**Using Mock objects to test our loan class.**

The process of Unit testing is defined as testing classes or methods in isolation.

Java classes usually depend on other classes. A mock object is a dummy implementation for an interface or a class in which you define the output of certain method calls. Mock objects allow you to unit test the class which should be tested without any dependencies.

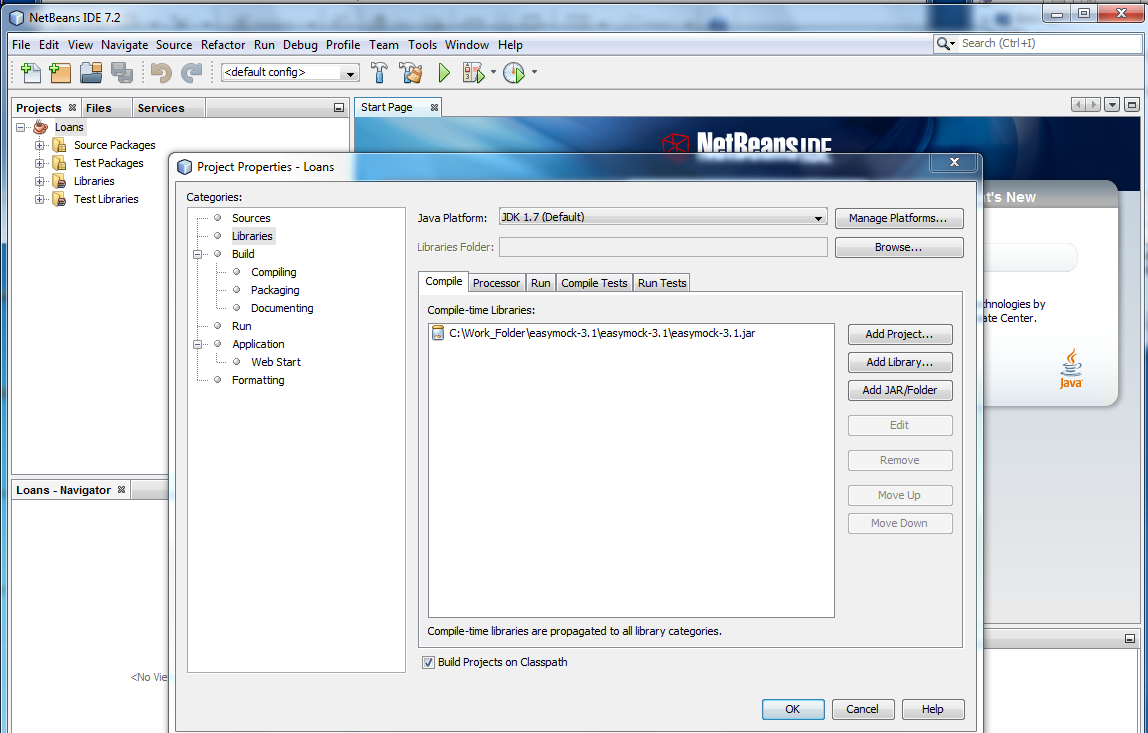
You can create these mock objects manually (via code) or use a mock framework to simulate these classes. Mock frameworks allow you to create mock objects on the fly and define their behaviour.

The classical example for a mock object is a data provider. In production a real database is used but for testing a mock object simulates the database and ensures that the test conditions are always the same.

These mock objects can be provided to the class which is tested. Therefore the class to be tested avoids any hard dependency to external data.

**Mocking Classes and interfaces.**

We will use the **easymock** tool with the powermock addition to create our mock objects. To install easymock take a copy of the powermock-easymock-junit-1.7.1 folder from the x drive and copy it to your own folder. We must add the easymock jar files to our libraries. Right click on the project structure and select Libraries. Then select + and browse to add all the files in the powermock folder.



**Mocking a class constructor.**

Let’s add an Owner class to our Robot project. A robot can be assigned an Owner. This method interacts with the Owner class. In unit testing we must test each class and method in isolation. The Robot method calls the Owner class. We want to test the Robot method without actually calling the Owner class. We are not interested in testing the Owner class right now. It may not even exist at this stage. We are only concerned that the Robot method does what it is supposed to do. This is where mocking classes comes in. We can set up a mock class to mimick the behaviour of the Owner class. This will allow us to test the Robot class in isolation. Easymock alone will not let us mock constructors. We need to add an extension called PowerMock. This allows us to test static methods and constructors. Our Owner class is as follows. Create a new Owner Class in the Robot project and copy the code below.

**public class** Owner {  
 **private** String **name**;  
 **private double monthly\_payments**;  
   
  
 **public** Owner(String name, **double** mpayment){  
 setName(name);  
 setMonthlyPayments(mpayment);  
 }  
 **public void** setName(String name){  
 **this**.**name** = name;  
 }  
   
 **public void** setMonthlyPayments(**double** mpayment){  
 **this**.**monthly\_payments** = mpayment / 12;  
 }  
   
 **public** String getName() {  
 **return name**;  
 }  
  
 **public double** getMonthlyPayments()  
 {  
 **return monthly\_payments**;  
 }  
   
 **public** String toString() {  
 **return "Name: "** + getName() + **" Payments: "** + getMonthlyPayments();  
 }  
  
  
}

Add the following method to the Robot class.

**public double** recordOwnerAndMonthlyPayments(String name) {  
  
 **double** cost = checkCost();  
 System.***out***.println ( name + cost);  
 Owner myOwner = **new** Owner(name, cost);  
 **return** myOwner.getMonthlyPayments();  
}

We will create a new Junit test class. Select the Robot class and press Alt – Enter

Call the file RobotMockTest.

