingAccount’ in Test class to verify the amount deposited.. .
- Inside the test function create the SavingAccount object , call the deposit function two times with different amount values and verify the balance on the SavingAccount object with assertion.

The test


```
@Test
public void VerifyDepositsOnSavingAccount()
{
	SavingAccount obj = new SavingAccount();
	obj.deposit(1000);
	obj.deposit(2400);

	//verify the final balance   value
	Assert.assertEquals(3400, obj.balance);
}
```

Run the test..



junit.framework.AssertionFailedError: expected:<3700> but was:<1000>

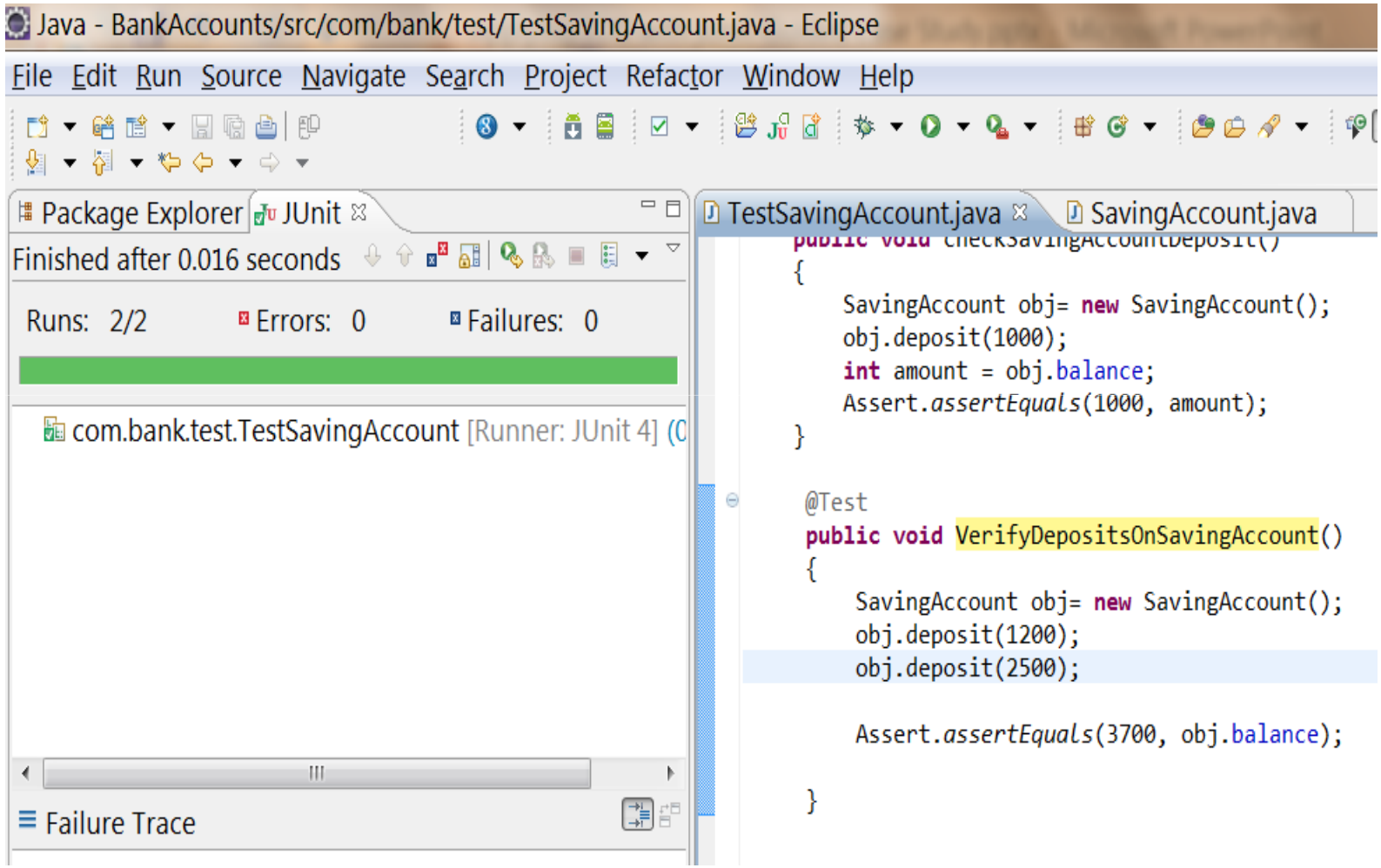
Why the test failed ?

- We have earlier faked the balance value with fixed value..
- Now we will have modify the SavingAccount with minimum code required to update the balance added by the deposit method.

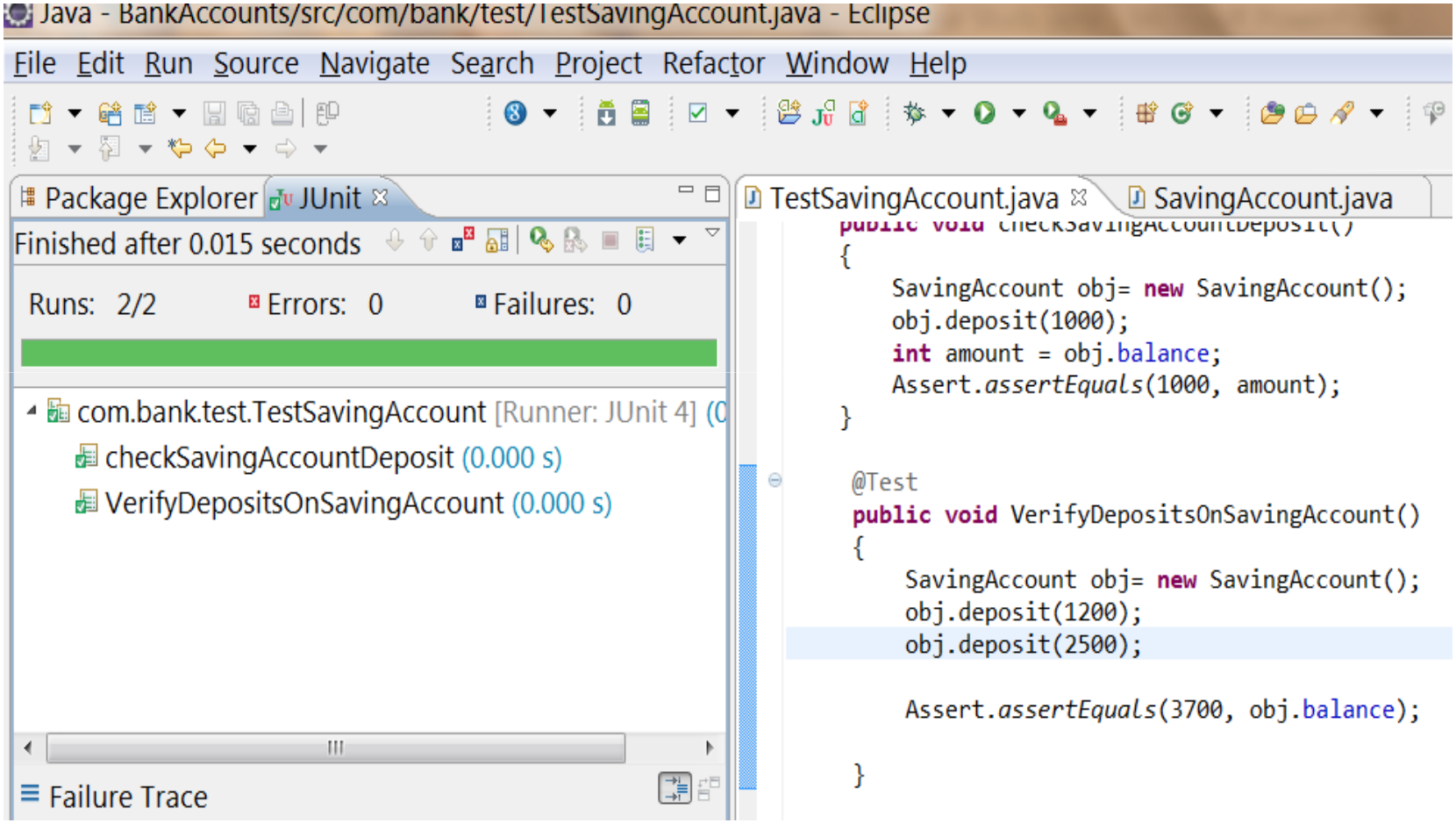
SavingAccount Deposit


```
public class SavingAccount {

public int balance;

public void deposit(int amount) {

balance+= amount;

 }
```

Run the Test


Java - BankAccounts/src/com/bank/test/TestSavingAccount.java - Eclipse
File Edit Run Source Navigate Search Project Refactor Window Help
Package Explorer
JUnit
Finished after 0.016 seconds
Runs: 2/2
Errors: 0
Failures: 0
com.bank.test.TestSavingAccount [Runner: JUnit 4]
Failure Trace
TestSavingAccount.java
SavingAccount.java
{
SavingAccount obj= new SavingAccount();
obj.deposit(1000);
int amount = obj.balance;
Assert.assertEquals(1000, amount);
}
@Test
public void VerifyDepositsOnSavingAccount()
{
SavingAccount obj= new SavingAccount();
obj.deposit(1200);
obj.deposit(2500);
Assert.assertEquals(3700, obj.balance);
}

All other tests ?



All the tests are successful!!.

What is done ?

- Defined balance variable in SavingAccount class and exposed it to the user.
- Update the balance of SavingAccount by deposit method.

Is it a good Design ?

By exposing the balance of SavingAccount we are increasing the possibility of modifying the balance of SavingAccount without depositing the amount to it! !

Correct it..

- Limit the access of balance in SavingAccount to outside world by making it private.
- But to share the balance add a public method **getBalance** in SavingAccount class.
- In the test class instead of using the balance directly change it to call getBalance on SavingAccount object.

SavingAccount Revised


```
public class SavingAccount {

    private int balance;

    public void deposit(int amount) { }
    public int  getBalance()
     {
         return balance;
     }
```

Passing the test-Going Green

- Fake it
- Triangulation
- Obvious Implementation

Fake the results..

- Sometimes during development, in absence of actual code/objects/files the output results can be faked/hard coded with fixed values to return dummy data just to make the test success.
- This test has to be replaced later with real objects.
- It is Not recommended to use fake objects in production code.

Fake it (Until you implement it)

- Benefits:

2. it has psychological benefits for some, which can aid in stress reduction through taking small steps, receiving positive feedback, and providing momentum.
3. it facilitates a “safety net” which can be used to provide rapid feedback if you go off course during a refactoring effort.

Triangulation

- If I get stuck and I don’t know how a complex algorithm should work I’ll write a test for an error case. Then I’ll write a test for the simplest non-error case I can think of and return a hard coded value. Then I’ll write another test case and see if I can figure out the algorithm at that point. In doing so I gain some momentum and perhaps some insight in how the algorithm should behave on an edge case and a few normal cases.

Triangulation

- This is called triangulation and it was used in celestial navigation for thousands of years. It is easier to see you are moving when you compare your position to two or more points on the horizon rather than just one. The same applies to coding; it is often easier to figure out the behaviour of an algorithm by examining a couple of test cases instead of just one.

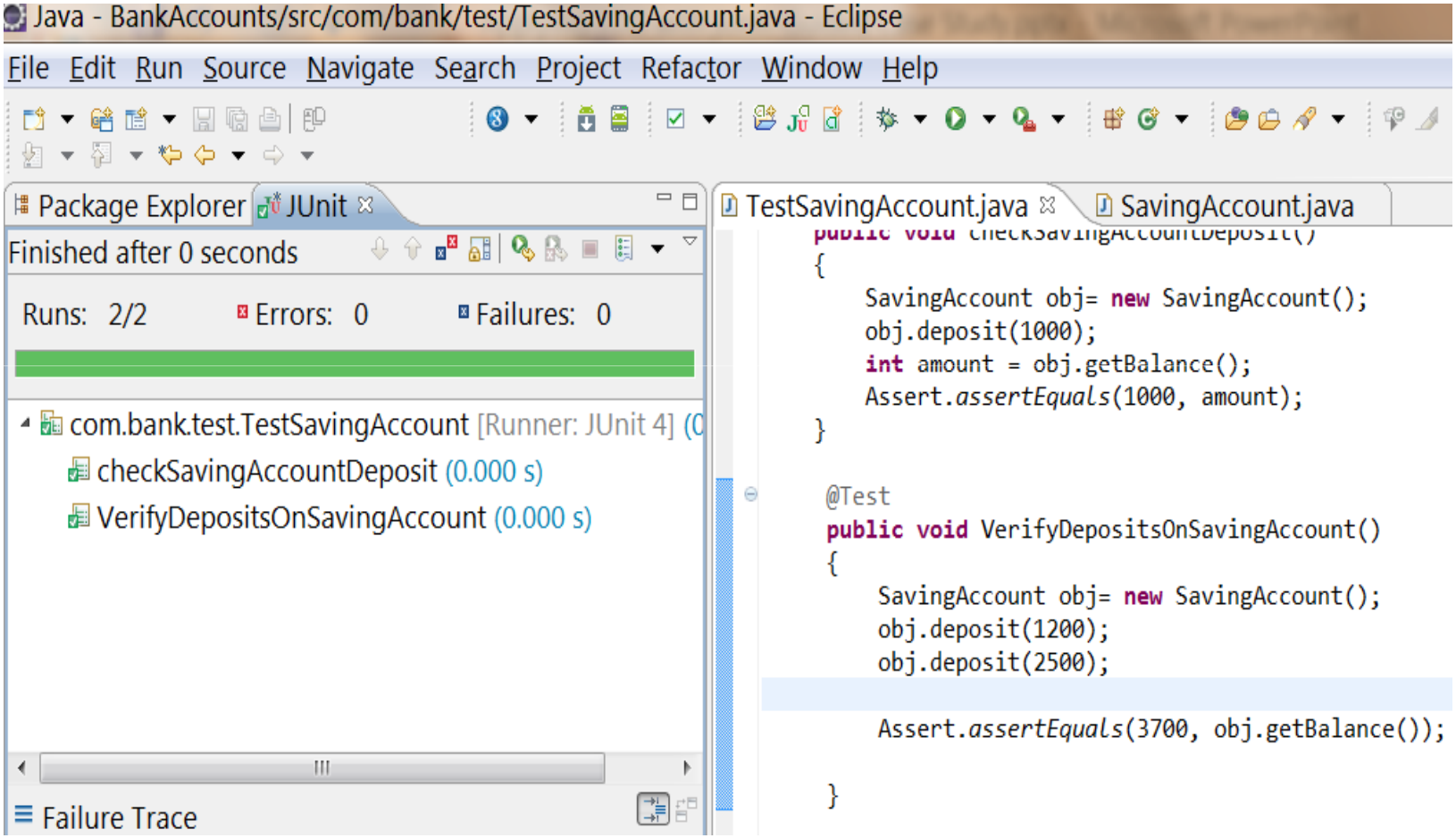
Obvious Implementation

- Write the actual implementation for the behavior expected by the test case while exercising the system.

Call getBalance in Test Class


```
@Test
public void VerifySavingAccountBalance() {
        SavingAccount obj = new SavingAccount();
         obj.deposit(1200);
         Assert.assertEquals(1200, obj.getBalance());
}
@Test
public void VerifySavingAccountMultipleDeposits(){
        SavingAccount obj = new SavingAccount();
        obj.deposit(1000);
        obj.deposit(2400);
        Assert.assertEquals(3400, obj.getBalance());//verify the balance
}
```

Run all the tests



All the tests have succeeded..

Technical Foundations to grow a system

- To grow system reliably by coping with unanticipated changes; we need

1. **constant testing** to catch regression errors so we can add new features without breaking existing ones. To perform constant tests, we need to automate testing.
2. **constant refactoring** to keep code as simple as possible so it is easier to understand and modify.

What if

- The SavingAccount implementation described here is for Indian banking environment where Rupee is the default currency..
- What if an NRI person wants to remit the amount in US Dollars to the corresponding Indian SavingAccount ?
- How this US Dollars amount will be accommodated here ?
- Obviously the US Dollars amount need to be first converted to Indian Rupees and then transacted…

The deposit Dollar test method


```
@Test
   public void testSavingAccountDollarTransfer()
   {
      obj = new SavingAccount();
      obj.deposit(1000);
      obj.deposit(100);

     //Compiler Errors Fix it

      Dollar usCurrency = new Dollar(100);
      obj.deposit(usCurrency);
      double amount = Dollar.getRupeesConvresionRate() *
    usCurrency.getCurrencyValue();

      //verify the current  balance
      Assert.assertEquals( 1000+amount, obj.getBalance());
}
```

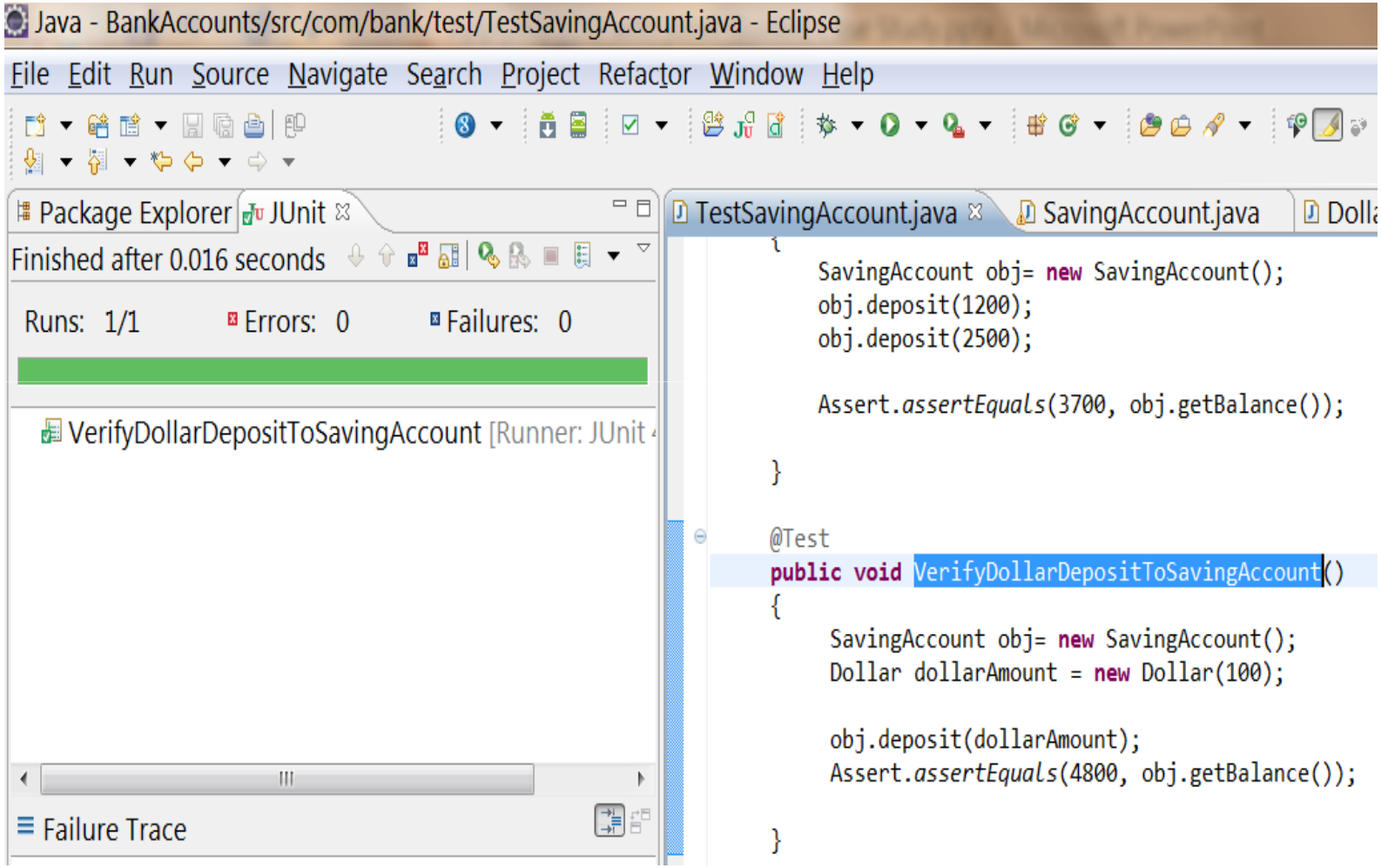
Overloaded Deposit methods


```
public void deposit(int amount) {

balance+=amount;
}
public void deposit(Dollar dollarAmount) {
  int dollarValue = dollarAmount.getCurrencyValue();
   int rupeeRate =
    dollarAmount.getRupeesConvresionRate();
   int amount = dollarValue * rupeeRate;
   balance+=amount;
}
```

