# 8.1 - Describe the properties of a good unit test

## Why is testing important?

<https://www.sundoginteractive.com/blog/top-ten-most-infamous-software-bugs-of-all-time>

<http://listverse.com/2012/12/24/10-seriously-epic-computer-software-bugs/> (Especially #7)

## Unit testing

Unit tests are a form of **white box** testing. White box tests are tests performed the knowledge of the software implementation. A “unit” is small, isolated part of the application. Typically a “unit” is one method, or at most one individual class or instance of a class. The goal of unit testing is to isolate part of the program and verify that each small part of the program works correctly. This is done at the time that piece of software is written. The important benefit of unit testing is that errors in the implementation are found earlier. The earlier in the software lifecycle that a bug is found, the cheaper it is to fix.

The unit test is designed with the knowledge of the parameters of the method, the values returned by the method and behaviour of the method. It is a good idea to design the unit test at the time the software design is done for the method. Defining unit tests also helps with software design of the method since it requires thinking about all necessary functions of that method.

Unit tests are usually performed with a test driver. A test driver is a program whose sole purpose is to test other software. The test driver will be responsible for exercising the functionality in the code, method by method. Test drivers can typically be set up to run automatically. If test drivers are created as the software is implemented the tests drivers can be continually run as automated tests as other software development continues. It is important to run these tests to ensure new software does not adversely affect software that was already written. Rerunning tests to ensure existing functionality still works is referred to as *regression testing*. JUnit is an example of a tool used to perform unit tests.

## Properties of good unit tests

Well written unit tests will generally lead to more modular software; software components that are easy to write tests for are generally easier to modify.

Unit tests should be:

* **Isolated**. The outcome or internal behaviour of one unit test should not affect other unit tests. The success or failure of a previous unit test should not affect the success or behaviour of a subsequent test, unless both have failed due to a bug or changing requirement.
* **Specific**. Each unit test should focus on as few methods as possible, preferably a single method. The more methods or objects are involved in a test, the more difficult it will be to create an isolated test. Each test should be covering at most a single, small requirement.
* **Small**. Unit tests should be at most 10 to 20 lines of code each. Creating small unit tests avoids testing more than one specific requirement in each test.
* **Descriptive**. Each unit test should be named in a way that concisely describes the content of the test. A common method is to use the "TestThat" syntax for naming test methods; a test to ensure that calling the *incrementPoints* method of a Customer object results in the Customer's frequent flier points being increased by the correct amount might be named "TestThatIncrementPointsAddsTenPoints". Test driver software will often include the name of any failing test in the output, making descriptive test names essential in tracking down the failure point.
* **Clear**. Each test should have clear expectations of the method or unit it is running. Since unit tests are white-box tests, developers should know exactly what outcome will result from certain input provided while the test is running.
* **Fast**. Unit test suites should be as fast as possible, to encourage developers to run them often. Slow unit tests are a sign that the tests are not testing individual methods or units, but instead are testing the integrated system.
* **Repeatable**. Unit tests are intended to be run many times throughout the life of your project. Any unit tests you write must be able to be reliably run again and again, returning the same results every time, unless some bug has been introduced in the code they are testing, or some requirement has changed and made the tests obsolete.

# 

# 

# 8.2 - Describe the differences between unit and integration tests

As described in the previous learning step, unit tests are intended to assure that a single, small unit of functionality, either a single method or at most a class, is operating as the requirements leading to its creation dictate. Unit tests should be isolated. That is, the code they are exercising (running) should depend on as few external classes as possible. The classes it does depend on should be controllable by the unit tests themselves, allowing the unit tests to replace any outside classes being used with implementations used specifically for the test.

Integration tests are tests which cover a comparatively large amount of functionality within the application, or a larger number of classes or modules. They require that at least some portion of the application be configured and able to run as it normally would. As their name implies, integration tests check that the smaller units of the application are able to work correctly when combined together as they would be when the application is actually being used.

|  |  |
| --- | --- |
| **Unit Test** | **Integration Test** |
| White box test - developed with knowledge of how the code functions, provides input and checks output at code level | White or black box test - sometimes developed or run with knowledge of how the system is implemented, but often exercises and checks the operation of the system from a user's perspective |
| Uncovers issues with the implementation of a single module, class or method | Uncovers issues that arise when many modules are used together |
| Written in code, using a test driver such as JUnit | May be written in code using a user interface automation tool to simulate mouse clicks or user interaction, but more often written as a script of user input and expected output |
| Carried out while classes and methods are being written | Carried out after a portion of the system or major feature has been implemented |
| Makes use of very few outside classes, and where they are used, they are instantiated and controlled by the unit test method | Combines many classes and modules together to test their interactions |
| Isolated - does not make use of outside systems such as permanent storage or a database | Often makes use of test-specific storage systems or databases. May check for existence of rows or modification to data in permanent storage as part of test results checking |
| Fast and automated | While it may be automated, integration tests are usually much slower, due to the amount of setup work involved |
| Large number of unit tests written during the implementation of a single feature or user story. Many written for each method or class involved | Only a few likely to be written for each story or feature, covering high level requirements |
| Smallest test written during a software project | Tests code and features for which unit tests have already been created |

The business-level acceptance tests created as part of a user story are forms of integration tests, since they are often written from the perspective of a user interacting with the system. While most acceptance tests will at some point be encoded as a series of unit tests, an integration test will ensure that the feature implemented by the user story works as part of the system as a whole.

## Types of integration tests

### Big Bang Testing

Big bang testing (also sometimes known as top level testing) is probably the most commonly used form of integration testing. In this form of testing, the entire system is intended to be assembled and running in an environment that closely resembles its intended production environment. The system is interacted with as if the tester were a real end user. Test results may be checked through more interaction with the system (creating a user account allows that newly created user to log in), or by checking back-end storage. Big bang testing is generally performed near the end of a software development cycle, and will usually consist of a large battery of individual acceptance tests being run. Because they are a high level test, they are better for detecting errors in workflow or process, rather than small errors such as "off-by-one" errors or boundary conditions in calculations.