In the RobotMockTest class add the following 2 lines after the import statements.

Because we need to use PowerMock we need to import it and tell PowerMock to run the test Class. This is done using @RunWith and @PrepareForTest.

@RunWith(PowerMockRunner.**class**)  
@PrepareForTest(Robot.**class**)

We are now ready to write our test. The steps for working with mock objects are.

1. **Create Mock** - This phase creates the mock object.
2. **Expect**- This phase records the expected behaviours of the mock object. These will be verified at the end.
3. **Replay** - Replays the previously recorded expectations.
4. **Verify** - In order for a test to pass, the expected behaviours must have been executed. The verify phase confirms the execution of the expected calls.

@RunWith(PowerMockRunner.class)

@PrepareForTest(Robot.class)

public class RobotOwnerTest {

@Test

public void testOwner() throws Exception {

System.out.println("in test");

Robot buddy = new Robot("buddy", 1);

/\*call the createMock to create a mock for the Owner class \*/

Owner mockOwner = EasyMock.createMock(Owner.class);

/\* Tell PowerMock to intercept any new call and return mockOwner \*/

PowerMock.expectNew(Owner.class, "John Smith", 10000.00).andReturn(mockOwner);

/\* set up the expected values and return values \*/

expect(mockOwner.getMonthlyPayments()).andReturn(833.33);

/\*activate the mock \*/

PowerMock.replay(mockOwner, Owner.class);

double expResult = 833.33;

double result = buddy.recordOwnerAndMonthlyPayments("John Smith");

assertEquals(expResult, result, 0.05);

/\* verify that PowerMock was called and used \*/

PowerMock.verify(mockOwner, Owner.class);

}

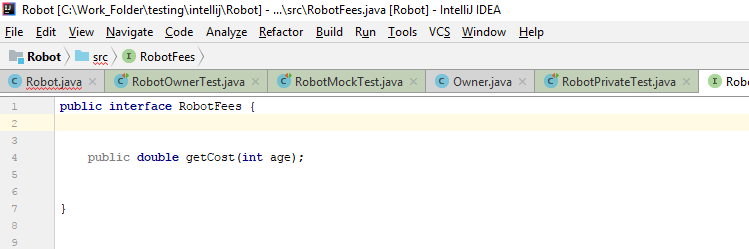
}

**Mocking Interfaces.**

If we add the following method to our Robot class. This method calls the RobotFees to get the cost of maintenance on the Robot per year. This may call a server where the costs are stored in a database. We don’t have the connection set up at the moment but we want to test our method and check that if the connection worked ok and returned a cost our method would handle the result correctly.

*//Returns the total cost***public double** getTotalCost(RobotFees rFees) {  
  
 **double** cost;  
  
 **double** fees;  
  
  
 fees = rFees.getCost(**age**);  
  
 System.***out***.println(**"for "** + **age** + **"the cost is "** + checkCost());  
  
 cost = (checkCost()) + fees;  
  
 **return** cost;  
  
}

We can create an interface that will be passed the Age of the robot and return the cost of handling the yearly maintenance . To do this in the src folder create an new java interface . Call it RobotFees. Add the method as shown below.



Now we want to create a test to check that the Robot method getTotalCost is working.

Next create an instance of RobotFees.

public class RobotMockTest {

private RobotFees mockRobotFees;

We are now ready to write our test. The steps for working with mock object are

1. **Create Mock** - This phase creates the mock object.
2. **Expect**- This phase records the expected behaviours of the mock object. These will be verified at the end.
3. **Replay** - Replays the previously recorded expectations.
4. **Verify** - In order for a test to pass, the expected behaviours must have been executed. The verify phase confirms the execution of the expected calls.

@Test  
**public void** testGetTotalCost() {

System.***out***.println(**"getTotalCost"**);

Robot instance = **new** Robot(**"robbie"**, 5);

*/\* Call the createMock to create a mock of the RobotFee class \*/* **mockRobotFees** = *createMock*(**"mockRobotFees"**,RobotFees.**class**);  
  
 */\* Set up the expected values and return values. \*/  
 expect*(**mockRobotFees**.getCost(5)).andReturn(100.00);

*// activate the mock  
 replay*(**mockRobotFees**);  
  
 **double** expResult = 10100.00;

**double** result = instance.getTotalCost(**mockRobotFees**);  
  
 *assertEquals*(expResult, result, 0.5);

*/\* verify that the mock was used correctly \*/  
 verify*(**mockRobotFees**);  
}

**Exercise 1.**

Add the following Engineer class to you src folder

**public class** Engineer {  
  
 **private** String **name**;  
 **private int phone\_number**;  
  
  
 **public** Engineer(String name, **int** phone\_no){  
 setName(name);  
 setPhoneNo(phone\_no);  
 }  
 **public void** setName(String name){  
 **this**.**name** = name;  
 }  
  
 **public void** setPhoneNo( **int** phone\_no){  
 **this**.**phone\_number** = phone\_no;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public int** getContactNo()  
 {  
 **return phone\_number**;  
 }  
  
 **public** String toString() {  
 **return "Name: "** + getName() + **" Phone Number: "** + getContactNo();  
 }

**public** String engDetails() {  
 **return** name + phone\_number;  
}

}

Add the following method to the Robot class.

**public** String recordEngineer(String name, **int** phone\_no){  
  
 Engineer creator = **new** Engineer(name, phone\_no);  
 **return** creator.engDetails();  
}

Now write a test that will test this method without calling the Engineering Class.