The Dollar Class

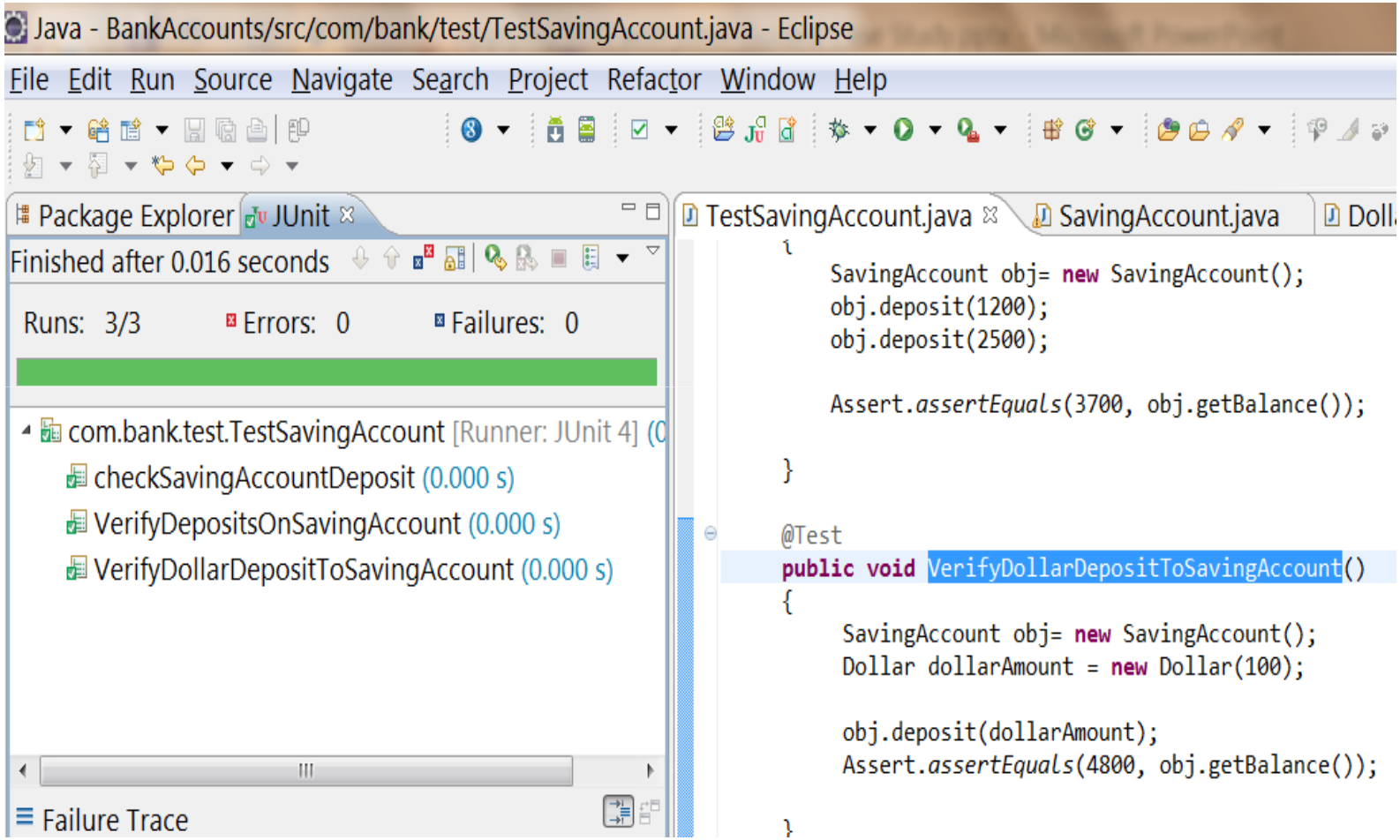

```
package com.bank.server;

public class Dollar {
private int currencyValue;

public Dollar(int value){
    this.currencyValue = value;
}
int getCurrencyValue() {
   return currencyValue;
}
  private static int rupeesConvresionRate = 48;
  static int getRupeesConvresionRate() {
      return rupeesConvresionRate;
}
```

Run test



The taste of success!

Run all the Tests



One more Currency..

What if the user wants to transfer the Euro currency to this SavingAccount ?

add a test method


```
@Test
public void VerifyEuroDepositToSavingAccount()
{
  SavingAccount obj= new SavingAccount();
 Euro euroAmount = new Euro(60);

 obj.deposit(euroAmount);
 Assert.assertEquals(1320, obj.getBalance());

}
```

Evolve the Euro class

- Define the Euro class to be used as a ValueObject to make the transactions.
- The attributes required for Dollar
  - currencyValue
  - rupeesConvresionRate for converting to Indian Rupees.

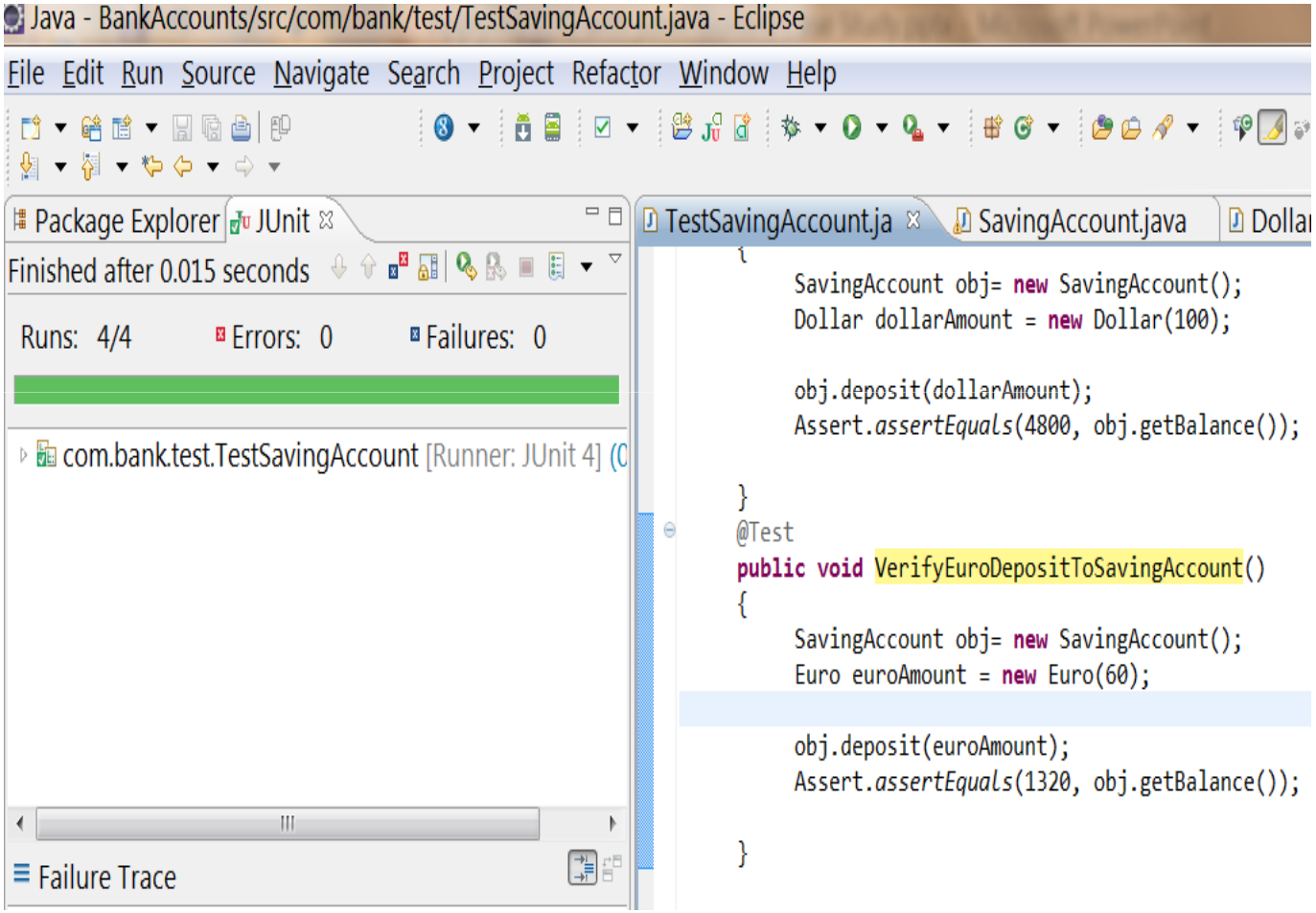
Euro class


```
package com.bank.server;

public class Euro {
 private int currencyValue;

public Euro(int value){
    this.currencyValue = value;
}
 int getCurrencyValue() {
   return currencyValue;
}
   private static int rupeesConvresionRate = 22;
 static int getRupeesConvresionRate() {
   return rupeesConvresionRate;
}
```

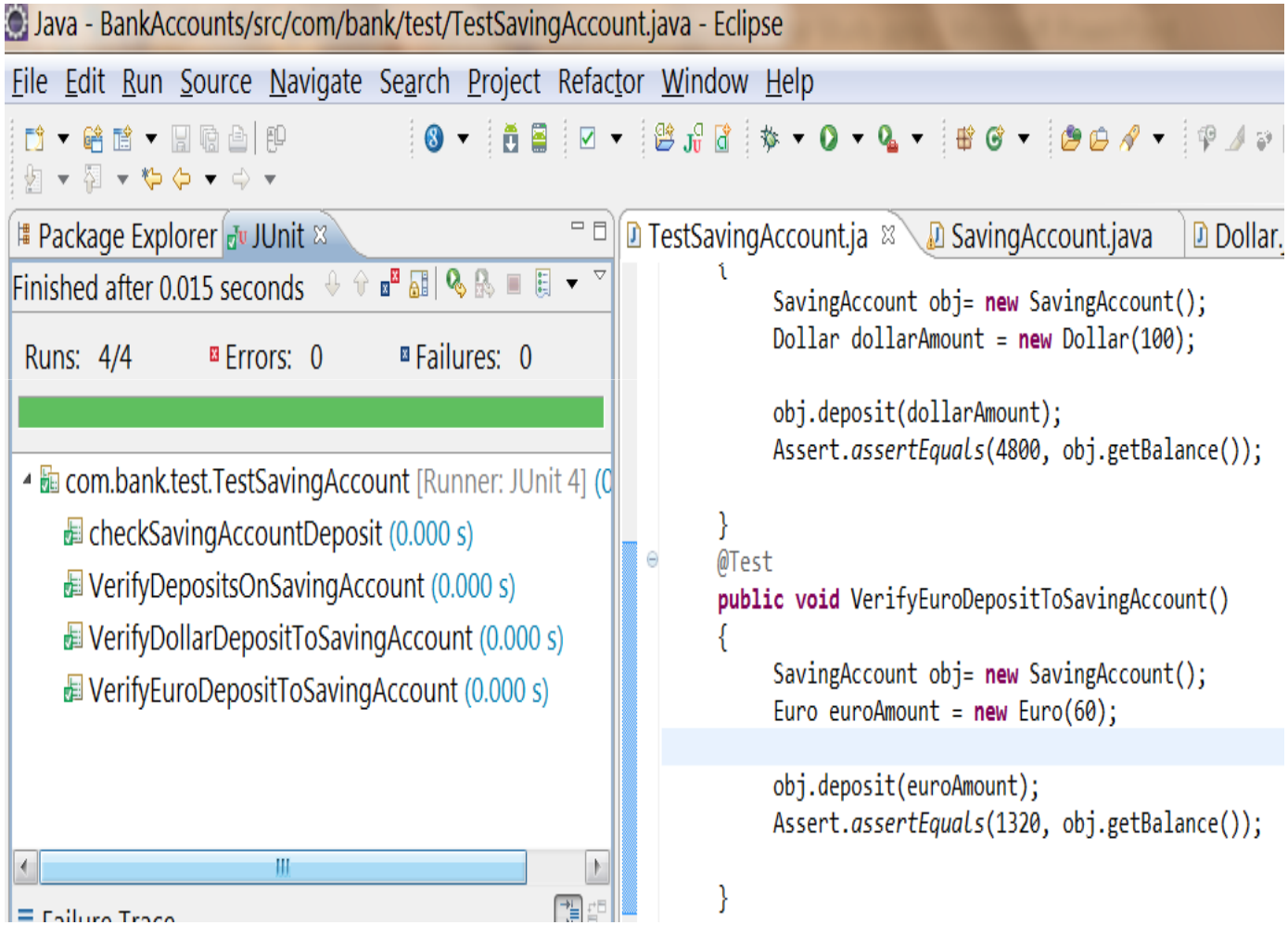
Overloaded deposits


```
 public void deposit(Dollar dollarAmount) {
int dollarValue = dollarAmount.getCurrencyValue();
int rupeeRate = dollarAmount.getRupeesConvresionRate();
int amount = dollarValue * rupeeRate;
balance+=amount;
}
public void deposit(Euro euroAmount) {
int euroValue = euroAmount.getCurrencyValue();
int euroRate = euroAmount.getRupeesConvresionRate();
int amount = euroValue * euroRate;
balance+=amount;
}
```

Run the Test


Java - BankAccounts/src/com/bank/test/TestSavingAccount.java - Eclipse
File Edit Run Source Navigate Search Project Refactor Window Help
Package Explorer
JUnit
Finished after 0.015 seconds
Runs: 4/4
Errors: 0
Failures: 0
com.bank.test.TestSavingAccount [Runner: JUnit 4]
Failure Trace
TestSavingAccount.ja
SavingAccount.java
SavingAccount obj= new SavingAccount();
Dollar dollarAmount = new Dollar(100);
obj.deposit(dollarAmount);
Assert.assertEquals(4800, obj.getBalance());
@Test
public void VerifyEuroDepositToSavingAccount()
SavingAccount obj= new SavingAccount();
Euro euroAmount = new Euro(60);
obj.deposit(euroAmount);
Assert.assertEquals(1320, obj.getBalance());

Run All the Tests



Is TDD process slower..?

- In TDD we take tiny/small steps..during initial stages of development.
- These are also called as ***baby*** steps.
- Multiple steps may also present similar/common behavior. In that case these can be combined as common steps and reused.
- When tested with common behavior for one module, for another module, same code can be reused.

Baby Steps in TDD

- How small or large the baby step can be ..?
- When you are NOT sure about the outcome of testing of complex code implementation ,go by small steps.
- When you are sure about the obvious outcome of test results and expect known behavior, you may take large jump also.

Parameterized test case

- Every JUnit test case method is without arguments and return void and public.
- All parameters setup either to be done in before method or inside the test method.
- Junit5 supports parameterized test cases with @ParameterizedTest annotations(as parameterized test runner) and @ValueSource for parameter source.

Test data from external file

- The Junjt5 supports CSV parameters with @CsvSource annotations.
- The @CsvFileSource supports reading CSV parameters from external files.
- This becomes *Data Driven Testing*!

How to access private data

- Using reflection api access and set the private data of the object under test.
- Field, Method,Constructor classes along with Class support access to class data and methods.
- It also supports setting the private data!

Code coverage measurement

- The testing should ensure maximum code coverage to ensure no dead code.
- The test should cover negative scenarios and borderline cases, in addition to positive scenarios of inputs and environment.
- Code coverage tools like JaCoCo included as plug-in in build tools like gradle,maven etc.
- Other tools Cobertura,Jcove, OpenClover, Serenity are also useful.

Testing with Mock Data

Just to make the test success..

- Sometimes during testing, the real test data may not be available.
- Just to make the test success, fake or dummy data or fake object is added to the application to make the test success.
- This fake data or fake objects should not be used in production.
- This fake data or fake objects should be removed at later stage when the real data or objects are available.

Fake Data or Fake objects

- Fakes are objects that have working implementations, but not same as production one.
- Usually they take some shortcut and have simplified version of production code.

Fake Usage

- This can be an in-memory implementation of Data Access Object or Repository.
- This fake implementation will not engage database, but will use a simple collection to store data.
- This allows to do integration test of services without starting up a database and performing time consuming requests.

Fake for Prototype

- The fake implementation are useful for prototyping and spikes.
- We can quickly implement and run our system with in-memory store, deferring decisions about database design.
- Another example can be also a fake payment system, that will always return successful payments.

Test Double

- Using these dummy data or objects reduces complexity of testing and allows to verify code independently from the rest of the system.
- A Test Double is a generic term used for these objects.
- Fakes are one type of test doubles.

The objects that are NOT Real

- The general term is **test double**. It includes **dummy**, **fake**, **stub and mock** objects**.**
- **Dummy** objects are passed around but never actually used; they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually take some shortcut or dummy values which makes them not suitable for production (an in memory database is a good example).

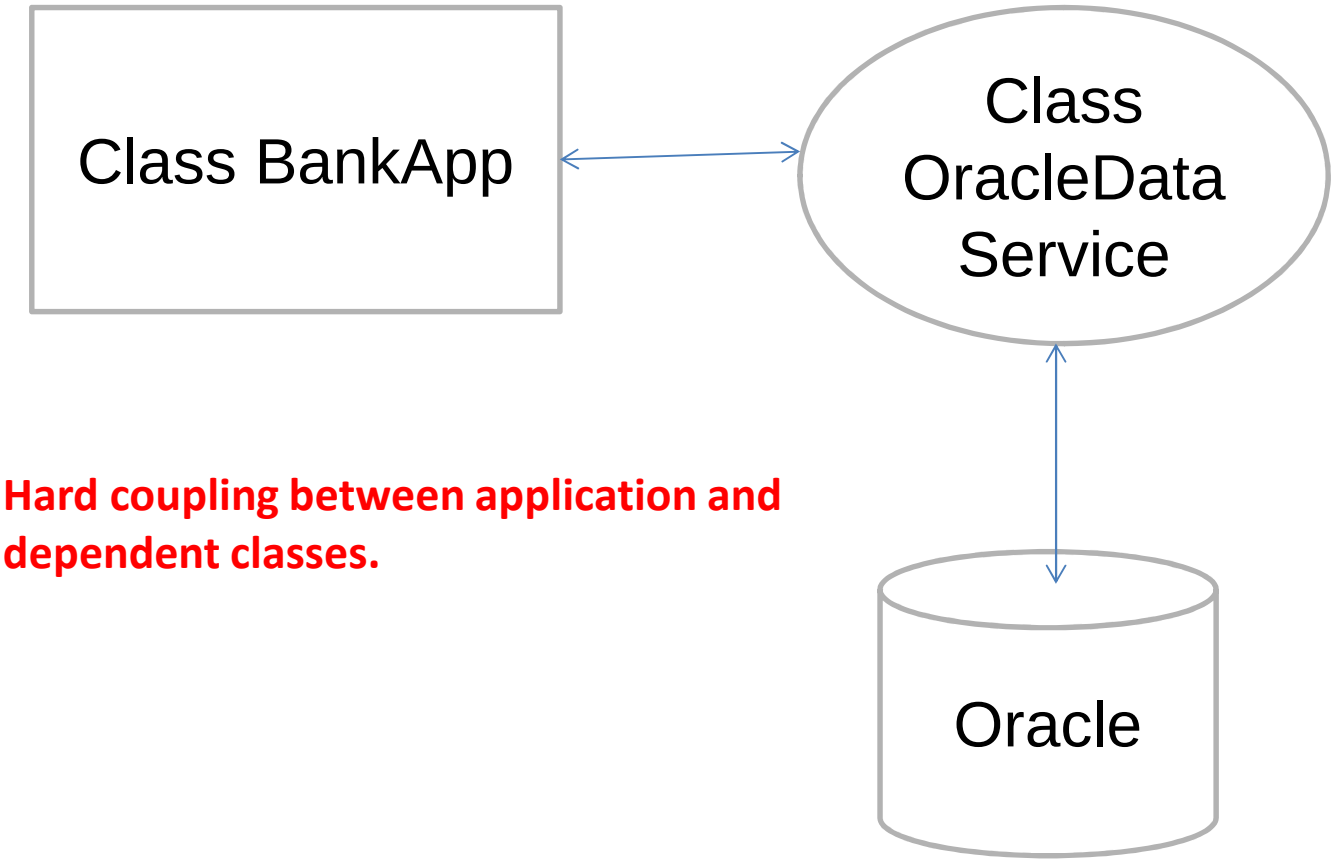
Test Isolation

- Every test should be isolated from other test cases.
- The test should not depend on other test cases execution.
- Every test cases should be self contained to have its own environment including test data and other dependent objects initialization and setup.