### Bottom-Up Testing

In bottom-up testing, the smallest modules are tested in isolation first, then larger and larger combinations of modules are tested at higher and higher levels. Testing the smaller modules first can be beneficial, since it prevents errors at that level from causing issues with more lengthy or resource intensive tests at a higher level. For example, a low level test to ensure that a newly created user has their password saved in a correctly encrypted form could catch errors which could later prevent them from being able to successfully log in and perform a more complicated task. Unit testing is often included under the idea of bottom-up testing. Developers write large numbers of small tests for small methods and classes, and later make use of those thoroughly tested methods and classes in other, higher level classes.

# 

# 8.3 - Construct a test plan

## Planning tests

Each functional requirement in the software project must have a complete set of unit tests developed for it, as well as acceptance and integration tests for the entire system. The definition of "complete" depends greatly on the complexity of the functionality as well as how critical it is to the correct functioning of the system.

Before writing tests, it is often a good idea to plan out the basic set of conditions under which the unit of functionality should be tested. Creating a set of data points for each of the following can assist in later creation of individual unit tests

* **Preconditions**
  + What other objects or configuration must be in place before the class or method under test can be used or exercised
  + What range of values for external variables used by the method being tested are allowable
  + Can external objects used by the method under test be replaced with placeholders or null for the purposes of the test
* **Input data**
  + What possible values could the method under test receive for each in each of its parameters
  + Unit tests should be planned for each of the following classes of input data
    - **Normal cases.** Typical functioning of the system. For example, providing a birth year within the last 120 years to a method which verifies that a user is old enough to access a system
    - **Exception cases.** Error conditions that may occur. Need to ensure the system is robust enough to handle errors without behaving abnormally. For example, providing 0 or 13 to a method which converts integers into textual months of the year
    - **Boundary cases.** It is important to test values right at the edge of the valid condition. Quite often software errors will occur right around boundaries of valid/not valid values. For example, providing 1 or 12 to a method which converts integers into textual months of the year
* **Post conditions and expected results**
  + What other objects or variables should have been affected by providing a certain set of input to the method under test
  + What value should have been returned from the method under test given the set of input data it was provided
  + Post conditions and expected results should be plotted for normal, exception and boundary cases, to prove that the method behaves as expected in each

You may have noticed that unit testing plans are very similar to the test plans and documentation created for acceptance tests in learning outcome 4, step 7. The two grids created are very similar, and some teams which create both may have them be identical, only differing in the level of detail used.

Acceptance tests are integration tests, and therefore exercise the system at a much higher level than a unit test would. Where a unit test would call a method with specific parameters and expect specific values as results, an acceptance test would define specific user interface screens and controls to use, and possibly inspect another screen or the database to check results. Even though the two types operate at very different levels, the overall documentation for them may seem similar. However, unit test documentation is less likely to be recorded in a permanent form. While a team would likely keep their acceptance test grid to be used and re-used many times, the team might create their acceptance test grid on a whiteboard, using it to drive the creation of their unit test code, then discard it once the code is complete.

## Planning Tests Practice

Following are two method signatures and descriptions of their expected functionality. For each, write down the elements of a test plan you could use to guide the creation of unit tests for that method. Include the following for each, where applicable:

* **Preconditions** - what objects or outside variables need to exists, and in what state, before testing can begin. The "state of the world" before the method under test is called
* **Input data** - for each of the different types of test cases (normal, exception and boundary), write down possible values for each of the parameters to the method under test which would fit in that type of test case.
* **Post conditions / expected results** - what value or result should the method return, and what should the state of external objects be, once the test has concluded.

Example:

* **boolean isValidAge(int age)**

Takes in an age. Should return true if the age is between 18 and 130 inclusive, false otherwise

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class (normal, exception, boundary)** | **Preconditions** | **Input data** | **Post conditions** | **Expected result** |
| normal |  | age: 20 |  | returns true |
| boundary |  | age: 18 |  | returns true |
| boundary |  | age: 17 |  | returns false |
| exception |  | age: -1 |  | returns false |

Complete the grid located below under the heading Unit Test Plan Practice – Solution for both methods.

### Method 1

* **int calculateRectangleArea( int height, int width )**

Takes in the width and height of a rectangle, and returns the area it covers. Width and height should be positive integers. If the parameters passed in are less than or equal to zero, then zero should be returned.

### Method 2

* **boolean createNewAccount( int accountId, float beginningBalance, String name )**

Account ids must be in the range 10000 – 20000. This method takes parameter values and sets up a new bank account for the person. The method is present in a BankAccount class having a database connection which is used to save the new account.

## Unit test plan practice - solution

Below is a possible set of test plan data which could be used to guide construction of unit tests for the given methods. Note that they may not be complete: you would likely discover more possible test cases as time went on.