Dependent objects

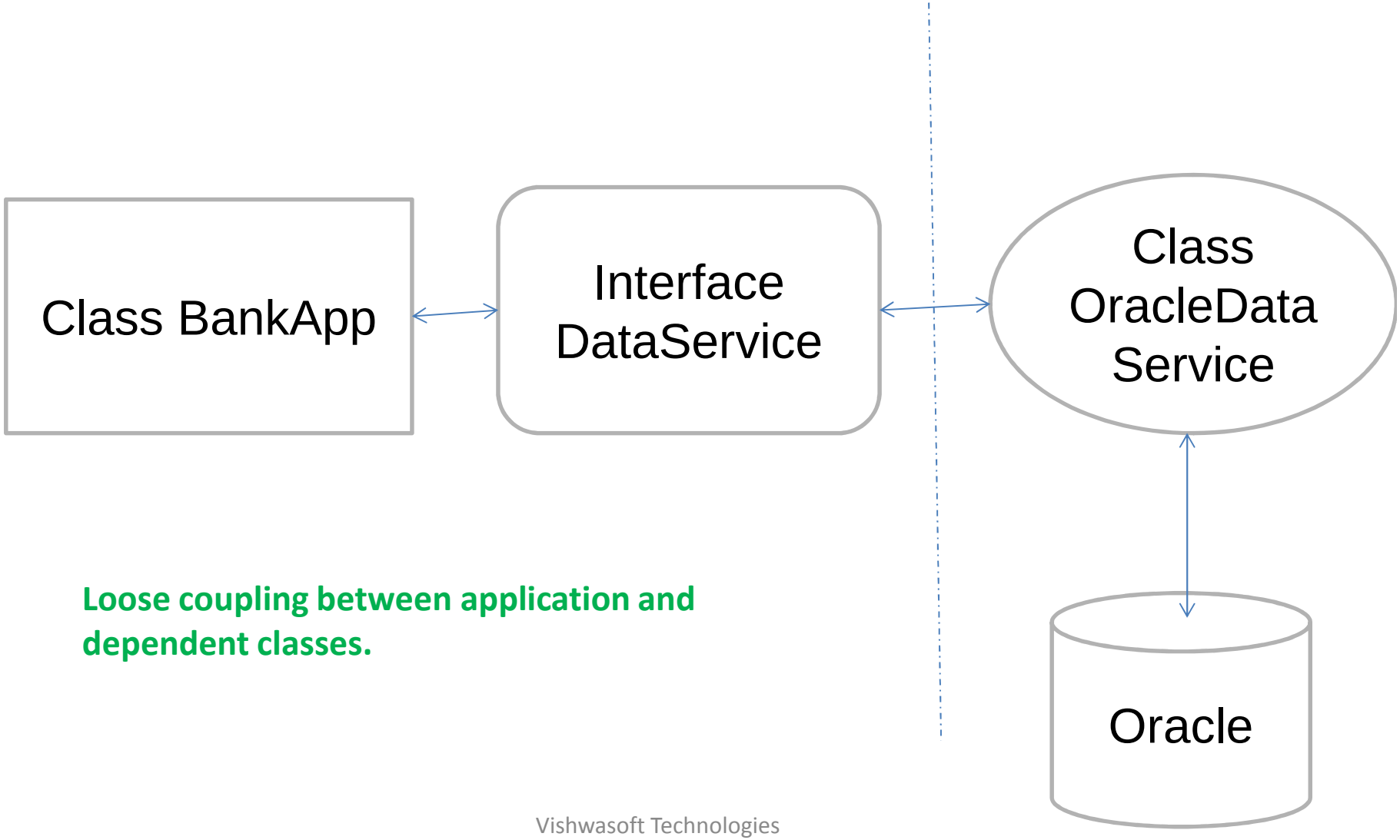
- When certain objects/classes depend on other objects/classes for certain methods invocation, how to test those depending parent classes in absence of dependent child classes.
- When development in team, each team is isolated and tasked with certain development of classes .
- How to test those individual items developed by independent teams.

Dependency Example



Hard coupling between application and dependent classes.

Break the Dependencies--Isolate



Loose coupling between application and dependent classes.

Dependencies NOT available

- How to test the unit when the required dependencies are not available.?
- When OracleDataService class is not available.
  - Add dummy implementation class with dummy test data.
  - The required backend data (i.e. Oracle Database)also needs to be created.
  - The real production data in backend may get corrupted because of test data.

Add a Mock DataService Class

- The mock object will represent the DataService interface and will not need the real backend database.
- The mock will provide only the required method.
- The mock has to be configured to return dummy data.
- The mock will allow to verify the Application class interaction with DataService.

When to use Mock?

- Isolate Interface implementation from application during testing and verify the application interaction with interface.
- Isolate abstract class from application for testing of abstract behavior and application interaction with abstract method.
- Isolate Concrete class from application for debugging and trouble shooting (*SPY* on the object methods) and verify the external interaction with application.

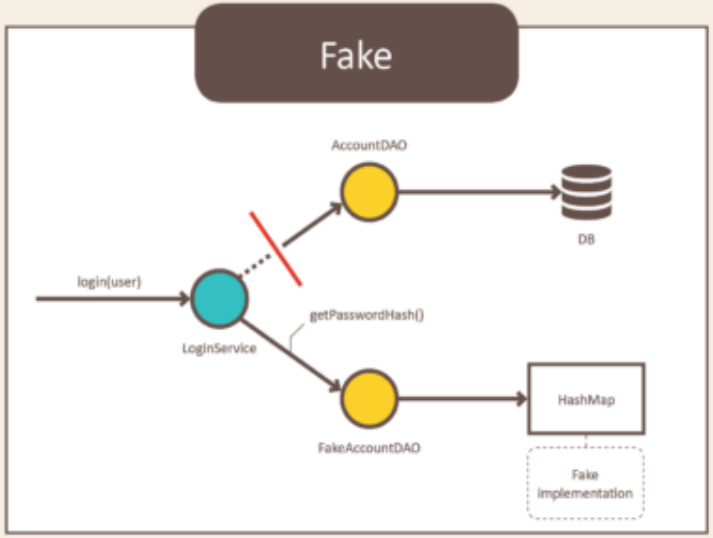
Create Mock in Java

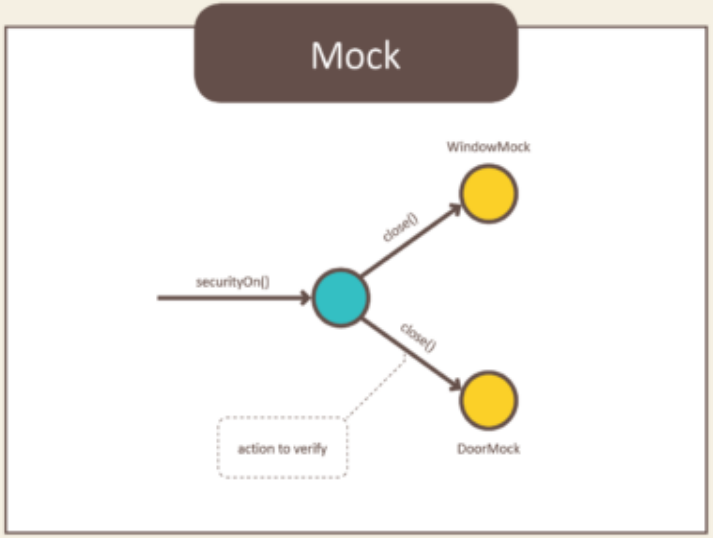
- Mocking frameworks
  - EasyMock
  - JMockit
  - JMock
  - Mockito

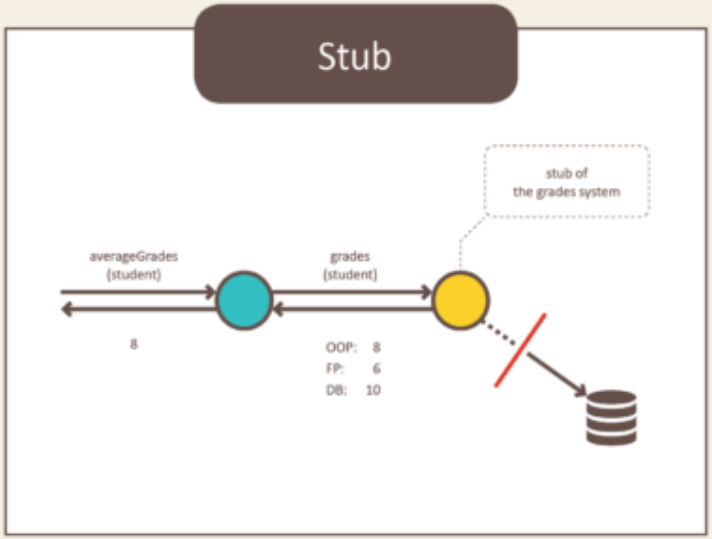
How Mock objects work..?

- The mock objects (classes) dynamically implement the interface at runtime with the choice of implementation of specific methods.
- The mock objects (classes) dynamically extend the concrete or abstract classes at runtime with the choice of override or implementation of specific methods.

Test Doubles







Mock Usage

- The mock object is created to present dynamic implementation behavior of interfaces/parent classes/helpers to those depending on it.
- The mock object is first configured to present some expected behavior/output values based on certain input data values.
- This configuration process is called as recording.
- The mock object is passed to the calling/dependent objects as real representation of absent object.

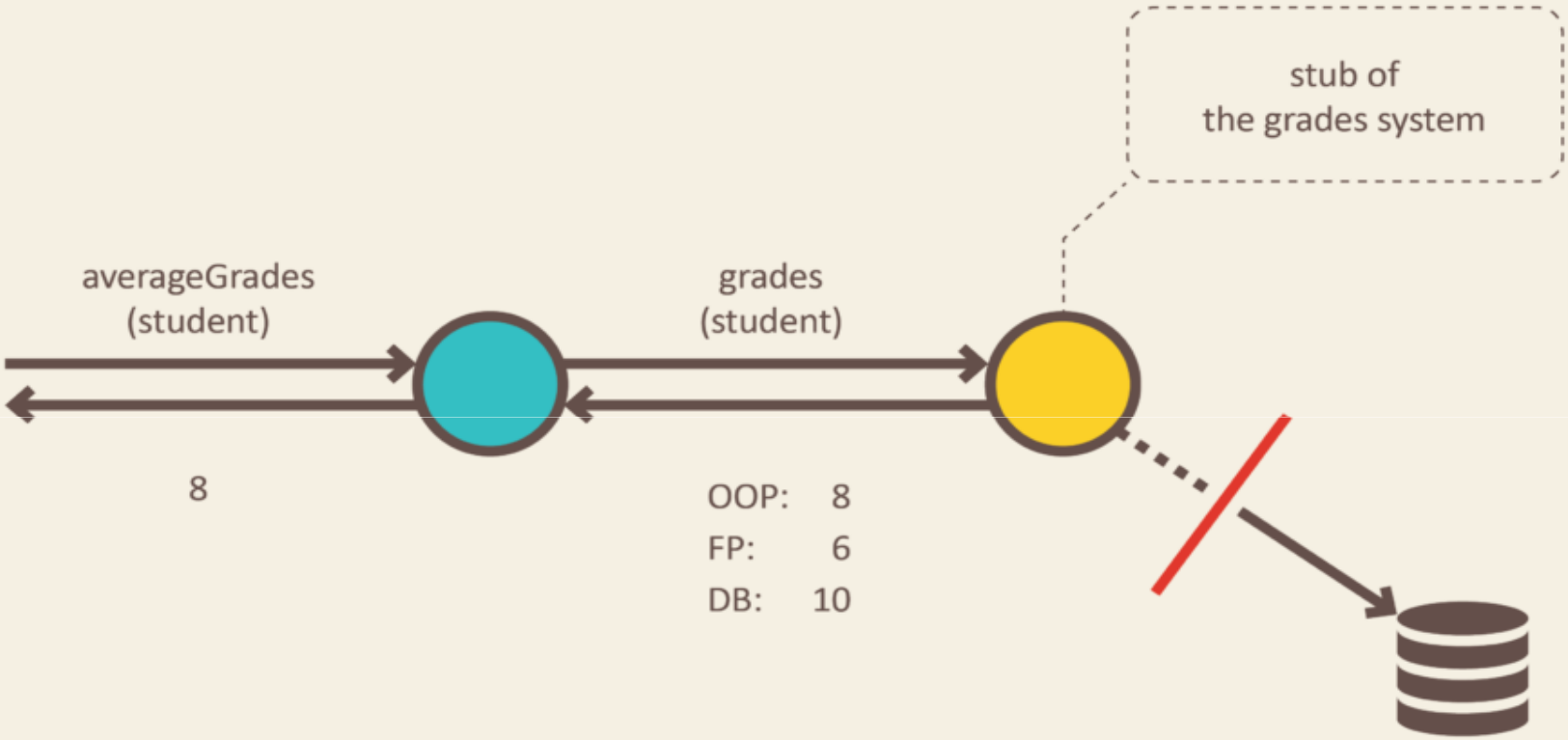
Mock as Stub Object

- Stub is an object that holds predefined data and uses it to answer calls during tests.
- It is used when we cannot or don’t want to involve objects that would answer with real data or have undesirable side effects.

Stub Usage

- An example can be an object that needs to grab some data from the database to respond to a method call.
- Instead of the real object, introduced a stub and define what data should be returned.

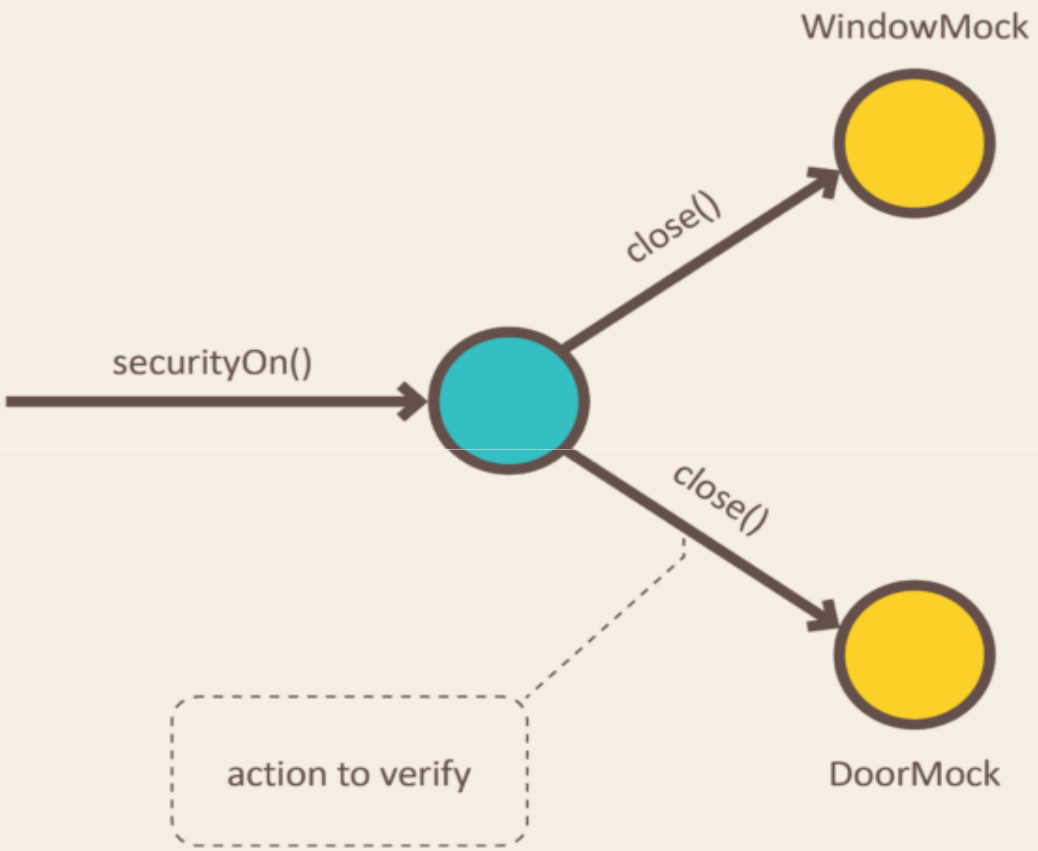
Stub


stub of
the grades system
averageGrades
(student)
grades
(student)
8
OOP: 8
FP: 6
DB: 10

Mock Usage

- **Mocks are objects that register calls they receive.**
  In test assertion we can verify on Mocks that all expected actions were performed.
- Use mocks when we don't want to invoke production code or when there is no easy way to verify, that intended code was executed.
- There is no return value and no easy way to check system state change.
- An example can be a functionality that calls e-mail sending service.
- We don't want to send e-mails each time we run a test. Moreover, it is not easy to verify in tests that a right email was send. Only thing we can do is to verify the outputs of the functionality that is exercised in our test. In other worlds, verify that e-mail sending service was called.

Mock


WindowMock
close()
securityOn()
close()
action to verify
DoorMock

Mock Simulation

- The mock object should be providing specific method and return specific data.
- The mock object can throw exceptions for testing with exceptions.
- The mock object method can do nothing in case the mocked method doesn't return data.
- The mock object method can process the passed parameters and update them.

Mockito Matchers

- Matchers are like regex or wildcards where instead of a specific input (and or output), you specify a range/type of input/output based on which stubs/spies can be rest and calls to stubs can be verified.
  - Argument Matchers during setup
  - Verification Matchers for verifying actual calls to stubs/mocks

ArgumentCaptor

- *The ArgumentCaptor* is used for making sure certain arguments were passed to mocks.

Dependency on container objects

- The classes/objects running in servlet web applications inside web containers.
- The database DAO classes in absence of service objects should be able to be tested outside those containers or in absence of those services.
- The environment which is able to simulate/mock/present the behavior expected by dependent objects is the **mock objects**.

The output of mock objects

- The mock objects present expected output/return values from methods when invoked from dependent objects.
- This process is called as mock playback.
- The output/return values from mock object invocation is compared with assertion and indicates success or failure.
- Later we can verify whether there was a actual call on mock objects and how many times it was called on the mock is also verified.
- This is assertion of mock behavior.

Stub vs.Mock

- **Stubs** provides canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.
- **Mocks** are what we are talking about here: objects pre-programmed with expectations which form a specification of the calls they are expected to receive.
- Mocks vs Stubs = Behavioral testing vs State testing

Testing with Stub object

- Test lifecycle with stubs:
- Setup - Prepare object that is being tested and its stubs collaborators.
- Exercise - Test the functionality.
- Verify state - Use asserts to check object's state.
- Teardown - Clean up resources.
- The stub methods return hard-coded values.

Test with Mock

- Test lifecycle with mocks:
- Setup data - Prepare object that is being tested.
- Setup expectations - Prepare expectations in mock that is being used by dependent object.
- Exercise - Test the functionality.
- Verify expectations - Verify that correct methods has been invoked in mock.
- Verify state and behavior- Use asserts to check object's state.
- Teardown - Clean up resources.

Mock vs. stub

- Mock is very similar to Stub but interaction-based rather than state-based.
- This means you don't expect from Mock to return some value, but to assume that specific order of method calls are made.

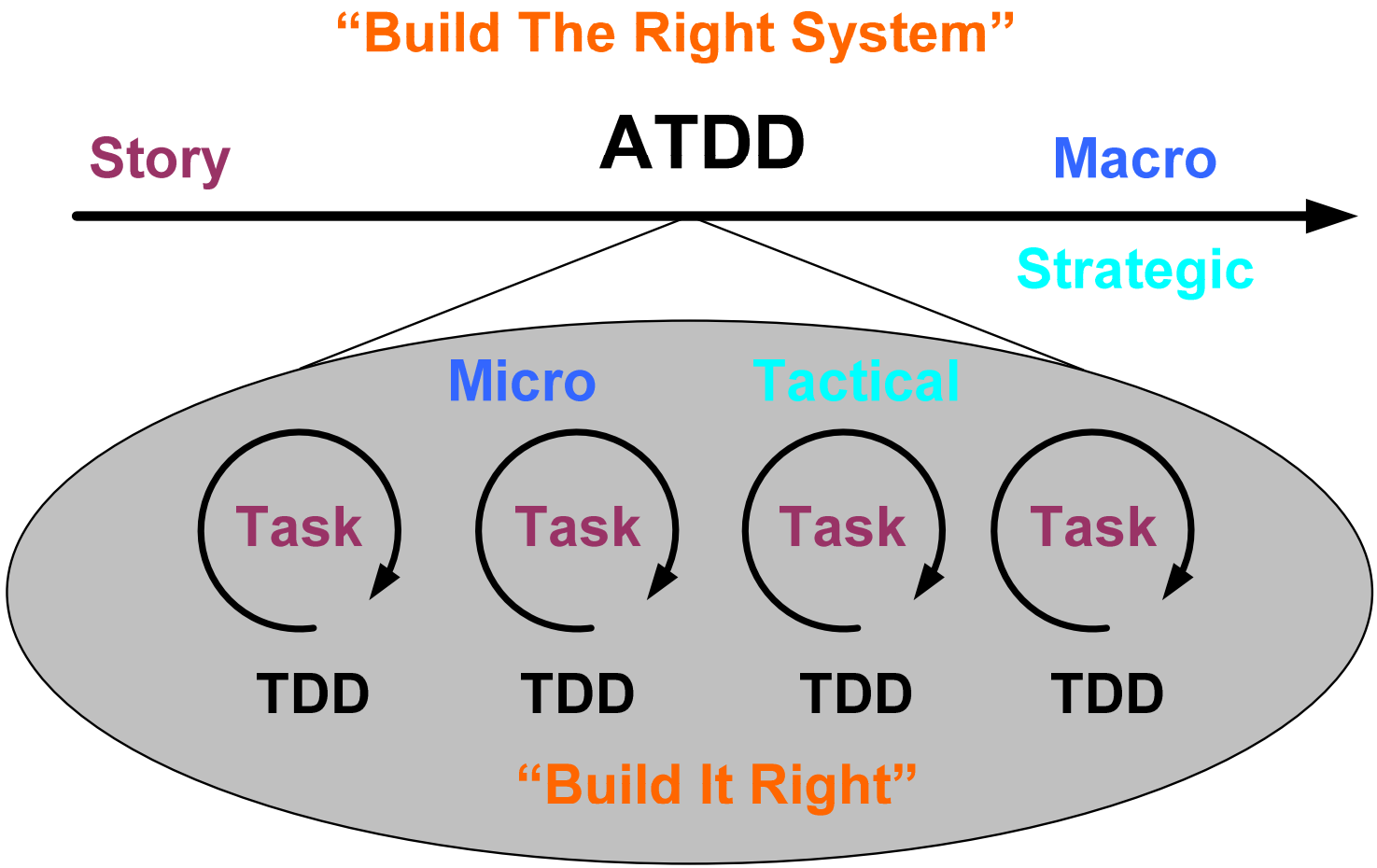
Mockito limitations

- Cannot mock private methods.
- Cannot mock final methods.
- Cannot mock final classes.
- Cannot mock static methods.
- Limited Matchers

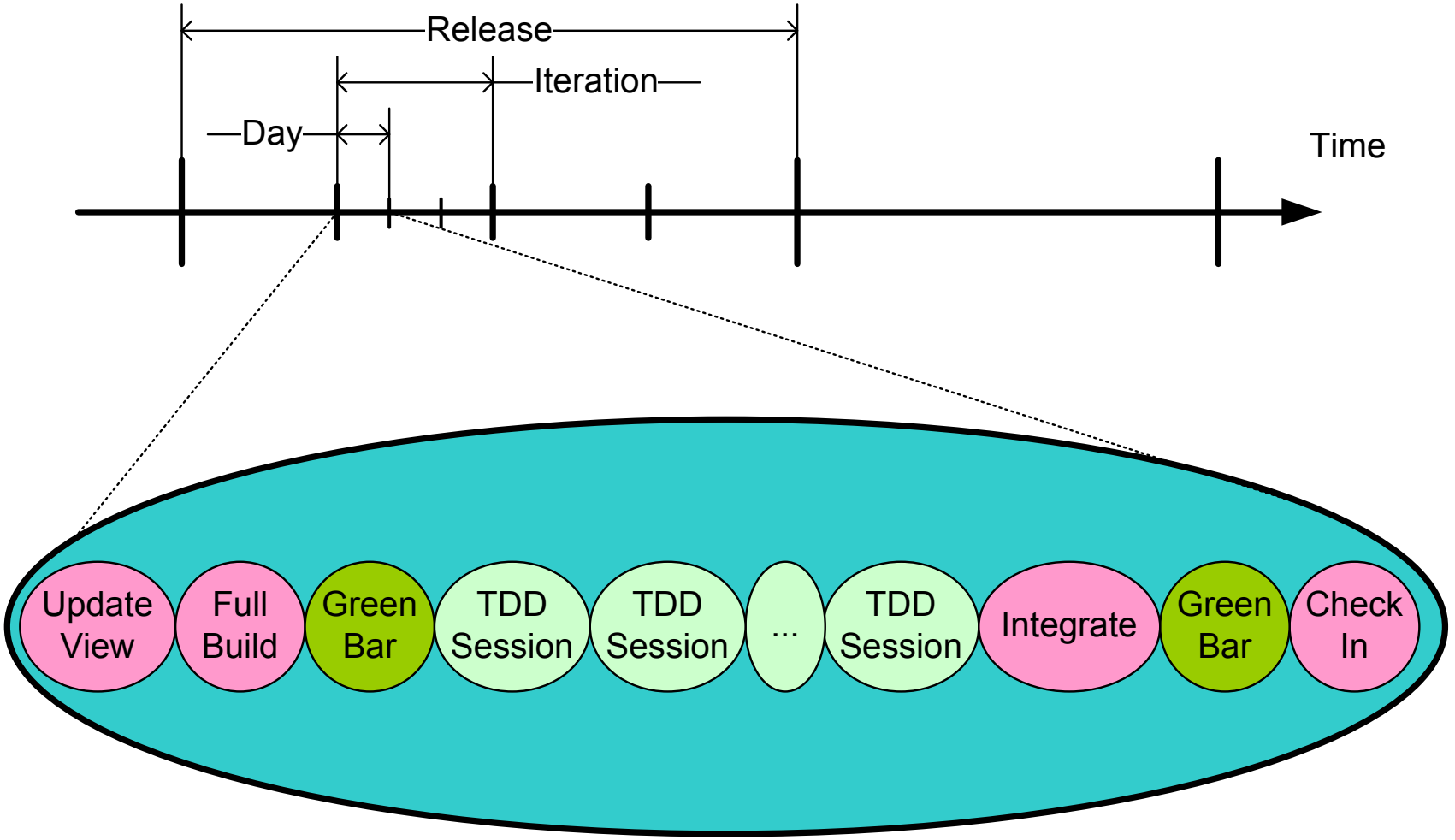
PowerMock with Mockito

- Supports mocking with
  - Private methods
  - Static methods
  - Final methods
  - Final classes
  - Non-static methods
  - Advanced matchers

TDD: The Micro-Cycle of Agile Development Process



TDD: The Micro-Cycle of Agile Development Process



Levels of testing in TDD

- **Acceptance:** Does the whole system work?
- **Integration:** Does our code work against code we can’t change?
- **Unit:** Do our objects do the right thing, are they convenient to work with?

Acceptance in TDD

- Acceptance test should exercise the system end-to-end.
- Acceptance test may generally mean “ functional test”, “customer test” or” system test”.
- For TDD, acceptance test helps us, with the domain experts, understand and agree on what we are going to build next.

Integration test in TDD

- It is to refer to the tests that check how some of our code works with code from outside the team that we can’t change.

TDD gives feedback

- TDD gives us the feedback on both, implementation (“Does it work?”) and design (“ Is it well-structured?”)

Benefits of TDD efforts

- **Writing tests:**

1. Makes us clarify the acceptance criteria for the next piece of work-we have to ask ourselves how we can tell when we are done (design)
2. Encourages us to write loosely coupled components so they can be easily tested in isolation, at higher levels, combined together (design)
3. Adds an executable description of what the code does (design); and
4. Adds to a complete regression suite.

Benefits of TDD efforts

- **Running tests:**

1. Detects errors while the context is still fresh in our mind (implementation) and
2. Lets us know when we've done enough, discouraging "gold-plating" and unnecessary features (design)

External and Internal Quality

- **External Quality:**

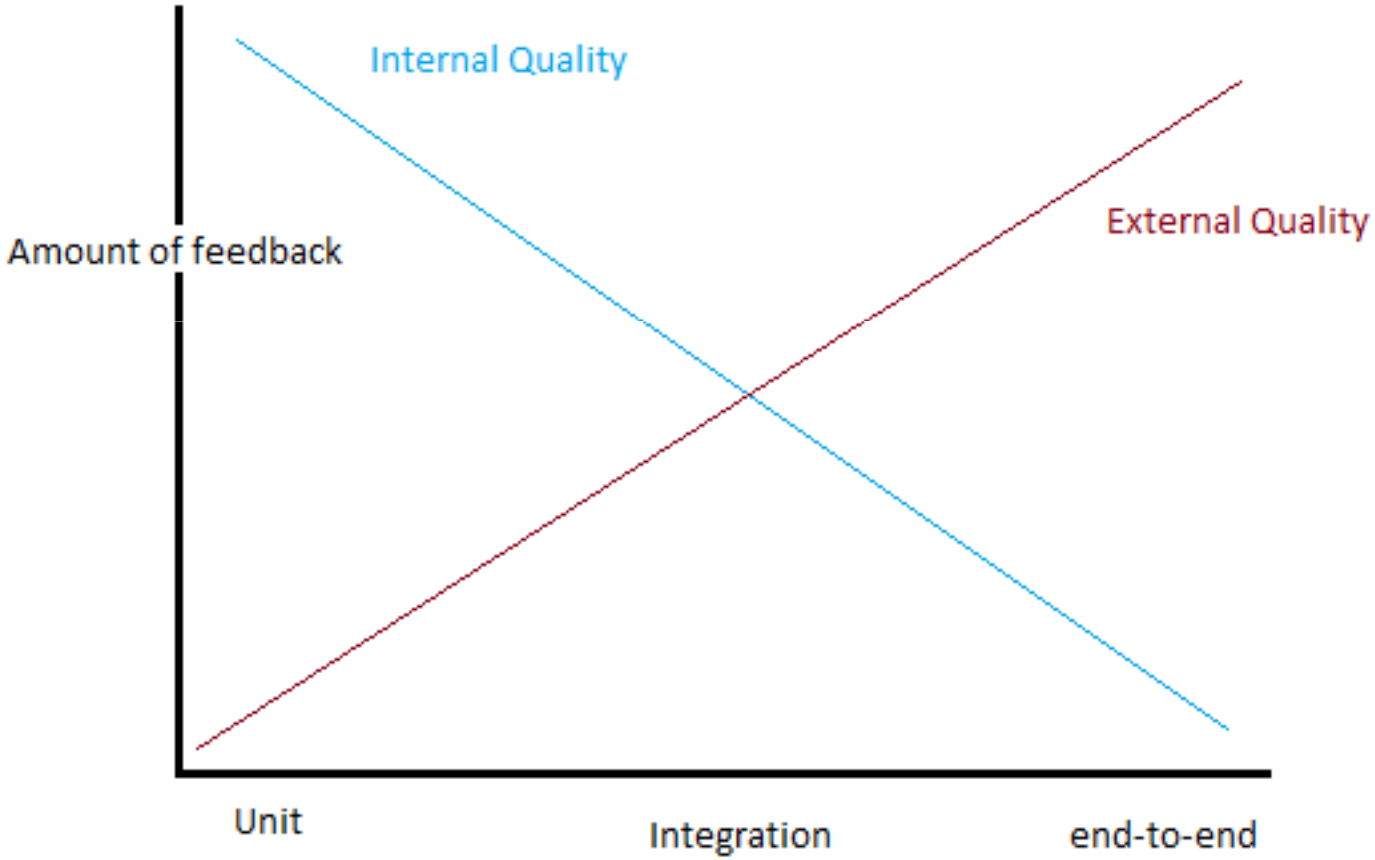
  How well the system meets the needs of its customers and users. (is it functional, reliable, available, responsive etc.) It is usually a part of the contract to build.
- **Internal Quality:**

  How well the system meets the needs of is developers and administrators (Is it easy to understand, easy to change etc.)

Testing and Quality in TDD

- **Running end-to-end tests** gives us the confidence on external quality and **writing** tells us how well we understand the domain. But it doesn’t tell us about code quality.
- **Writing unit tests** gives us a lot of feedback of quality of our code and **running unit tests** tells us that we haven’t broken any classes. But they don’t give us confidence that the system as a whole works.
- Integration test falls in between.

Quality and testing in TDD



Listening to tests

- For Unit testing, an object needs to be created, provide with its dependencies, interact with it and check that it behaved as expected.
- So, for a class to be unit-testable, it must have explicit dependencies that can be easily substituted and clear responsibilities that can be invoked and verified.
- For this, the code must be loosely coupled and highly cohesive.
- If such a test cannot be written for a class, then we need to investigate the reasons why it is hard to write a test and refactor to improve its structure.
- This is called as “Listening to tests”.

Approach of TDD for Legacy systems

Changing software

- To mitigate the risks behind changing software

1. What changes do we have to make?
2. How will we know the we have done them correctly?
3. How will we know that we haven’t broken anything?

Changing software

- We avoid change, conservative approach.
- Why add a method, I'll add code in the existing one.
- If it is not broke, don't fix it.

We can't minimize software problems by avoiding them.

Approaches to make change

Edit and pray

- Working with **care**
- Study the system, make the changes, see if the changes are working, see if something else is broken, pray for everything to be alright.
- Unfortunately, safety is not solely the function of **care**

Cover and modify

- Working with **Safety net**
- **Covering software means covering it with tests.**
- We do apply **care**, but we add **feedback** from tests to do it **more carefully.**
- If changing software means uncertainty, then uncertainty can be removed by

1. Doing little at a time
2. Taking feedback every time.

The Legacy code change algorithm

1. Identify change points
2. Find test points
3. Break dependencies
4. Write tests
5. Make changes and refactor

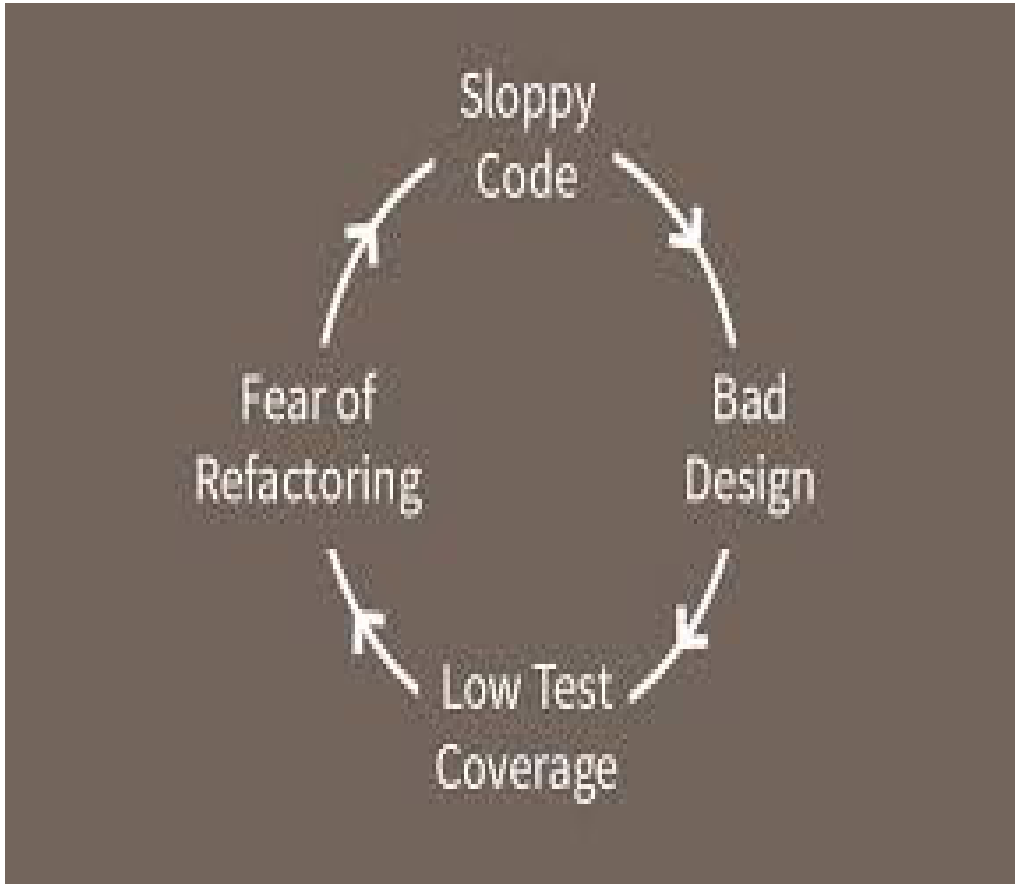
Code Refactoring

AgileSoft Methodologies

Retrospect..

- Duplicate code across the functions of SavingAccount class.
- What if we change the strategy to deal with negative values in deposit and withdraw methods?
- Are we going to modify all the three functions again in the SavingAccount class?

Need of refactoring



What is Refactoring?

- Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
- Refactoring is a series of small transformations while preserving the behavior of code.
- Refactoring is a powerful Agile technique for improving existing software.

Refactoring is code restructuring

- Each transformation (called a 'Refactoring') does little, but a sequence of transformations can produce a significant restructuring.
- The system is also kept fully working after each small Refactoring, reducing the chances that a system can get seriously broken during the restructuring.

Why to refactor ?

- Having source code that is understandable helps ensure a system is maintainable and extensible.
- Improves the design of the software.
- Easier to maintain and understand.
- Easier to facilitate change.
- More flexibility.
- Increased reusability.
- To help find bugs

Preserve and verify with Tests

- Refactoring simply means *"improving the design of existing code without changing its extrenal behavior".*
- The behavior of code **should not** get changed after refactoring.
- The tests execution before and after the refactoring stage should ensure that the behavior of the code is **NOT** changed .
- When changing a lot of the code at one time it is possible that bugs get introduced. But when and where these bug were created may no longer reproducible.
- Hence the refactoring is to be done with **smaller** steps followed by **rigorous testing**, before and after the refactoring stage.

Small change with refactoring

- If a change is implemented in small steps with tests running after each step, the bug likely surfaces in the test run immediately after introducing it into the system.
- Then the step could be examined or, after undoing the step, it could be split in even smaller steps which can be applied afterwards.

Testing after Refactoring

The benefit of comprehensive unit tests in a system, as advocated by Extreme Programming techniques.

These tests, give the developers and management confidence that the refactoring has not broken the system, the code behaves the same way as it behaved before.

Why Do we refactor ?

- Improves the design of the software.
- Easier to maintain and understand.
- Easier to facilitate change.
- More flexibility.
- Increased reusability.
- To help find and locate the bugs

When to refactor ?

- Duplicate code.
- Large class
- Class with multiple different unrelated behavior
- Large switch-case blocks makes it difficult to add one more case block
- Long methods.
- Long parameter list for methods.
- Large no of local variables
- Dead code(un used)
- Hard coupling between classes
- Improper naming of variables/classes/methods.

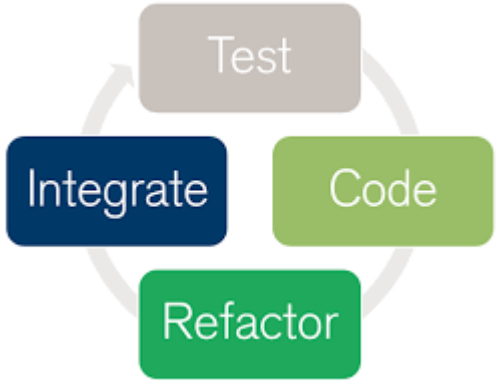
What to refactor ?

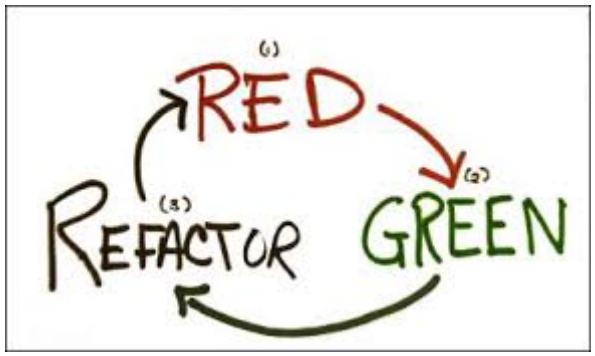
- Programs that are hard to read (hard to modify).
- Programs that have duplicate logic(hard to modify)
- Programs that require additional behavior that requires you to change running code(hard to modify.)
- Programs with complex conditional logic(hard to modify.)

Refactoring Steps..

Each transformation (called a 'Refactoring') does little, but a sequence of transformations can produce a significant restructuring.

The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring.





Refactor and go GREEN!..

- Extract the functions out of existing duplicate code.
- Remove duplications without changing the external behaviors of functions.. All the tests must succeed.
- Run all the tests again..
- It must be GREEN bar always..

Refactoring advantage

- The development tasks, maintenance and enhancement, often conflict since new features, especially those that do not fit cleanly within the original design, result in an increased maintenance effort.
- The refactoring process aims to reduce this conflict, by aiding non destructive changes to the structure of the source code, in order to increase code clarity and maintainability.