* **int calculateRectangleArea( int height, int width )**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class (normal, exception, boundary)** | **Preconditions** | **Input data** | **Post conditions** | **Expected result** |
| normal |  | height:5, width:5 |  | returns 25 |
| boundary |  | height:1, width:1 |  | returns 1 |
| boundary |  | height:0, width:0 |  | returns 0 |
| exception |  | height:-1, width:5 |  | returns 0 |
| exception |  | height:-5, width:-2 |  | returns 0 |
| exception |  | height:100, width:INT\_MAX |  | This may not be defined behaviour. Since this would result in a nonsense return value, you may need to discuss the required behaviour in this case with your client. |

* **boolean createNewAccount( int accountId, float beginningBalance, String name )**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Class (normal, exception, boundary)** | **Preconditions** | **Input data** | **Post conditions** | **Expected result** |
| normal | database connection is established | accountId:15000, balance: 500, name "Jim Jones" | Database connection still open. New account record present in database | returns true |
| exception | database connection is down | accountId:15000, balance: 500, name "Jim Jones" | You would have to confirm behaviour with your client and team. It's possible that they may want an exception to be thrown in this case | returns false |
| boundary | database connection is established | accountId:9999, balance: 500, name: "Jim Jones" | Database connection still open. No account record created | return false |
| boundary | database connection is established | accountId:10000, balance: 500, name: "Jim Jones" | Database connection still open. Account record created | return true |
| exception | database connection is established | accountId:5000, balance: 500, name "Jim Jones" | Database connection still open. Account record is not created | Return false |

While further test plan items could be created at this point, from the information given you would likely be guessing as to correct functionality for all given parameters. At this point, it would be a good idea to consult your project documentation, existing code and standards, or with your team and client to discover the correct set of behaviour for the requirement your method is implementing.

# 

# 

# 8.4 - Identify testable methods of software construction

## Software development principles

In essence, more modular, loosely coupled software will almost always be easier to test. The fewer other methods or objects a certain method depends on, the simpler it will be to test it in isolation. Methods which can be more easily isolated will be easier to test. If a method depends on several other objects being present or configured in certain ways, the test will require more comprehensive setup, as well as more complex verification of the state of those objects when the method under test has completed.

Recall the principles of object oriented design discussed in learning outcome 9:

**Single Responsibility Principle**

* Classes should have only one reason to change. In other words, classes or methods should implement as few requirements as possible. When a class has multiple roles or units of functionality, it is likely to be making use of a larger number of other classes, making it more difficult to test.

**Separation of Concerns**

* Software projects should be constructed as separate layers of functionality. Each layer should be concerned with a specific set of tasks. Classes in the user interface layer should be concerned with presenting data to and receiving input from the user, and should use well-defined interfaces when communicating with other layers of the software project. When this principle is violated, user interface code, business logic and data storage code tend to end up closely coupled within the same method or class. When this occurs, testing becomes difficult, as more outside classes and conditions must be set up before an individual test can be run.

**Tell, Don't Ask**

* Code which uses encapsulation well is generally easier to test. When encapsulation fails, methods and classes tend to be fetching data from other classes in order to perform work that relates to that other class. While it may still be possible to write unit tests for this work by providing properly configured data and objects as part of test preconditions, doing so can become much more onerous. Keeping data and related behaviour together within a class and having other classes request that the class do the work itself, rather than fetching data from it, can result in tests which are easier to set up.

**Dependency Inversion Principle**

* The dependency inversion principle states that high level classes should not depend on low level classes, but instead should depend on high level descriptions (interfaces) of what those classes will do. Code which follows the dependency inversion principle tends to not create any other classes it needs on its own, as classes which can be directly instantiated are usually concrete or low level. Instead, their methods or constructors will require an instance of a class implementing a higher level interface which describes the behaviour they need. When classes do this, it becomes easier to substitute their dependencies at test time with placeholder implementations of those interfaces, which behave in ways that can be well controlled by the test.

## Practice - identify testable code

Below are 3 code snippets. For each, identify whether you think the method would be simple or difficult to test. Include your reasoning, possibly quoting one of the software development principles from LO9 which is being violated.

1. Calculate Rectangle Area

|  |
| --- |
| public int calculateRectangleArea( int width, int height ) {  int area = 0;  if( width >= 0 && height >= 0 )  {  area = width \* height;  }  return area;  } |

1. Create account

|  |
| --- |
| public boolean createNewAccount(int id, double balance, String name)  {  boolean result = false;  if(this.isValidId(id) && this.isValidBalance(balance) && this.isValidName(name))  {  DBConnection connection = new DBConnection("username", "password", "database");  if(connection.isActive)  {  result = connection.createAccountRecord(id, balance, name);  }  }  return result;  } |

1. Calculate area of a circle ( in class named JavaFXDrawScreen )

|  |
| --- |
| public double areaOf( Circle aCircle ) {  double radius = aCircle.diameter / 2.0;  double area = Math.PI \* (radius \* radius);  return area;  } |

## Solution - identify testable code

* This method would be very easy to test. It is very short, performs few operations and has well defined inputs and outputs. It does not have any dependencies at all, as it does not use any outside objects or instantiate any classes.
* This method, while it does have some good qualities, would be difficult to test. It instantiates a database connection every time it is executed, which would make it more difficult to check that a record had actually been created. The test would also likely be slow, since it takes time to start up and close down a database connection. However, having the three separate "*isValidX*" methods aids in the testability of this method, since they could each be tested and proven in isolation. If the developer were to have the class containing this method store the database connection as a property, and have this method use that property instead, it would become much easier to test in isolation.
* This method would likely be easy to test individually, but a developer writing this test should be wary of the method violating the "Tell, Don't Ask" principle. This method seems to be implementing behaviour that should probably actually be implemented within the Circle class itself. In fact, a test for the "areaOf" method on the Circle class might look very much like a test for this method as it is written. However, since the method is outside of Circle, it's possible that code similar to this method may have been or will be implemented in several other locations, leading to duplicated code, unit tests and more possibility for error.

# 

# 

# 8.5 - Describe test driven development

Extreme Programming(XP), a component of the agile methodology, emphasizes writing unit tests before the code is written. This is called Test Driven Development(TDD). Writing the unit tests before the code is a good idea, but XP takes this to the extreme and states that unit tests should be written and run before any code is written.