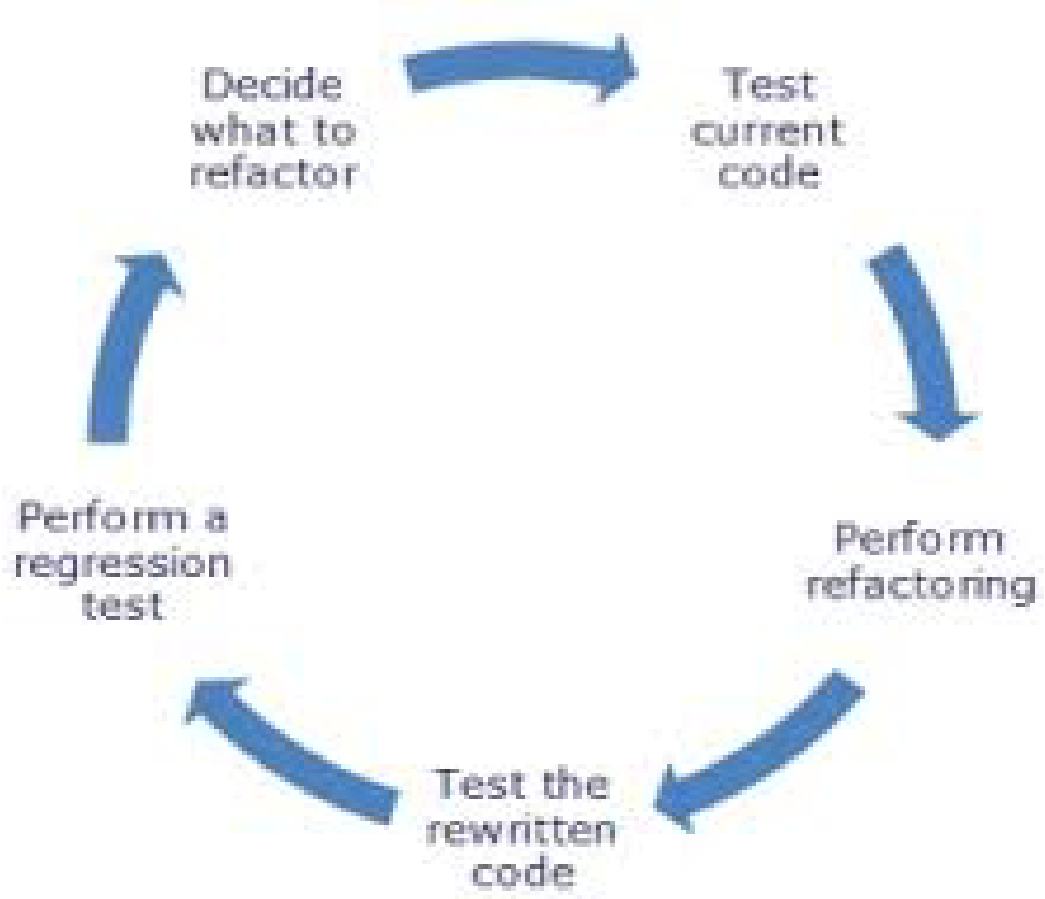
Benefit of refactoring

- For the well structured code, new requirements can be introduced more efficiently and with less problems.
- However people are hesitant to use refactoring Because of the amount of effort required for even a minor change, and a fear of introducing bugs.
- These problems can be solved by using an automated refactoring tool and well defined set of test cases to ensure re-testing of re-factored code again and again.

Code Clarity

When system's source code is easily understandable, the system is more maintainable, leading to reduced costs and allowing precious development resources to be used elsewhere.

Refactoring in TDD Cycle



Refactoring techniques

- Extract method
- Inline method
- Extract variable
- Extract class
- Extract super class
- Extract interface
- Rename
- Move class in related packages
- Replace Conditional with Polymorphism

Refactoring to patterns

- Replace state altering conditionals with state pattern e.g. door close/open
- Move creation knowledge to Factory
- Extract Composite
- And on and on…..

Refactoring with tools

- The automated tools are available to enhance code quality by safely performing refactoring tasks.
- If refactoring are carried out manually, one needs to frequently rebuild the system and run tests.
- Manual refactoring is applicable the following conditions hold:
- The system, of which the refactored code is a part, can be rebuilt *quickly.*
- There are automated "regression" tests that can be frequently run to verify the impact of refactoring.

Addition to the case study

Currency abstraction

- Extract the top level abstract class as Currency from Dollar and Euro classes and override the required methods
- Currency as top level abstract class with required abstract methods.
- Define Dollar and Euro classes extending and overriding the currency conversion and value methods.

The Currency Parent


```
package com.server.bank;

abstract class Currency {

abstract double getRupeesConvresionRate() ;

abstract int getCurrencyValue() ;

}
```

The Dollar revised


```
package com.server.bank;
public class Dollar extends Currency {
 private int currencyValue;
int getCurrencyValue() {
     return currencyValue;
}
private static int rupeesConvresionRate = 48.6 ;
Int getRupeesConvresionRate() {
     return rupeesConvresionRate;
}
public Dollar(int currentValue) {
      this.currencyValue = currentValue;
 }
}
```

Revised Euro Class

```
package com.server.bank;
public class Euro extends Currency {
   private int currencyValue;
   private static int rupeesConvresionRate = 25 ;
Euro(int currentValue) {
     this.currencyValue = currentValue;
 }
Int getRupeesConvresionRate() {
    return rupeesConvresionRate;
}
 public int getCurrencyValue() {
    return currencyValue;
}
}
```

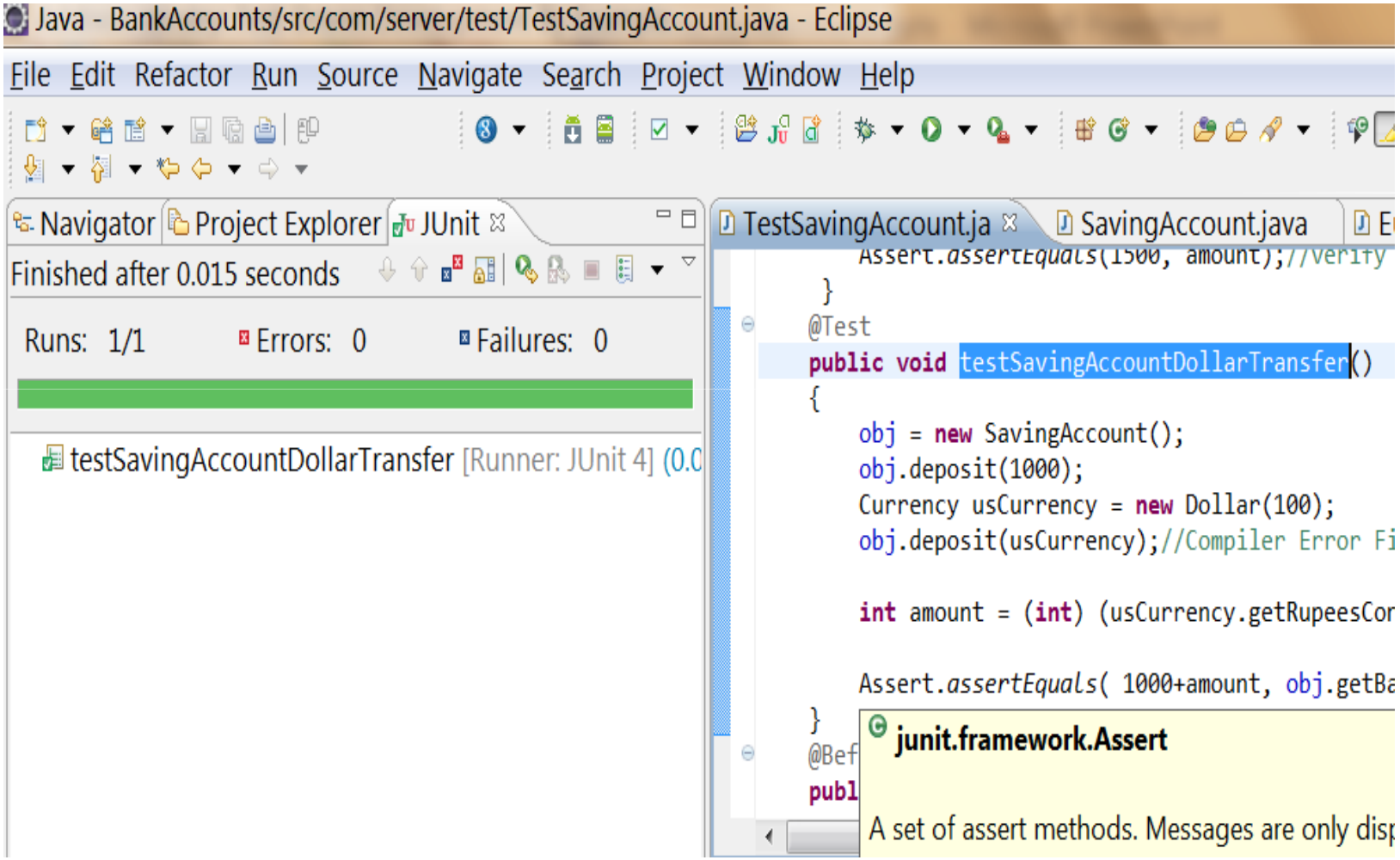
Modify the SavingAccount


```
public class SavingAccount {

private int balance;

public void deposit(int amount) {
balance+= amount;
}
   public void deposit(Currency  currencyType) {
     double amount = currencyType.getRupeesConvresionRate() *
     currencyType.getCurrencyValue();
     balance+= amount;
}
```

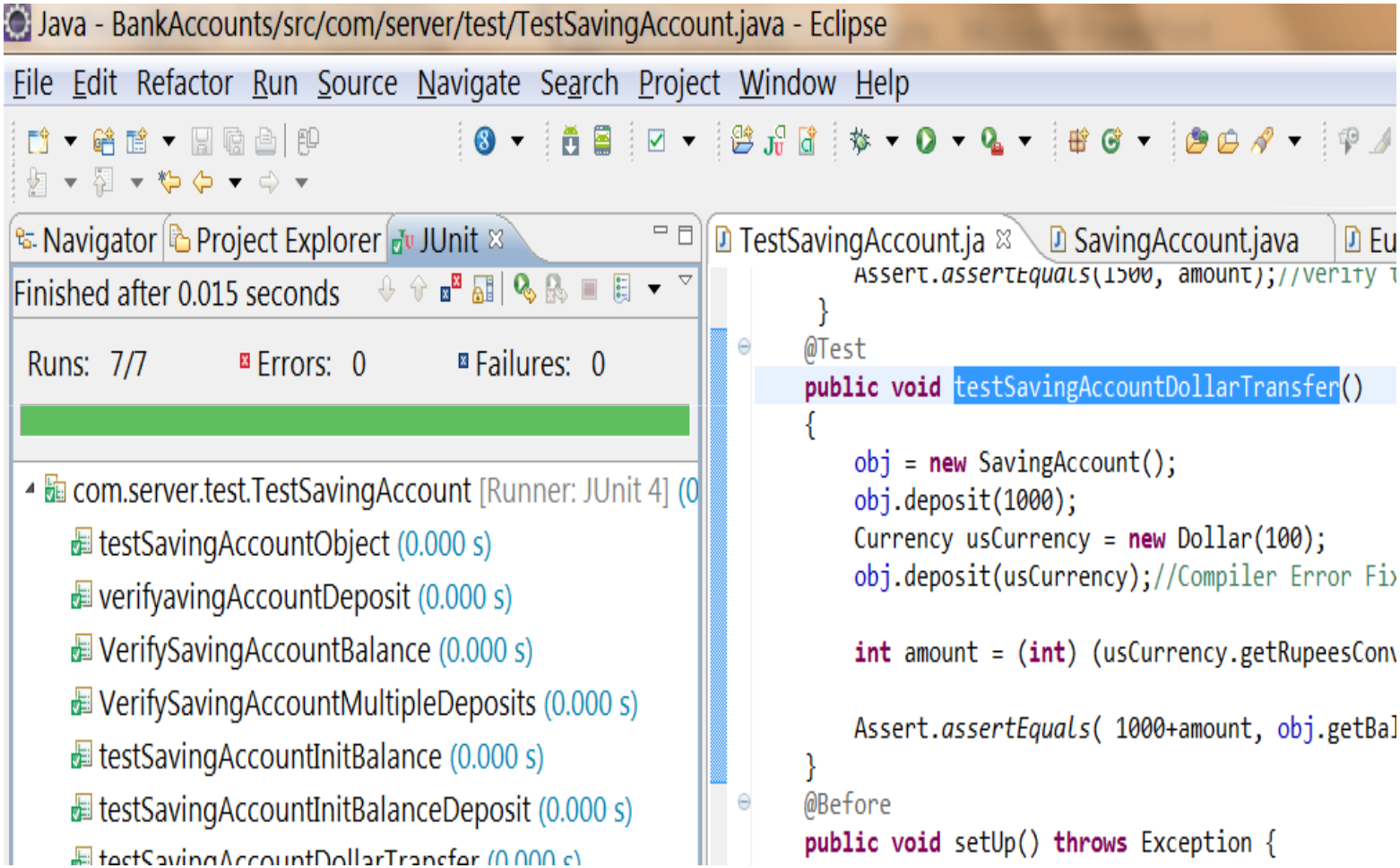
Update the test method code


```
@Test
 public void testSavingAccountDollarTransfer()
 {
    obj = new SavingAccount();
    obj.deposit(1000);
    Currency usCurrency = new Dollar(100);
    obj.deposit(usCurrency);

    int amount = (int) (usCurrency.getRupeesConvresionRate() *
   usCurrency.getCurrencyValue());

  //verify the current  balance
    Assert.assertEquals( 1000+amount, obj.getBalance());

 }
```

Run the test


Java - BankAccounts/src/com/server/test/TestSavingAccount.java - Eclipse
File Edit Refactor Run Source Navigate Search Project Window Help
Navigator
Project Explorer
JUnit
Finished after 0.015 seconds
Runs: 1/1
Errors: 0
Failures: 0
testSavingAccountDollarTransfer [Runner: JUnit 4] (0.0
TestSavingAccount.ja
SavingAccount.java
Assert.assertEquals(1500, amount);//verify
}
@Test
public void testSavingAccountDollarTransfer()
{
obj = new SavingAccount();
obj.deposit(1000);
Currency usCurrency = new Dollar(100);
obj.deposit(usCurrency);//Compiler Error Fi
int amount = (int) (usCurrency.getRupeesCor
Assert.assertEquals(1000+amount, obj.getBa
}
@Bef
publ
junit.framework.Assert
A set of assert methods. Messages are only disp

Run all the Tests



Next User story

SavingAccount – withdraw amount from account balance and verify with getBalance.

TestCase for the withdraw

In the test class add another test function TestSavingAccountWithdraw

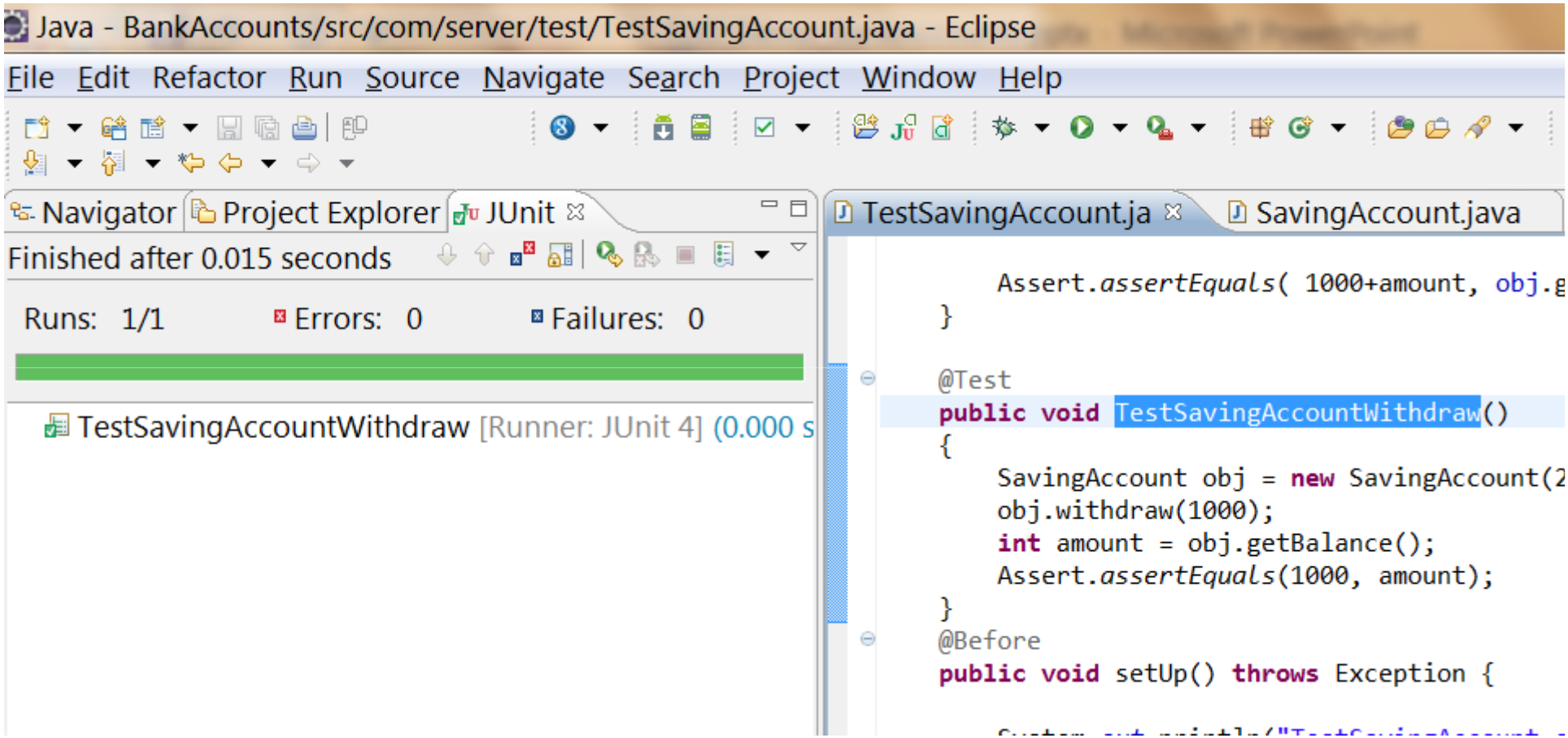
TestSavingAccountWithdraw


```
@Test
public void TestSavingAccountWithdraw()
   {
      SavingAccount obj = new SavingAccount(2000);
      obj.withdraw(1000);  //Compiler Error..Fix It!
      int amount = obj.getBalance();
      Assert.assertEquals(1000, amount);
   }
```

SavingAccount - withdraw

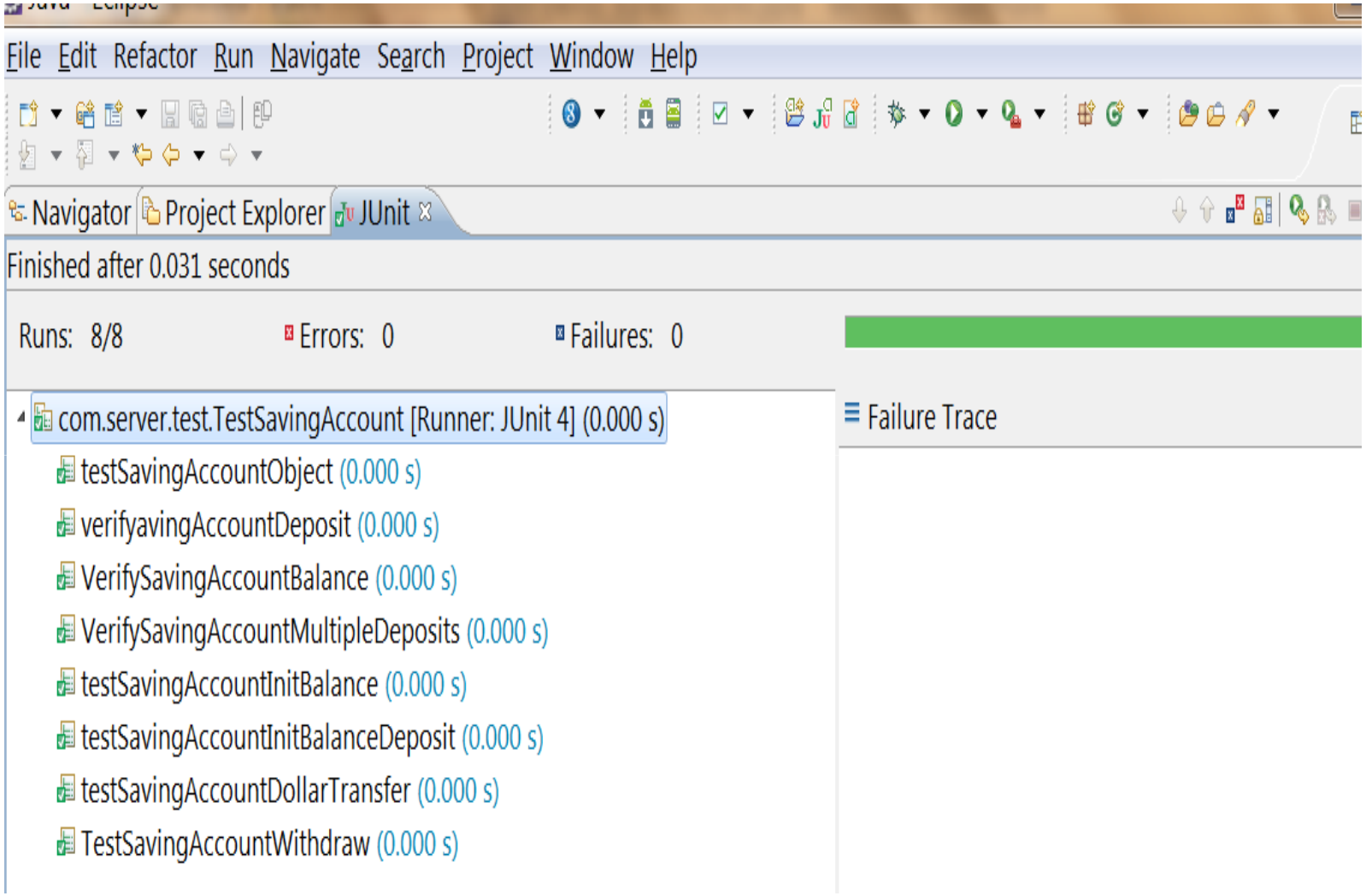

```
public void withdraw(int amount)
    {
       //obvious implementation
       balance-= amount;

    }
```

Run the Test



Cheers! We implemented successful function right at first..

Run all the tests …



Huryo!!! All the tests have been successful..

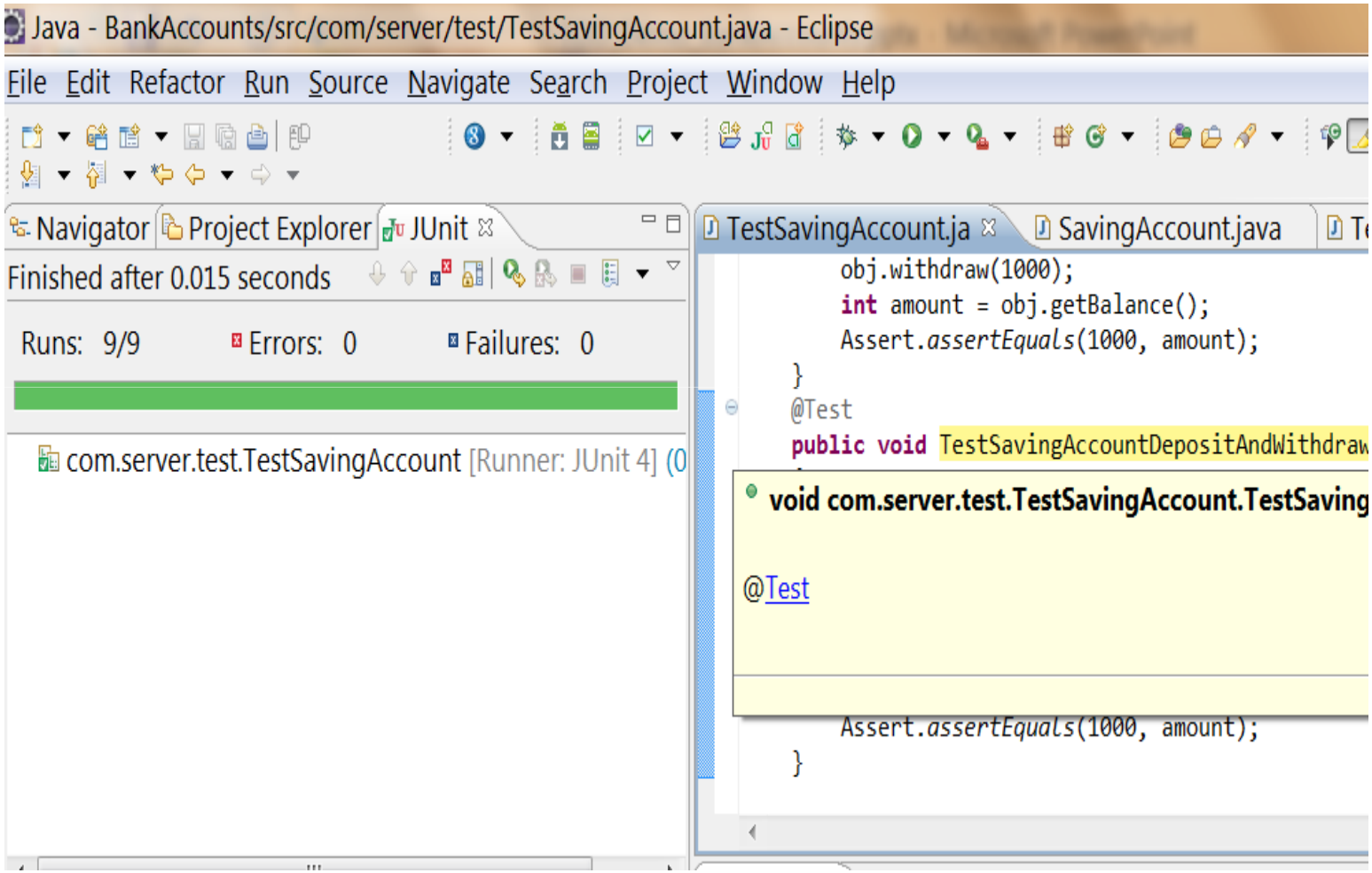
What if

- If I invoke withdraw after deposit, I must get the cumulative balance added by deposit and subtracted by withdraw..
- Add another test function TestSavingAccountDepositAndWithdraw in Test class.

TestSavingAccountDepositAndWithdraw

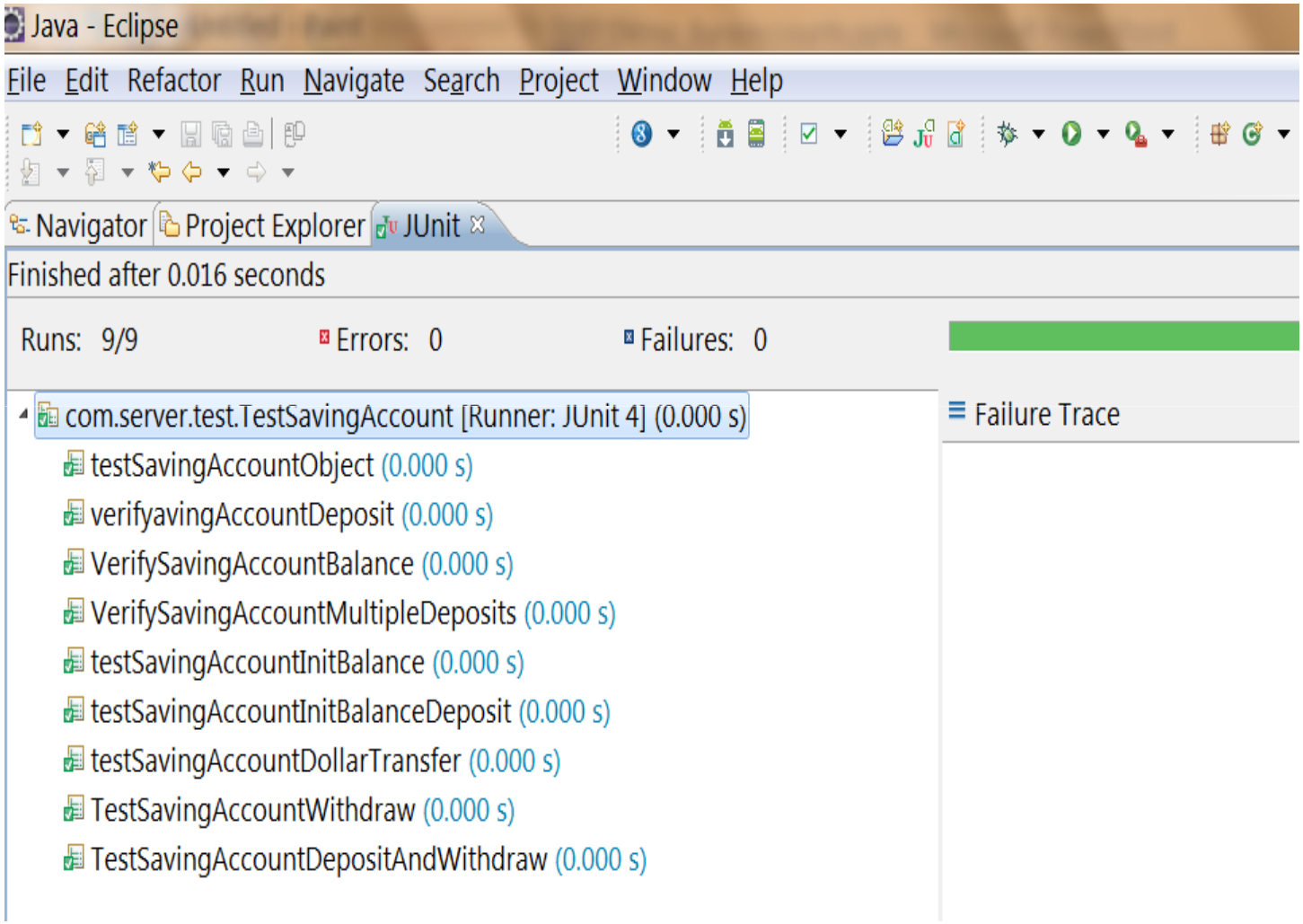

```
@Test
public void TestSavingAccountDepositAndWithdraw()
{
    obj.deposit(2000);

    obj.withdraw(2000);

    int amount = obj.getBalance();

    Assert.AreEqual(1000, amount);
}
```

Run the Test



Cheers! This time also we were successful..

Run all the tests..



What If

- If I try to withdraw more amount than the balance amount, I should get error WithDrawAmountMoreThanBalance and balance intact..
- Add another test function TestSavingAccountWithdrawMoreAmount in TestSavingAccount class.
- Define new class WithDrawAmountMoreThanBalance.

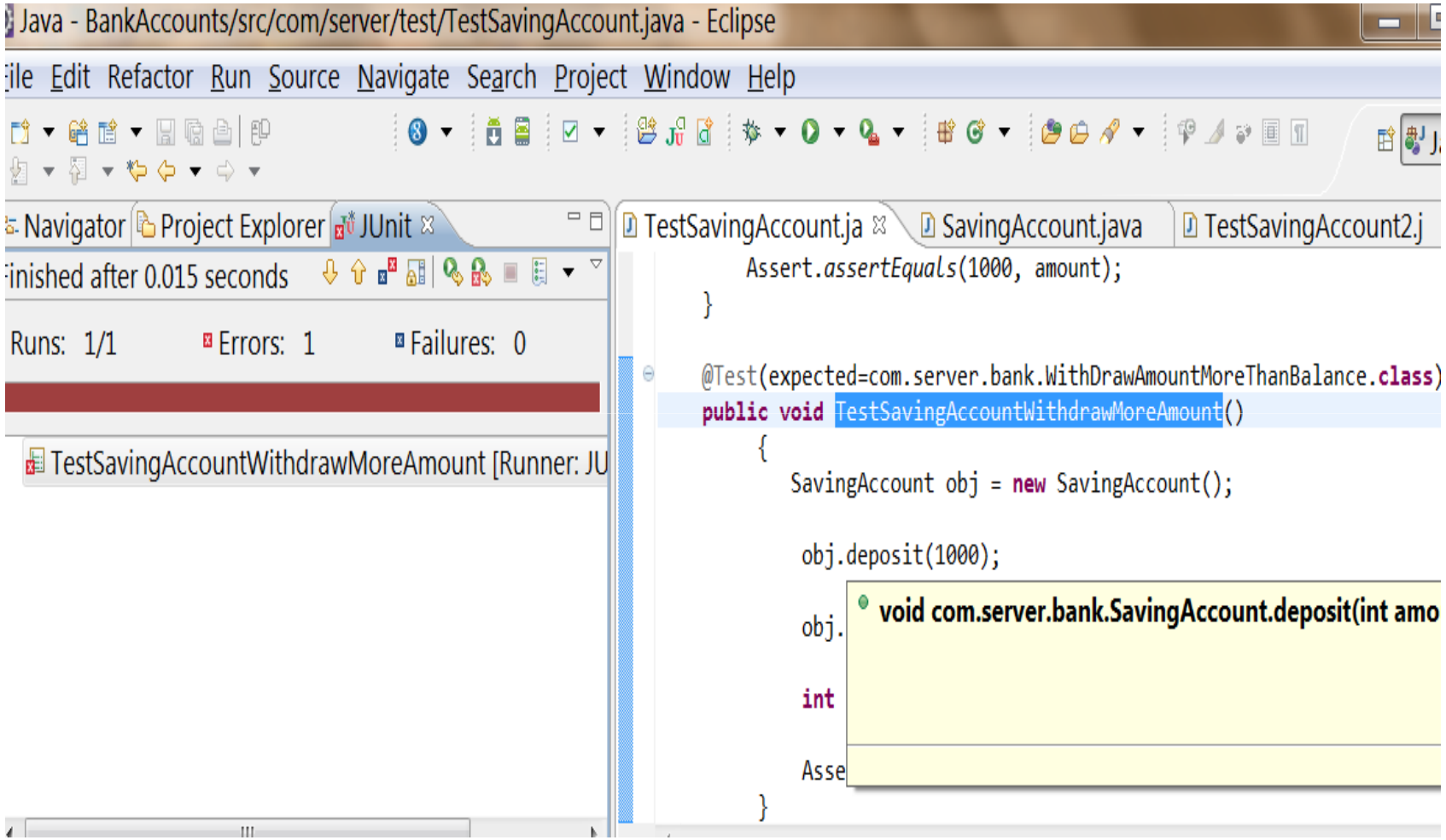
TestSavingAccountWithdrawMore_Amount


```
@Test(expected=com.server.bank.WithDrawAmountMoreThanBalance.class)
   public void TestSavingAccountWithdrawMoreAmount()
       {
         SavingAccount obj = new SavingAccount();

          obj.deposit(1000);

          obj.withdraw(3000);

          int amount = obj.getBalance();

          Assert.assertEquals(2000, amount);
       }
```

The Exception class

```
package com.server.bank;

public class WithDrawAmountMoreThanBalance
    extends RuntimeException {

public WithDrawAmountMoreThanBalance(String error)
    {
     super(error);
}

}
```

Run the Test..Let it fail!

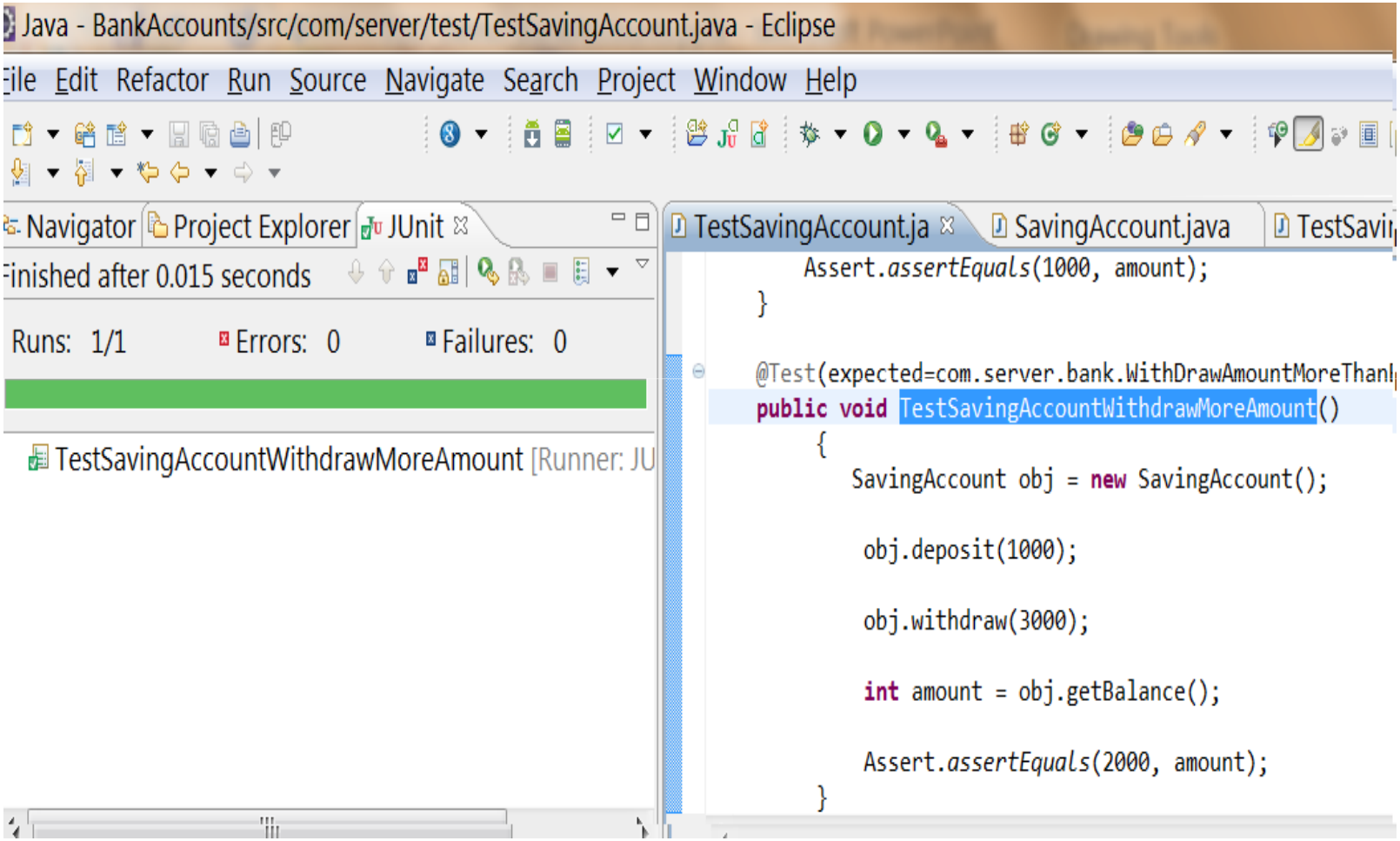


The test failed..expected exception not thrown..

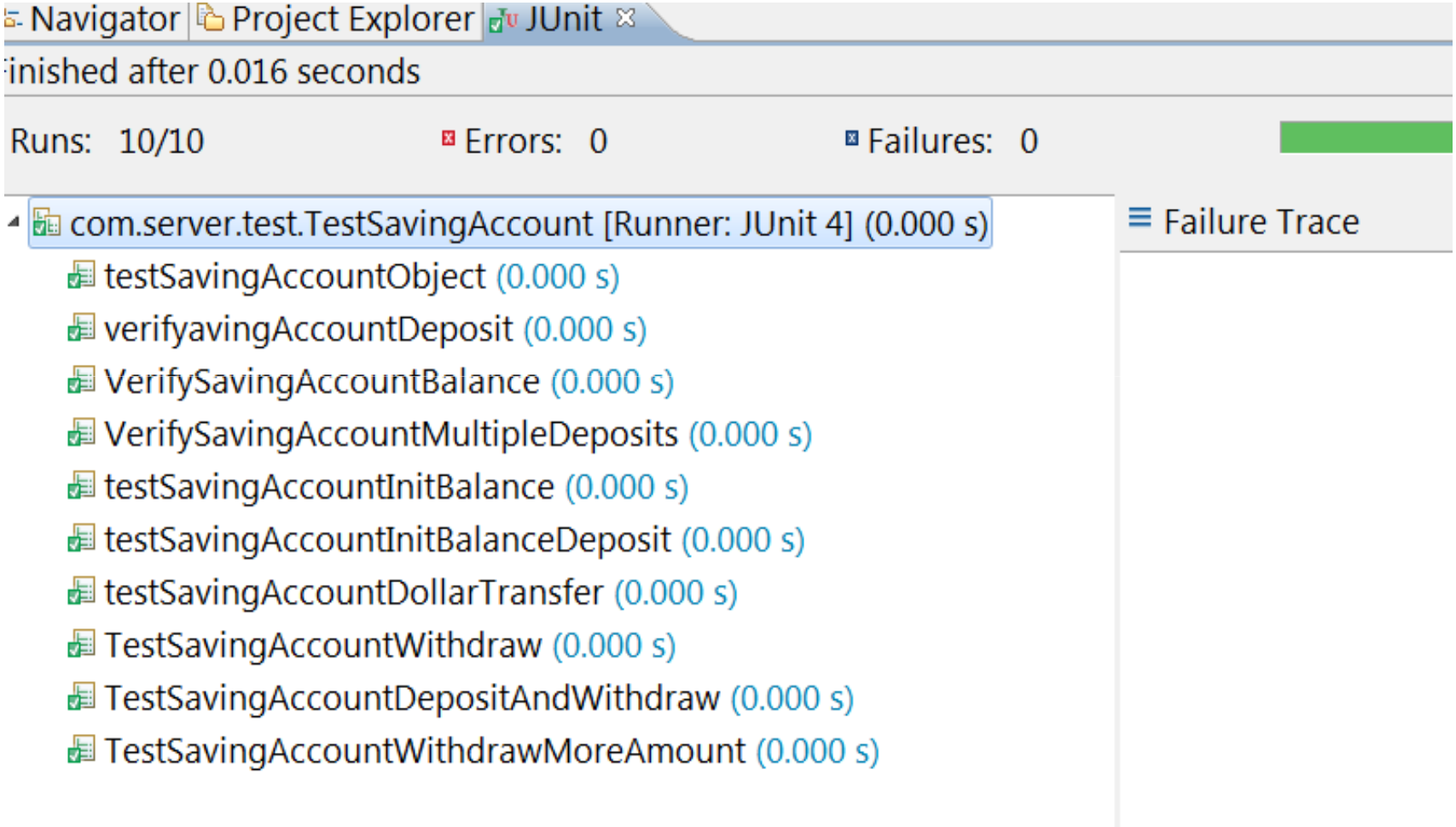
Modify withdraw

```
public void withdraw(int amount)
   {

   if(amount  >  balance )
   {
     throw new WithDrawAmountMoreThanBalance("More amount
withdrawn");
   }

    //obvious implementation
     balance-= amount;
   }
```

Run The Test


Java - BankAccounts/src/com/server/test/TestSavingAccount.java - Eclipse
File Edit Refactor Run Source Navigate Search Project Window Help
Navigator
Project Explorer
JUnit
Finished after 0.015 seconds
Runs: 1/1
Errors: 0
Failures: 0
TestSavingAccountWithdrawMoreAmount [Runner: JU
TestSavingAccount.ja
SavingAccount.java
Assert.assertEquals(1000, amount);
}
@Test(expected=com.server.bank.WithDrawAmountMoreThanI
public void TestSavingAccountWithdrawMoreAmount()
{
SavingAccount obj = new SavingAccount();
obj.deposit(1000);
obj.withdraw(3000);
int amount = obj.getBalance();
Assert.assertEquals(2000, amount);
}

Run All the Tests



How much we wrote ?