A unit test is coded into a test driver, such as JUnit.The test driver is run to verify that the new test fails. Then, just enough code is written to get the new and existing test cases to pass. Next, a new test case is written in the test driver to test the next small section of the requirement being implemented. When this new test is run, it should fail since the code to implement it has not yet been implemented. Enough code is written to pass the new failing test, and the cycle continues.

The style of test driven development described above is sometimes called "red/green" unit testing. Most test driver software, including JUnit, colour codes test results into either "green", meaning all tests are passing, or "red", meaning at least one test is failing. While it can be practiced by an individual, red/green testing is often used with pair programming. One developer will write a test for the next part of the requirement being implemented, then run it to prove that it fails (red). The other developer then writes just enough code to pass the test (green), then writes a new test for the next part of the feature. Since the code to implement this part has not yet been written, the test will fail (red). The first developer then takes over, writing enough code to pass the test, then writes a new failing test. While it may seem inefficient, this method of test driven development can help to keep both developers highly involved in the task while pair programming, and keeps a dialog going between them.

Test driven development is often used as a tool of software design. TDD, especially the red/green style, tends to lead developers to write tests which cover very small portions of the code to be implemented. In order to make writing the many tests easier, they will tend to put more effort into separating their code into many smaller modules, be they separate classes or methods. Modular code is the goal of any software project, as modular code tends to be simpler and less costly to modify. Modular code depends on fewer outside classes or data, and is simpler to set up and interact with in an isolated unit test. Thus, code which is written in a way that is simple to unit test should be modular and amenable to change.

An example of test-driven development is covered in the next learning step.

### Further reading:

<http://www.codeproject.com/Articles/47747/Test-Driven-Development-TDD-Step-by-Step-Part-In>

# 8.6 - Construct Unit Tests

In learning outcome 7, you worked through an example of constructing a set of classes to represent a transactional banking system, which followed good object-oriented design principles and practices. In this example, you will construct a set of unit tests to cover the functionality already implemented by this system, then use test driven development to implement additional functionality.

Download and unzip the file UnitTests\_Initial.zip. This contains the starting files from the transactional banking example in an Eclipse project. Open the project in Eclipse.

1. New -> Java Project-> Project Name: UnitTests -> Use jdk-15.0.1 (on my laptop) -> uncheck “Create module-info.java” on the second dialog box.
2. Drag .java source files (from File Explorer) under src.
3. Select Copy files.

## Unit testing setup

It is not uncommon in software development for a developer to be **required** to create unit tests for existing code. Not all developers subscribe to the idea of test driven development, or even unit tests at all, and a large amount of existing code exists which is covered simply by acceptance or integration tests, or by no tests at all.

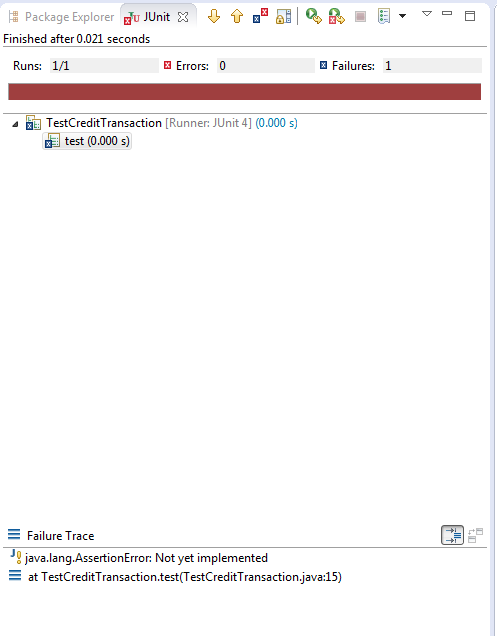
First, we will construct tests for the existing body of code in the sample project. With the project open:

* + - 1. Create a new "JUnit Test Case" alongside the existing Java files.
      2. Name it "TestCreditTransaction".
* NB: An animation of this process is shown in CreateTestCase.wmv.

Notice that in addition to creating the test file, Eclipse automatically imported the JUnit 5 library. The Junit 5 library is a test automation library commonly used for Java projects, and which is included with most Eclipse installations. The created file also contains a class definition and a single public method. In JUnit 5, test classes are *like any other class*. They are *not* required to extend any special classes in order to be considered a unit test class.

Each test in a JUnit 5 test class is annotated with "@Test", which marks the method as being a unit test case, which should be run by the JUnit test runner. You can run the existing JUnit test the view the results:

* + - 1. Right-click on the TestCreditTransaction file.
      2. Select "Run As".
      3. Select "JUnit Test".
      4. The output should appear as below:



NB: The single test method within the file has been marked as "failed", and the method can be selected in the top dialog in order to see its output in the lower dialog.

## Constructing tests for existing code

We will now construct a set of tests for the CreditTransaction class. It is common for developers to test the smallest, most concrete (lower in the inheritance hierarchy) classes before more abstract classes or the other classes which may make use of them. This allows them to verify the depended-on classes behaviour first, so that they can possibly make use of it in dependent classes.

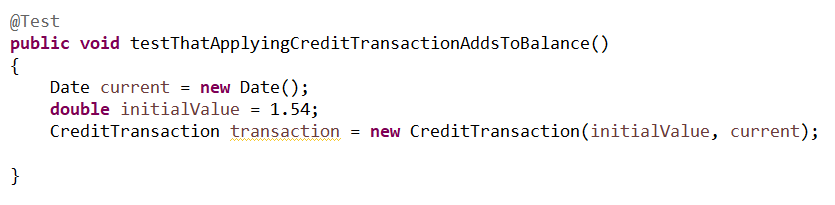
In your TestCreditTransaction class, remove the existing test case, then add a new method, proceeded by the "@Test" annotation.

1. *testThatApplyingCreditTransactionsAddsToBalance()*

Though this method name seems long, the descriptive nature can help to easily identify it and the functionality being tested if it fails at some point in the future.