- Total 10 test functions to test on SavingAccount class.
- Evolved 5 functions on SavingAccount class.
  - SavingAccount default constructor
  - SavingAccount int parameterized constructor
  - deposit(int amount);
  - withdraw(int amount);
  - getBalance(void);

What if..

- What if I want to have initial balance for SavingAccount?
- Define another test function testSavingAccountInitBalance in the TestSavingAccount class.

The Test code

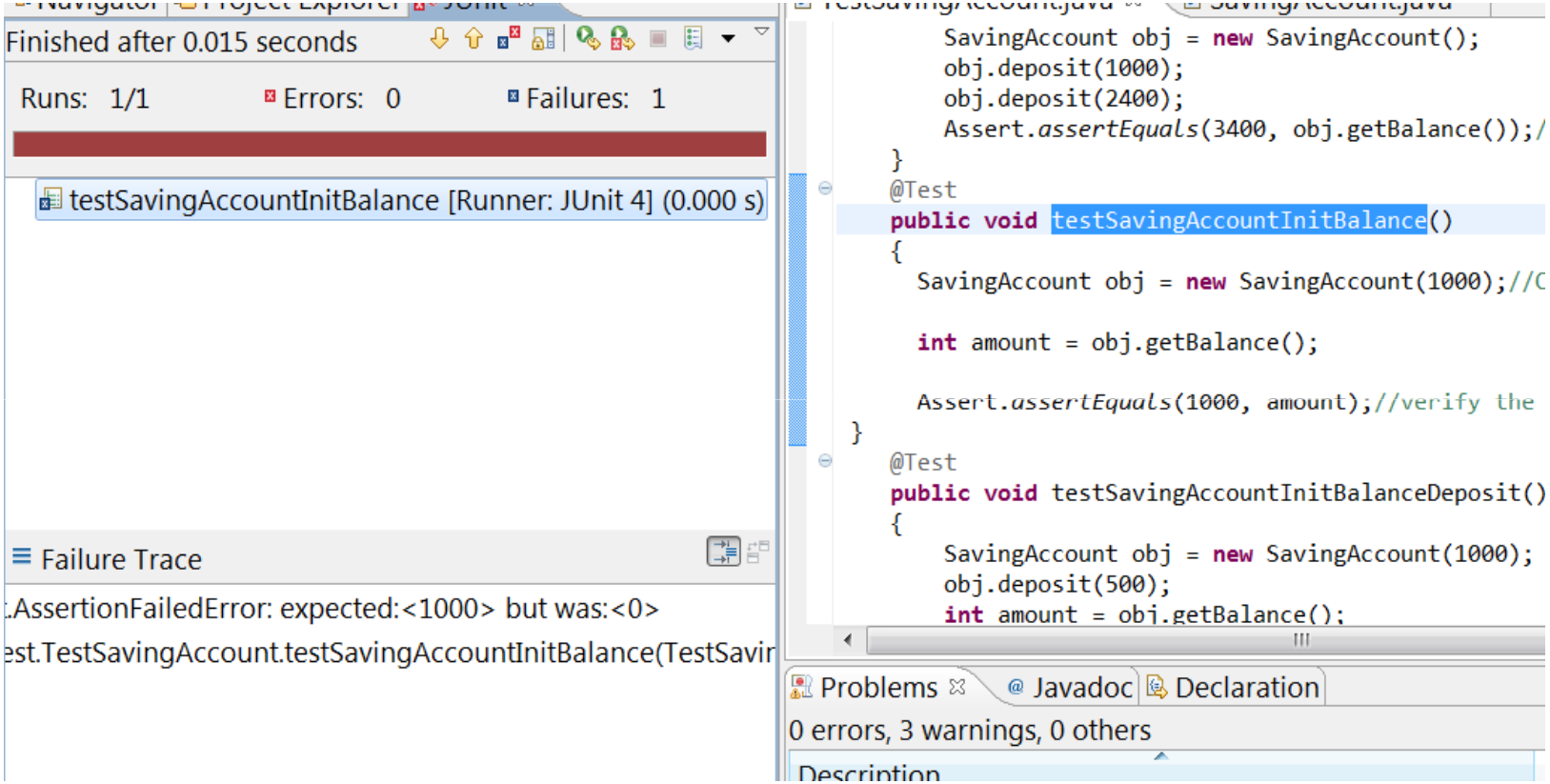
```
@Test
public void testSavingAccountInitBalance()
    {
     SavingAccount obj = new SavingAccount(1000);//Compiler
   Error..Fix it

     int amount = obj.getBalance();

     Assert .assertEquals(1000, amount);//verify the current  balance
   }
```

- Add the parameterized constructor in SavingAccount class
- You will be also required to add the definition of default constructor for the SavingAccount class !

SavingAccount with Initial Balance


```
package com.server.bank;

    public class SavingAccount
    {
        private int balance;
       //parameterized constructor
      public SavingAccount(int amount)
      {

      }
      //default constructor
      public SavingAccount()
      {

      }.....
```

Run the Test


Finished after 0.015 seconds
Runs: 1/1
Errors: 0
Failures: 1
testSavingAccountInitBalance [Runner: JUnit 4] (0.000 s)
Failure Trace
AssertionFailedError: expected:<1000> but was:<0>
est.TestSavingAccount.testSavingAccountInitBalance(TestSavir
SavingAccount obj = new SavingAccount();
obj.deposit(1000);
obj.deposit(2400);
Assert.assertEquals(3400, obj.getBalance());
}
@Test
public void testSavingAccountInitBalance()
{
SavingAccount obj = new SavingAccount(1000);//C
int amount = obj.getBalance();
Assert.assertEquals(1000, amount);//verify the
}
@Test
public void testSavingAccountInitBalanceDeposit()
{
SavingAccount obj = new SavingAccount(1000);
obj.deposit(500);
int amount = obj.getBalance();
Problems
Javadoc
Declaration
0 errors, 3 warnings, 0 others
Description

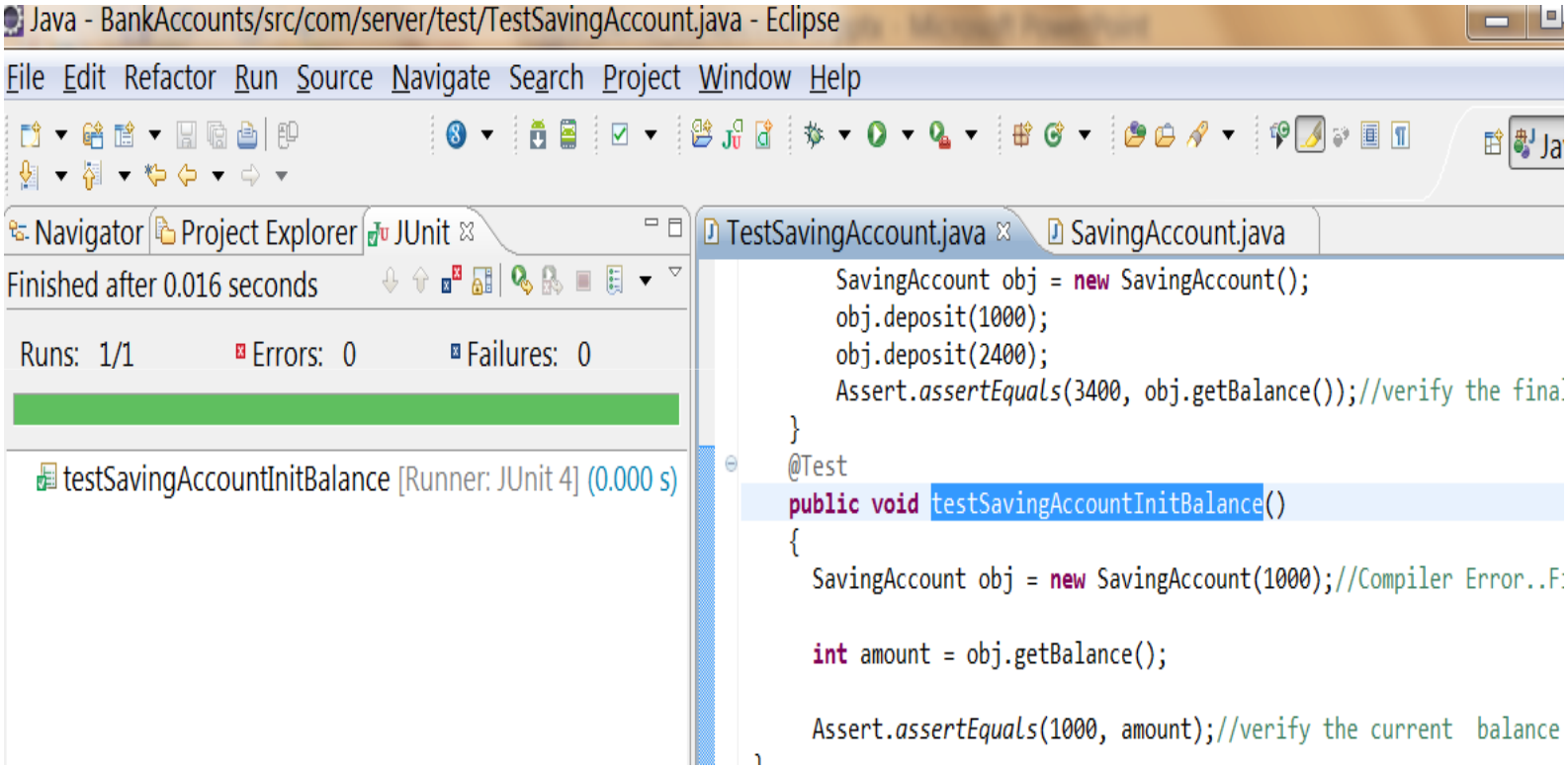
The test failed..

- What is the obvious implementation to make this test success ?

```
public SavingAccount(int amount)
{
    this.balance = amount;
}
```

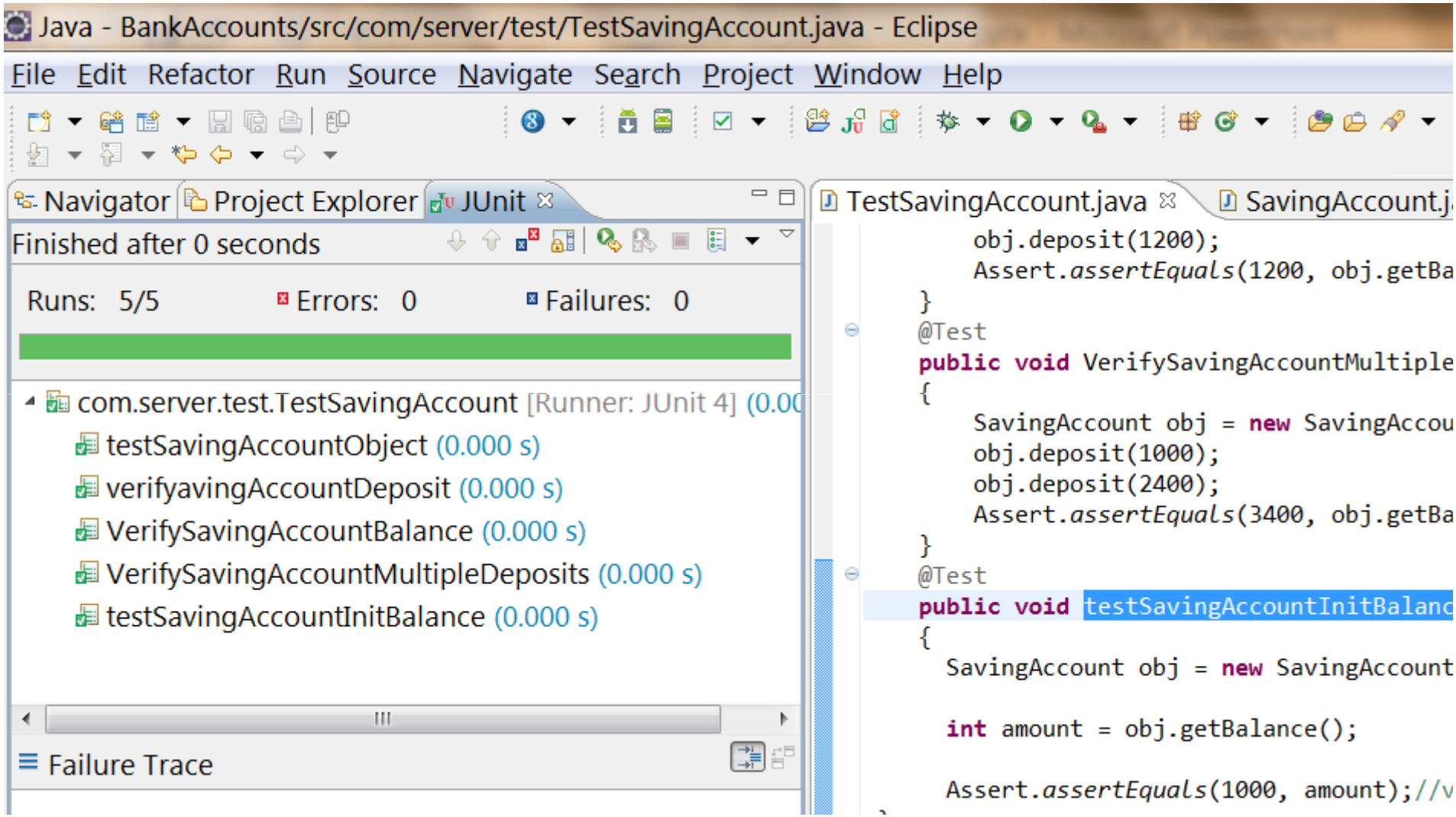
We know the obvious code..Not necessary to fake it here..

Run the Test



The obvious way to Success!

Run all the Tests..



The savingAccount object with initialBalance succeeded..

Test with Initial balance and Deposit

- Add another test function in TestSavingAccount class ‘testSavingAccountInitBalanceDeposit’ and verify result with getBalance on SavingAccount object..
- **@Test**
- **public void testSavingAccountInitBalanceDeposit()**

  {

  SavingAccount obj = **new SavingAccount(1000);**

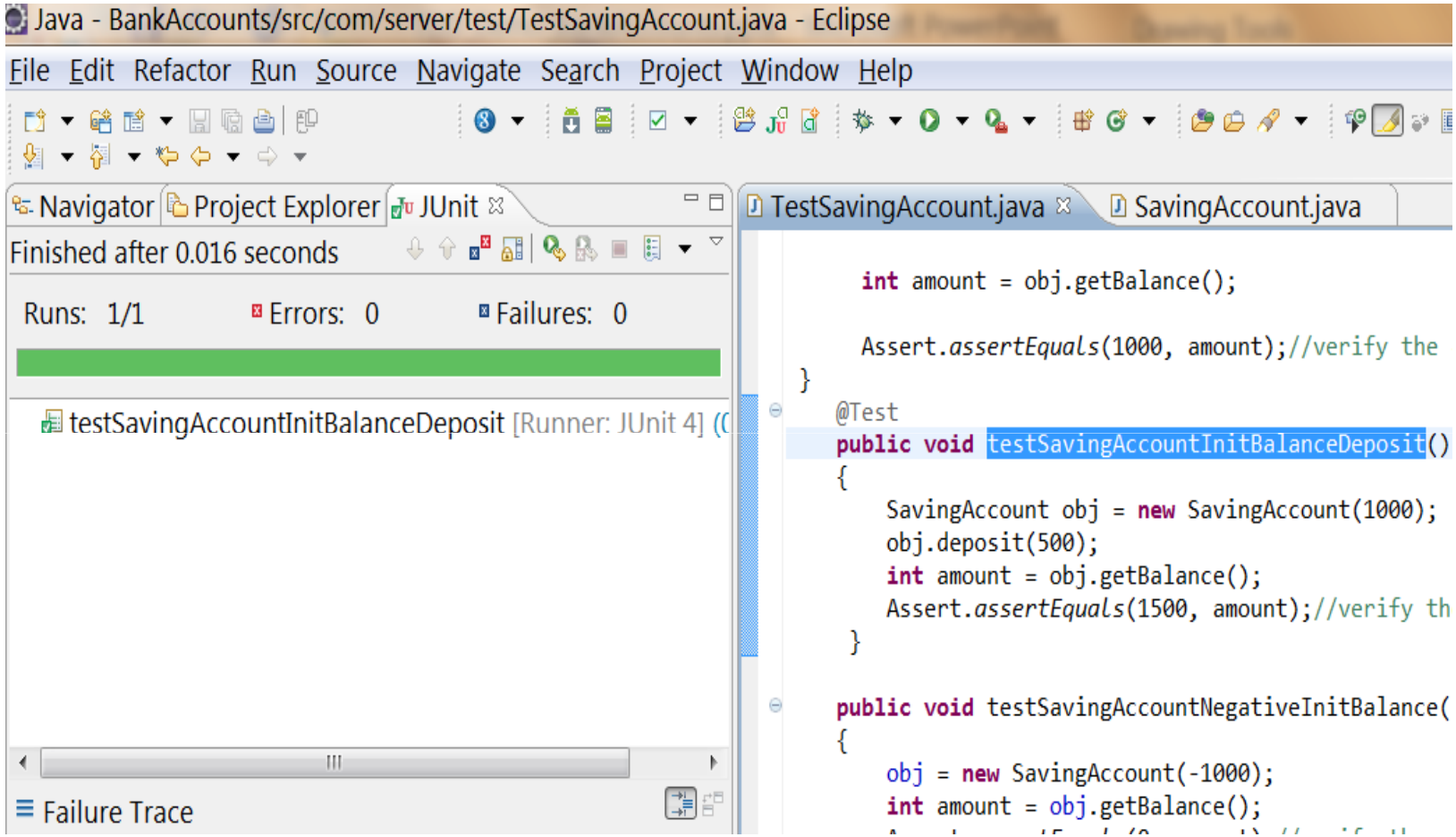
  obj.deposit(500);

  int amount = obj.getBalance();

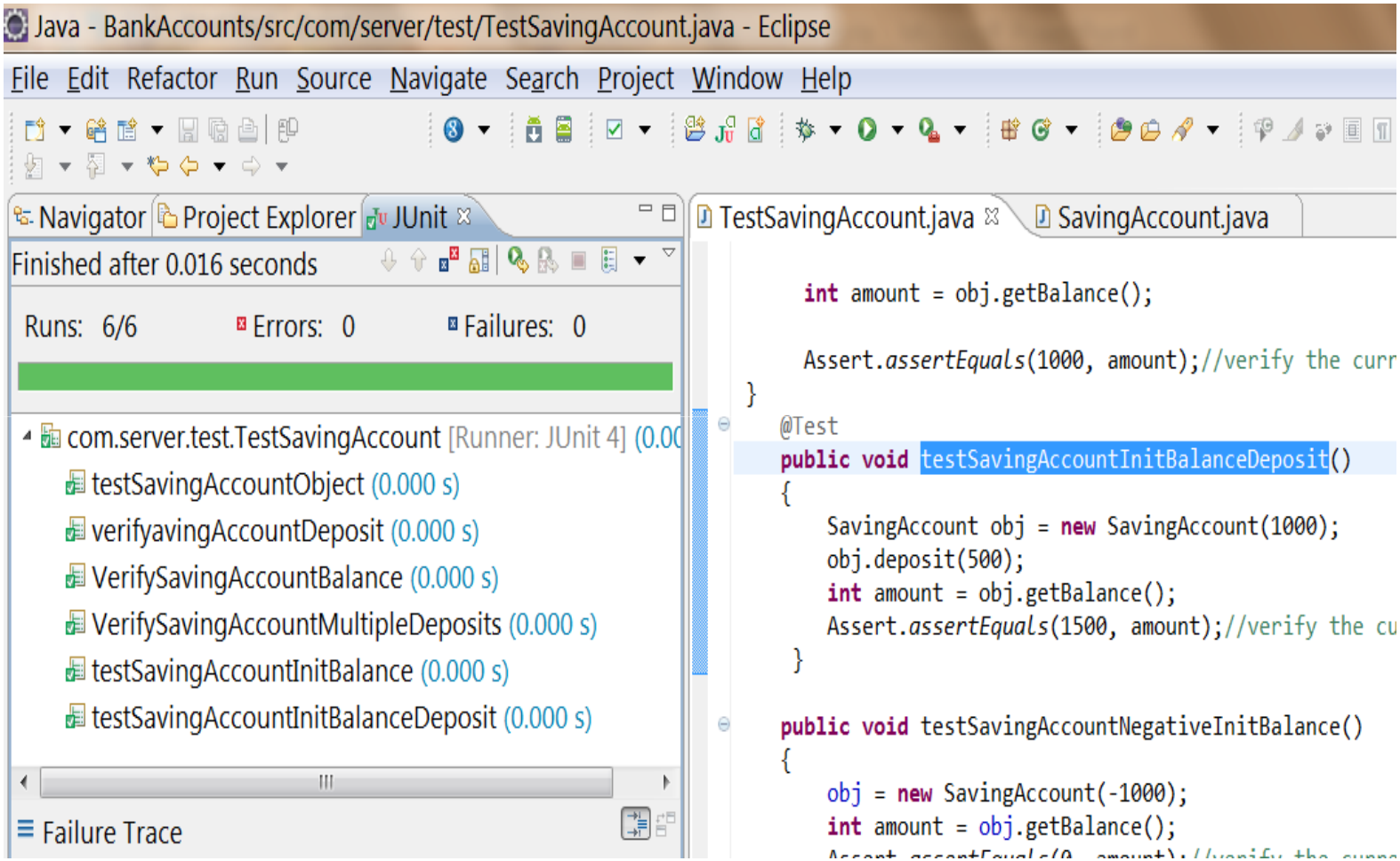
  Assert .*assertEquals(1500, amount);//verify the current balance*

- }
- Rebuild the test and run the tests..