In the test method, create a new Date object to represent the time the Transaction was posted, then a double value to represent the amount of the transaction. Next, instantiate a CreditTransaction using those two variables.

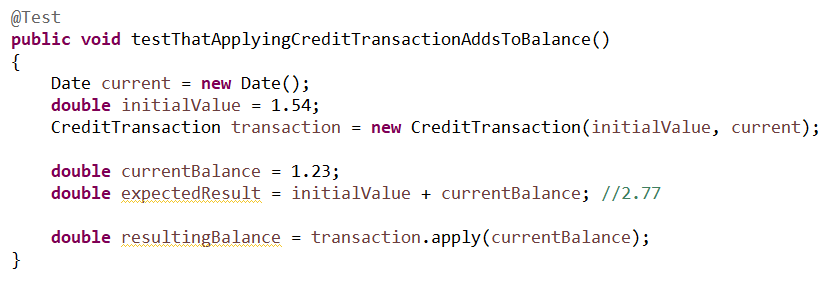
NB: You may have to import this annotation using "import java.util.Date" if your IDE does not perform the import automatically.



Now that an instance of the class under test has been instantiated, it is time to "exercise" its methods. In the context of a unit test, "exercise" means to run the method under test, setting up preconditions, calling the method then verifying the results.

Next we should:

* + - 1. Call the *apply* method, passing in an amount representing the current balance.
      2. Capture the value returned by the *apply* method to another variable.
      3. See below for a completed method:

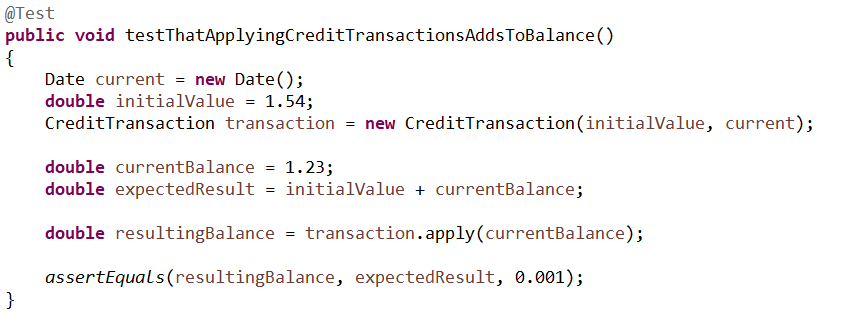


* + - 1. Now to verify the result below the line of code where apply is called, add the following line of code:



NB: You may have to import this annotation using "import org.junit.Assert.assertEquals" if your IDE does not perform the import automatically.

Your end results should look like the following:



JUnit tests consist of setup, such as the instantiation and configuration of objects, and *assertions*, which check the results of calling the method under test. This line of code uses the JUnit assertion *assertEquals*, which will show a successful result in the JUnit test output if the first two parameters match, and an unsuccessful result otherwise. This particular *assertEquals* is comparing floating point values, and so includes a "delta" value, since floating point values often cannot be directly compared. If the two values are less than "delta" apart, they will be considered equal.

As an exercise, create a new JUnit Test class named TestDebitTransaction, then fill in the same test for the DebitTransaction class as was just created for CreditTransaction. Once you are done, run both test cases to ensure they are passing. A completed set of tests for these two classes is included below.

NB: the use of the "@BeforeEach" syntax on the *setUp* method of the two classes. It is common to have a great deal of repeated setup code in each unit test method. The @BeforeEach annotation causes JUnit to run the annotated method once before each test, giving the developer to centralize their object initialization.

TransactionTests.zip

## Test-Driven Code - Requirements

In the next exercise, you will create a set of tests to describe the behaviour of code which has not yet been created. In the transaction management system, you will create tests for a new type of Transaction, the InterestTransaction. Though you may have created this class as part of the previous learning outcome, you will create a new, test-driven version here.

InterestTransaction is intended to model the concept of calculating and applying a payment of interest to an account balance. When created, it should be provided with four parameters:

1. The amount marking the boundary between low and high balances
2. The date on which the transaction occurred.
3. The interest rate to apply for low balances.
4. The interest rate to apply for high balances.

When the *apply* method is called, the interest transaction should decide which interest rate to apply, based on the boundary amount passed into the constructor and the balance passed in to *apply*, then calculate and apply that amount of interest to the balance. If the balance passed into the *apply* method is less than or equal to 0, the InterestTransaction should return the balance unmodified.

### Test-Driven Code - First Test

We will use TDD to test and implement each individual requirement, one at a time. The steps below

* + - 1. Create a new JUnit Test Case named "TestInterestTransaction".
      2. Remove the automatically created test method.
      3. Add a new method named "public void setUp()", annotated with the "@BeforeEach" JUnit annotation.

NB: You may have to import this annotation using "import org.junit.jupiter.api.BeforeEach;" if your IDE does not perform the import automatically.

* + - 1. Create an attribute on the test class to hold an instance of the InterestTransaction.
      2. Create attributes named "highBalanceInterestRate" with value of 0.012 and "lowBalanceInterestRate" with value of 0.006.

Graphical user interface, text, application

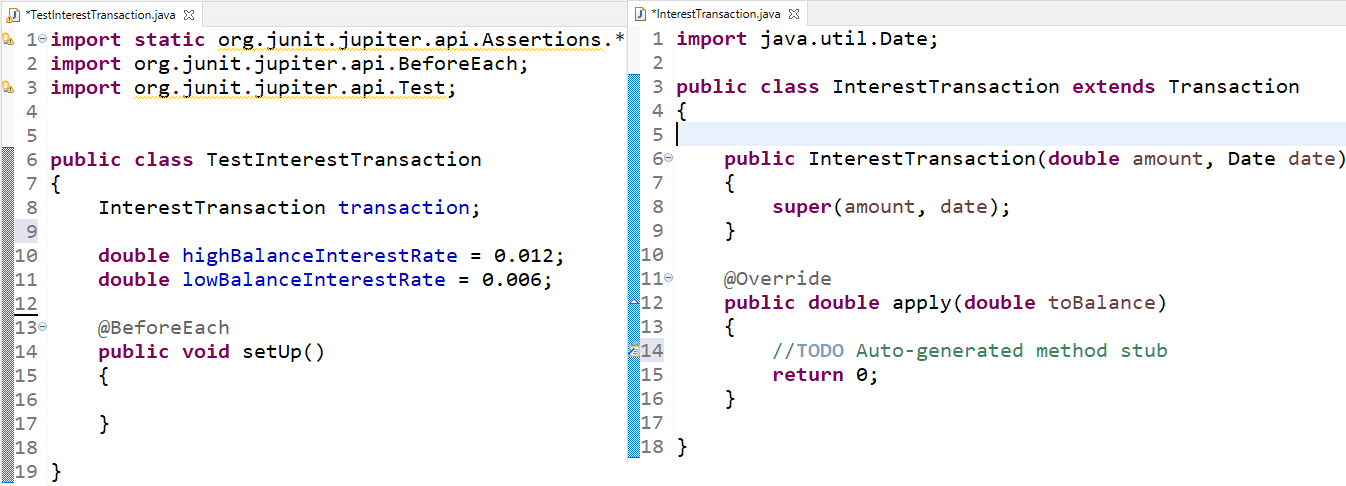
Description automatically generated

At this point, we have already created our first failing test. When we attempted to create an attribute to hold an instance of InterestTransaction, we created a compile error, since the InterestTransaction class does not yet exist. Test driven development considers a failure to compile due to use of an as-yet uncreated method or class, to be a failing test. If you attempt to run TestInterestTransaction, it will fail due to this missing class.

To pass the first test:

1. Create the InterestTransaction class as a child class of Transaction.
2. Create an empty implementation which simply returns 0 in order to allow the class and test to be compiled since the apply method is marked as abstract in Transaction.

At this point, the two files should appear as below; this is the minimum amount of code required to pass the first failing test.

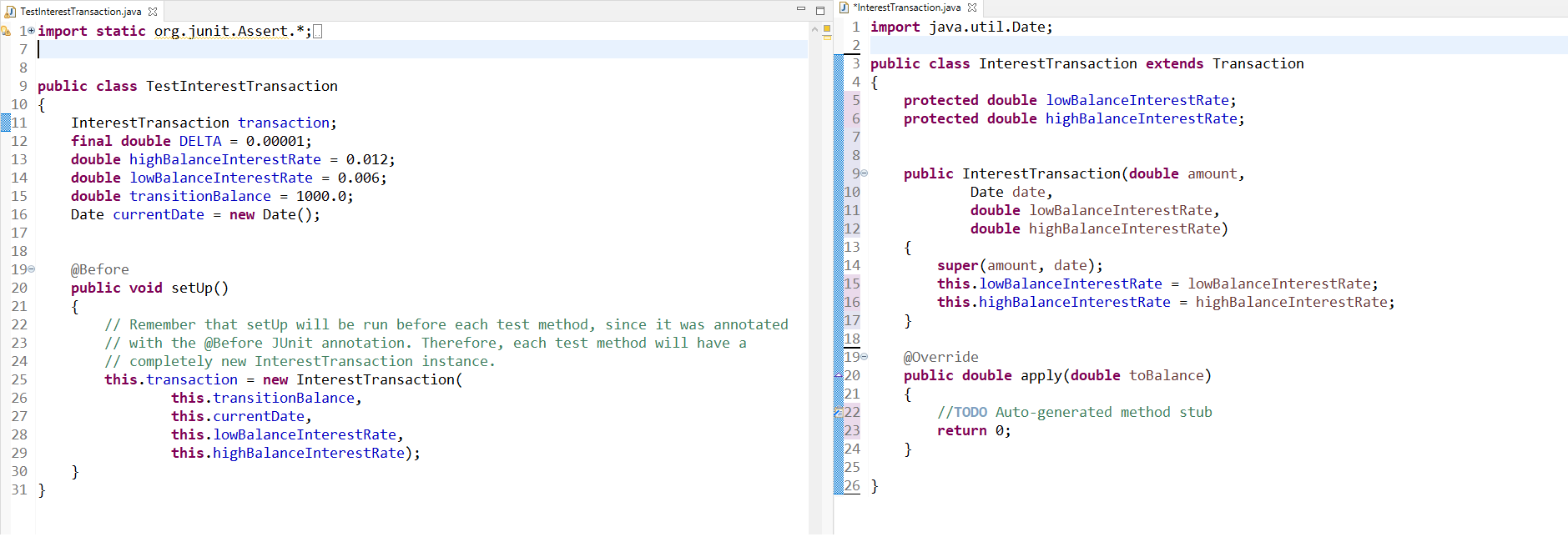


### Test-driven code - First requirement

The first major requirement stated in the class description was that when the *apply* method is called, the interest transaction should decide which interest rate to apply – if the balance passed into the *apply* method is less than or equal to 0, then InterestTransaction should return the balance unmodified. This is the first requirement which must be tested.

* + - 1. Modify the constructor of InterestTransaction to take in the 4 parameters as described (amount, date, lowInterestRate and highInterestRate).
      2. Add attributes for the lowBalanceInterestRate and highBalanceInterestRate to the InterestTransaction class.
      3. In *setUp* call the parent constructor with the correct values to fix the compilation errors.