Test result..


Java - BankAccounts/src/com/server/test/TestSavingAccount.java - Eclipse
File Edit Refactor Run Source Navigate Search Project Window Help
Navigator
Project Explorer
JUnit
Finished after 0.016 seconds
Runs: 1/1
Errors: 0
Failures: 0
testSavingAccountInitBalanceDeposit [Runner: JUnit 4] (
Failure Trace
TestSavingAccount.java
SavingAccount.java
int amount = obj.getBalance();
Assert.assertEquals(1000, amount);//verify the
}
@Test
public void testSavingAccountInitBalanceDeposit()
{
SavingAccount obj = new SavingAccount(1000);
obj.deposit(500);
int amount = obj.getBalance();
Assert.assertEquals(1500, amount);//verify th
}
public void testSavingAccountNegativeInitBalance(
{
obj = new SavingAccount(-1000);
int amount = obj.getBalance();

Run all the tests again..



The savingAccount object with all tests succeeded..

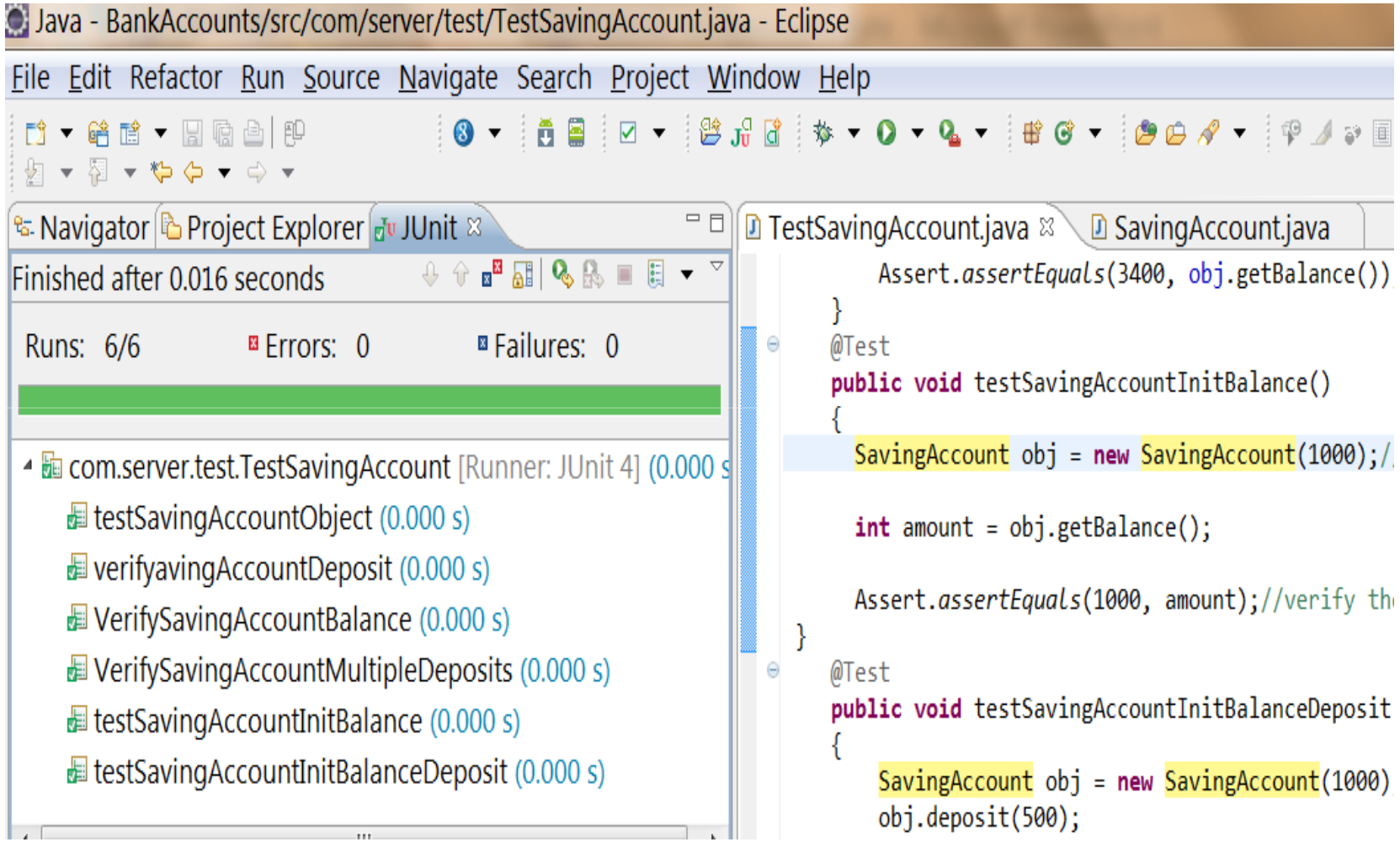
Relax!

- Now we have successfully implemented **deposit** and **getBalance** functionality in SavingAccount and also tested it..
- We have written couple of test methods and accordingly added required minimum code into the domain class SavingAccount..It is the *TDD Approach*!

Sharing common object across the tests..

- Can we share the single object of SavingAccount to all the test methods instead of creating a new one every time ?
- Define a private Saving Account object variable in Test class ,initialize in the TestClass constructor and use the same object across all the test methods..
- Rebuild the Test and run all the tests again..

Run all the tests again….



TDD Mantra for Final Success

- Fake the result in code->Test the code->Implement the obvious code ->Test it again->Fail ->Modify->Test->Success->Modify->Test->Failure->Update->Test Final->Success…
- Learn from our own Success and Failures!!!

Observation

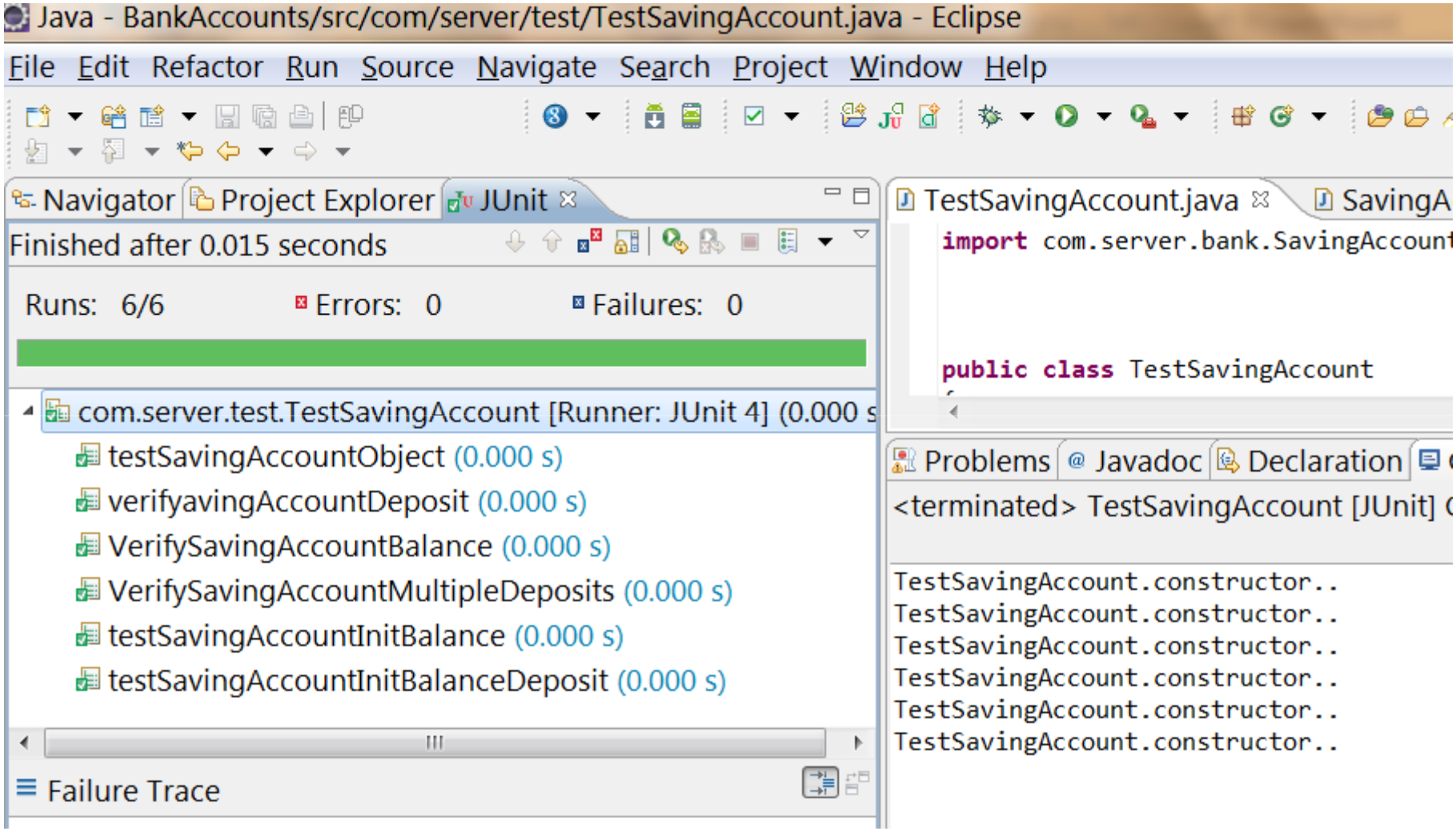
- When we run all the tests, the deposit amount passed in every test method was different..
- But still our all the tests have verified successfully their own balances even though there was a common shared SavingAccount object across all the test methods..
- How this is possible ?

Analyze..

- In TestSavingAccount class add the default constructor with console print statement and run all the tests again and watch the console.
- **public TestSavingAccount()**

```
{
System.out.println("TestSavingAccount.constructor..");
}
```

Console view



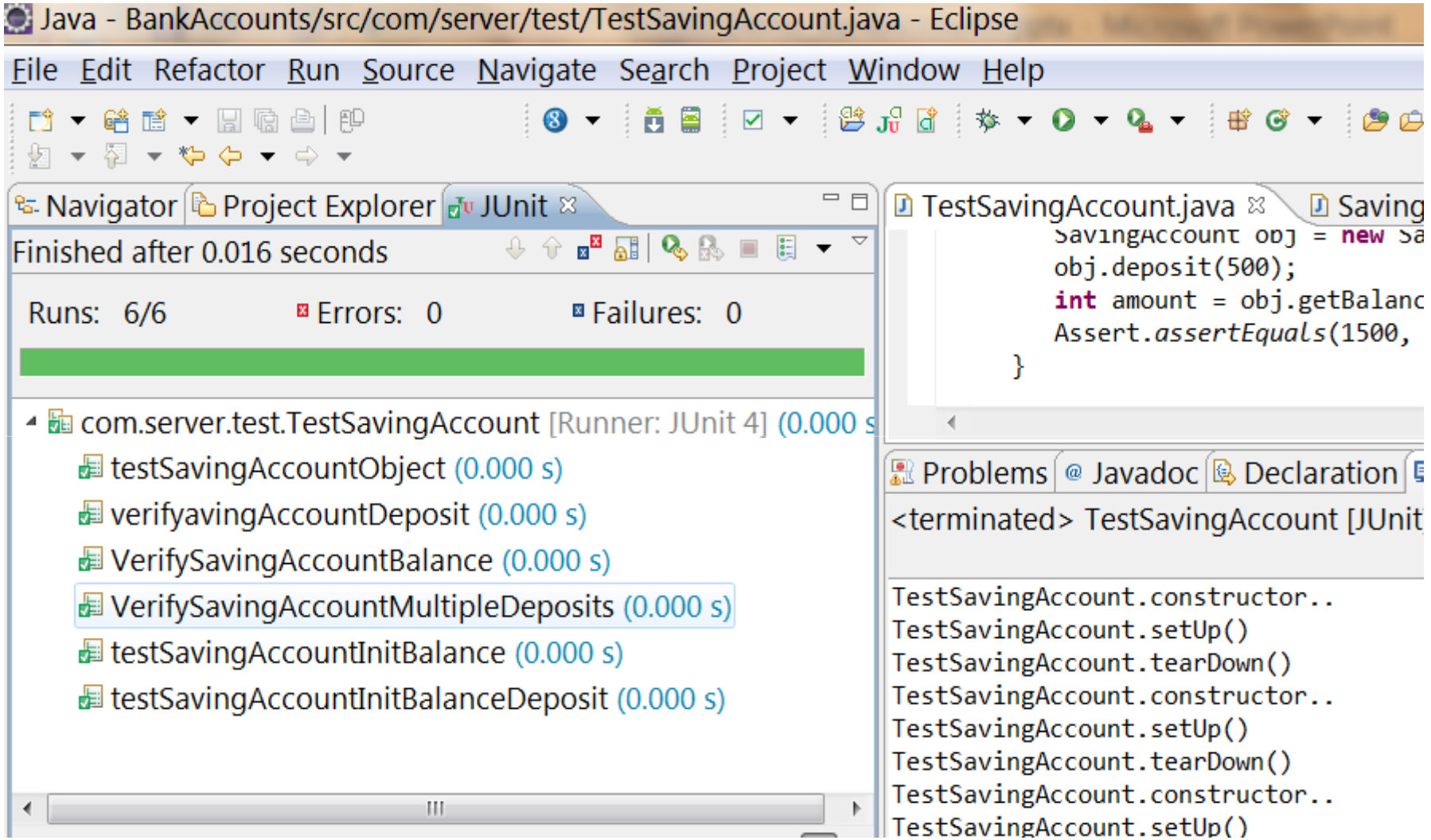
Test objects in JUnit Framework

- For every test case methods the framework has created a new instance of the test class..
- This is the default behavior..

Common shared methods for every test ?


```
public class TestSavingAccount  {
private SavingAccount obj;

@Before
public  void InitObjects() throws Exception {
System.out.println("TestAccount.setUp()");
obj =  new SavingAccount();
}

@After
public void closeObjects()  {
System.out.println("TestAccount.tearDown()");
obj = null;
}
```

Run All the tests..



All the tests in testSavingAccount succeeded..

Next User Story

- As a user I should be able to execute following operations on the LoanAccount
- deposit- the amount deposited should get subtracted from current balance.
- getBalance – the amount returned reflects the current LoanAccount balance.
- Withdraw - the amount to withdraw should get subtracted from current balance.

Evolve the logic for LoanAccount

- Define another Test class TestLoanAccount with test functions one by one and get evolved the functions for LoanAccount to succeed the TestLoanAccount test functions same way as earlier done for SavingAccount…
  - Write the test…
  - First time fake the functionality and run the test and observe GREEN Bar!
  - Write the obvious implementation code to replace fake code and run the test and observe GREEN Bar!

New Project_Refactored

- Create new project BankAccounts_Refactored to define new versions of Loan and Saving Account functionalities.
- Add the SavingAccount and LoanAccount classes along with their corresponding test classes here in new project.

Account restructuring

Now We have two Account classes

- SavingAccount
- LoanAccount

- We have duplicate code scattered across both the above classes.
- To minimize the this code duplication we need to define a class hierarchy through which common code can be shared.

Evolve Account

- Extract/Design the class Account as base class for both
- To standardize the common functions we push the deposit and withdraw functions as abstract in base class.
- Push the getBalance implementation in base class Account.
- Push the balance variable as protected in base class Account.
- Push the ValidateAccountBalance function as protected to base class Account.

Class Account


```
public abstract class  Account
 {
     protected int balance;
     public abstract void deposit(int amount);
     public abstract void withdraw(int amount);

     public int getBalance()
     {
         ....
     }
     protected boolean ValidateAccountBalance()
     {
         ......
     }
 }
}
```

Account Functions


```
public int getBalance()
   {
      //if (balance < 0)
      //    balance = 0;
      if (!ValidateAccountBalance())
          return 0;
      return balance;
   }
   protected boolean ValidateAccountBalance()
   {
      if (balance < 0)
          return false;
      else return true;
   }
```

Restructured SavingAccount


```
public class SavingAccount  extends Account
{  //parameterized constructor
    public SavingAccount(int amount)
    { ....    }
    //default constructor
    public SavingAccount()
    {       ....      }
    public override void deposit(int amount)
    {   .............       }
    public override void withdraw(int amount)
    {   ..............      }
}
}
```

LoanAccount-deposit

```
public override void deposit(int amount)
      {

         if (!ValidateAccountBalance())
            return;

         if (amount < 0)
            throw new NegativeDepositException();

         balance -= amount;
      }
```

New Behavior

- In this application we are allowing the users to create the objects of SavingAccount and LoanAccount classes and calling their individual functions.(e.g. in TestSavingAccount and TestLoanAccount classes).
- We want to isolate the object creation and usage behaviors so that objects can be managed independent of usage and functions used independent of creation process.

New Project

- Define Java project Factory_BankApp as windows Forms application
- Add LoanAccount and SavingAccount classes from previous project with their dependencies except the test classes to this new project.

Factory to manage objects

- To isolate the creation and usage of objects we decide to use a factory class to manage the Account objects.
- To specify the type of object to be created by the factory we define ‘enum AccountType ‘ to specify possible types.
- The factory will be responsible for creating the specific type of Account and sharing it to clients.

Test the Factory

- Add test class TestAccountFactory to test the and build/evolve the functionality of AccountFactory class.
- Add test function TestGetLoanAccountObject to test the LoanAccount type object returned from AccountFactory.

TestGetLoanAccountObject


```
public class TestAccountFactory
  {
     @Test
     public void TestGetLoanAccountObject()
     {
        Account loanAccountObject=
                    AccountFactory.getAccountObject(Loan);
       //Compiler error ..Fix it
       Assert.IsNotNull(loanAccountObject);
     }
  }
```

AccountFactory class

- Define class AccountFactory with static function as ‘getAccountObject(AccountType mode)’
- Fake the implementation right now with returning the NULL.

The factory and the enum AccountType


```
  public enum AccountType
    {
        Saving,
        Loan
    }
}

    public class TestAccountFactory
    {

    }
}
```

The factory ::getAccountObject


```
Public Account   getAccountObject(AccountType mode)
{
   return NULL;
}
```

AccountFactory getAccountObject


```
public static  Account getAccountObject(AccountType acType)
      {
         Account obj = null;

         switch (acType)
         {
        case AccountType.Loan: obj = new LoanAccount(1000);
                break;
         case AccountType.Saving: obj = new SavingAccount();
                break;
         }
         return obj;
      }
```

Added code to return corresponding Account type object

Test for SavingAccount object

- Add test function TestGetSavingAccountObject to test the SavingAccount type object returned from AccountFactory.

TestGetSavingAccountObject


```
@Test
 public void TestGetSavingAccountObject()
 {
    Account savingAccountObject =
        AccountFactory.getAccountObject(AccountType.Saving);

    Assert.IsNotNull(savingAccountObject);

System.out.println(savingAccountObject.GetType().FullName);

 }
```

LoanAccount deposit Test


```
@Test
public void TestLoanAccountDeposit()
{
    Account loanAccountObject =
AccountFactory.getAccountObject(AccountType.Loan);

    loanAccountObject.deposit(500);
    int balAmount = loanAccountObject.getBalance();
    Assert.AreEqual(500, balAmount);
}
```

SavingAccount deposit test


```
@Test
 public void TestSavingAccountDeposit()
 {
    Account savingAccountObject =
AccountFactory.getAccountObject(AccountType.Saving);

    savingAccountObject.deposit(500);
    int balAmount = savingAccountObject.getBalance();
    Assert.AreEqual(500, balAmount);
 }
```

Kudos to TDD and refactoring !!

Thank You!