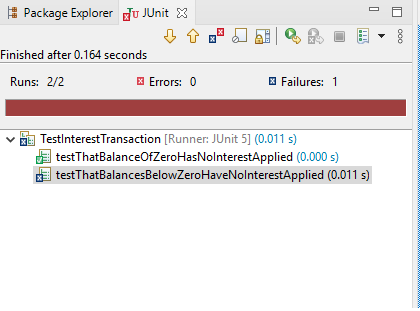
The code should appear as below:



Now we are ready to start creating our tests.

* + - 1. Create a new test method in TestInterestTransaction, naming it *testThatBalancesBelowZeroHaveNoInterestApplied*. In this method you need to ensure:
         1. Balances less than zero should be returned unchanged.
         2. Add an assertion which checks the initialBalance against the resultingBalance;

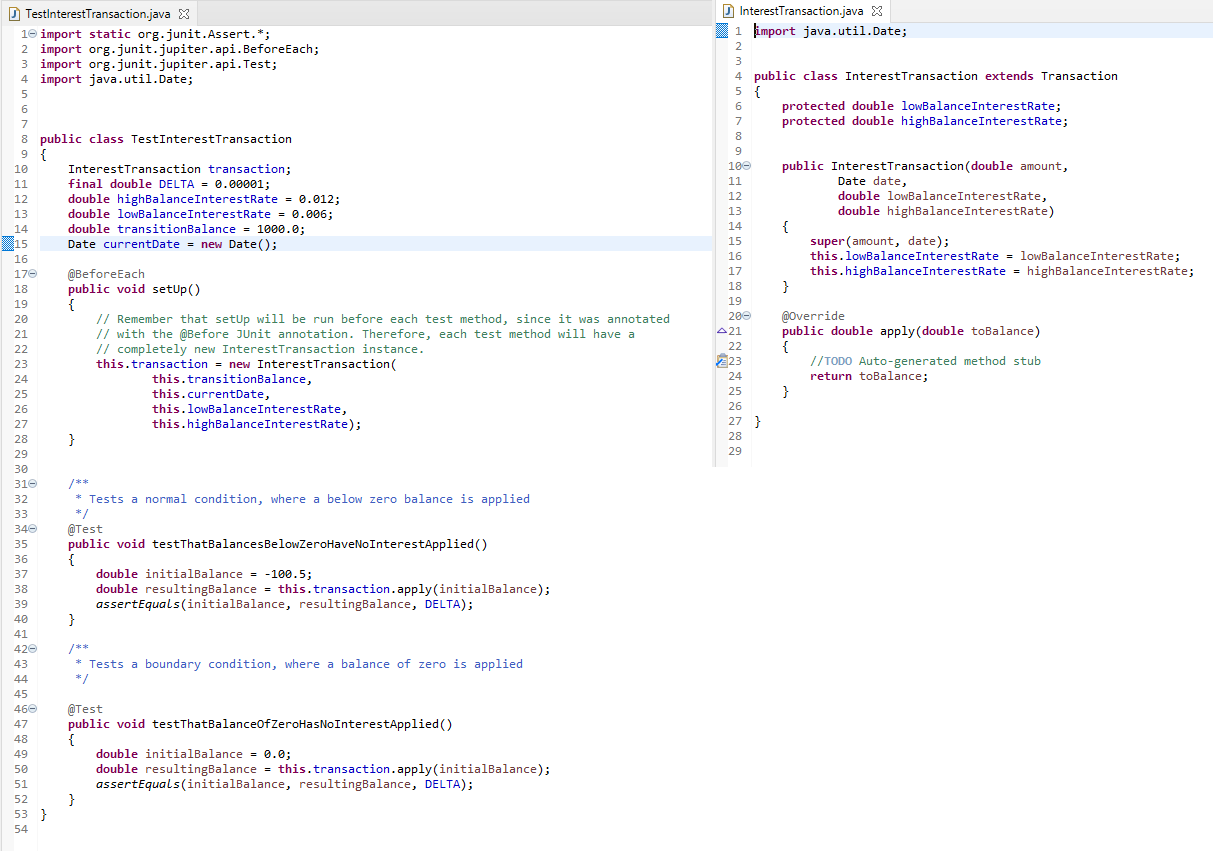
1. Create a new test method in TestInterestTransaction, naming it *testThatBalanceOfZeroHasNoInterestApplied.*
   * + - 1. Balances equal to zero should be returned unchanged.
         2. Add an assertion which checks the initialBalance against the resultingBalance.
2. You can run these JUnit tests by right-clicking on the TestCreditTransaction file, selecting "Run As", then "JUnit Test". The output should appear as below.



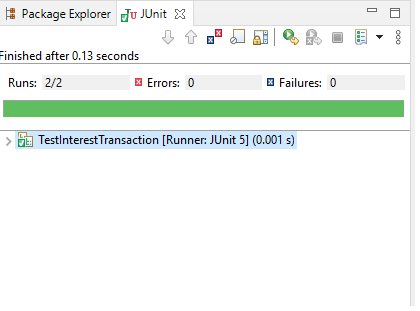
Note that the both test methods within the file has been marked as "failed", and that the methods can be selected in the top dialog in order to see its output in the lower dialog.

Create just enough code to pass these tests as they are now. Modify the *apply* method to return the balance unchanged.

Your code should appear as follows:



Your tests should now pass.



### Test-driven code - Next steps

Now that balances less than or equal to zero have been tested, you can proceed to developing tests and code for the remaining requirements around the *apply* method. For each of the following requirements, as described in the initial documentation, implement a failing test, then just enough code to pass the test. Remember to include tests for boundary conditions as well as normal operation. At the end of this page is a link to a zip file containing the test code and InterestTransaction code as they may have existed at each step. Attempt the exercise yourself before inspecting the completed code.

1. Balances less than the transition amount, but above 0, should have the lower interest rate applied and added to the returned value.
2. Balances greater than or equal to the transition amount should have the higher interest rate applied and added to the returned value.

1\_InterestTests.zip

2\_InterestTests.zip

### Further reading:

<http://www.amazon.ca/dp/0321146530?tag=vig07-20>

# 8.7 - Describe testing coverage

## Testing coverage

Testing coverage is a term that indicates the amount of code that has been tested. This is important during the testing phase as it is necessary to strive to test all of the code.

There are a number of coverage criteria:

* **Function Coverage** - Each function in the system has been executed.
* **Statement Coverage** - Every line of code is executed.
* **Condition Coverage** - Every boolean condition must be tested to both true and false conditions.
* **Path Coverage** - Every possible route through the code is executed.
* **Entry/Exit Coverage** - Every possible call and return of each function is executed.

There is some overlap in the above test coverage criteria. For example, to test every possible route through the code will also provide statement coverage. It is important to note though, that if all paths are covered, it may not be the case that all boolean conditions have been exercised, since there may be compound conditions. It is important to test all boolean conditions.

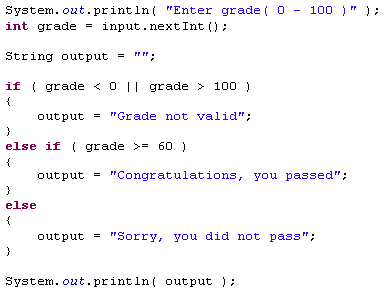
The testing requirements will vary from project to project. For example, safety critical system will require 100% statement coverage. There is no room for error. On less critical systems, a slight lower percentage of coverage will be acceptable. However, it is important to make the system as error-free as possible.

## Code coverage applied

There are numerous code coverage tools available, including some free tools. Code coverage tools will document which code is executed when the program is executed. This makes it possible to create tests that will provide 100% coverage of the code. Testers look at the results from the code coverage tool to determine which code has not been executed, and then determine what additional tests are required to test those lines of code.

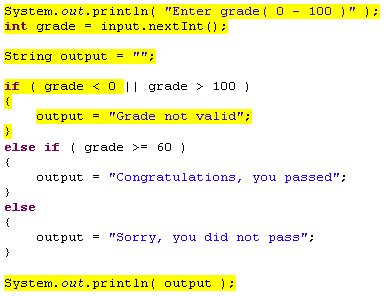
### Example

Consider the following Java code example:

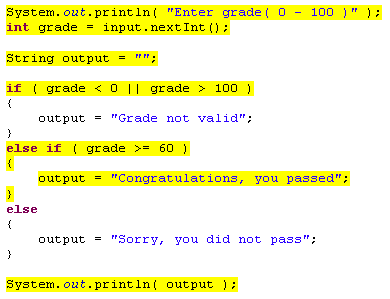


There are three paths through this java code:

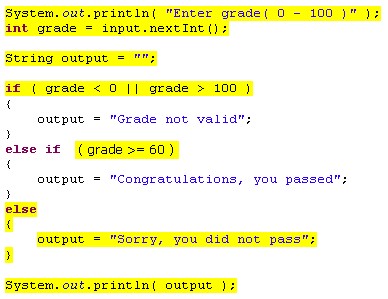
**Path 1: Grade is -1**



**Path 2: Grade is 60**



**Path 3: Grade is 58**

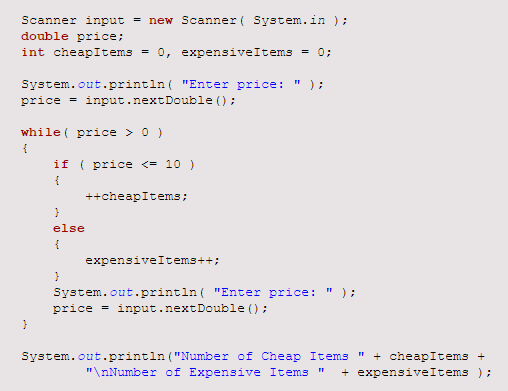


The above three case indicate all the possible paths through this Java code. However, there is one boolean expression that has not been tested. This is why it is necessary to consider condition coverage as well. The boolean expression “grade > 100” has not been tested, so if there was an error in this boolean expression, there would be an error in the program. This is why path coverage is not always sufficient. Each boolean expression should be tested as well. So one more test would be required – grade = 101.

Code coverage tools will be useful in identifying all possible paths through the code, but this is not necessarily the only testing that should be done.

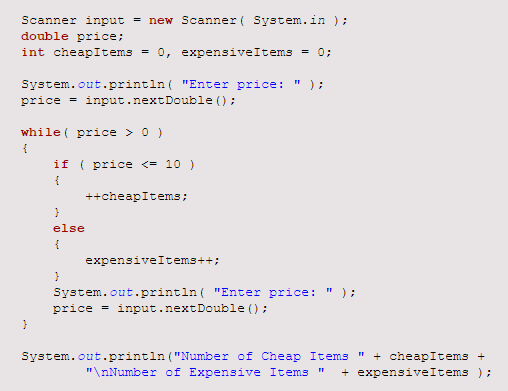
## Identifying paths through code

For the following code snippet, identify all possible paths through the code that should be tested.



Once you have written down all of the possible paths through the code (the set of possible variable values that would result in every line being executed), proceed to the next page

## Identifying paths through code - solution



There are three paths through this code snippet.

* **Path 1:** Price is 0 or less first time through. Loop terminates and code following still works correctly even though loop statements were never executed.
* **Path 2:** Valid price that is greater than 0 but less than or equal to 10. The true portion of the if is executed.
* **Path 3:** Valid price that is greater than 10. The false portion of the if is executed.