LAB 6: Library system

Getting started

In this lab you will be given a set of classes which form part of a library management system. You will examine the code for these classes and identify the relationships between the classes. You will then modify two classes to change the relationship between them. You will then implement custom exception handling and unit testing within this system.

As in previous labs you will look **only at the code for the C# model classes**. You will build a GUI for the system in the next lab.

Task 1: Examining relationships between classes

Open the project:

1. Start a VM which has Visual Studio 2010 installed, and start Visual Studio.

2. Download the file *lab6.zip* from Blackboard, and extract the contents.

3. In Visual Studio, select **File>Open Project**.. and browse to find the file *Library.sln* inside the *Library* folder, then click **Open**. This solution contains one project, which includes the following classes:

    - **Book** – represents a Book item in the library catalog
    - **Copy** – represents a specific copy of a title which may be borrowed
    - **DVD** – represents a DVD item in the library catalog
    - **Loan** – represents the loan of a copy to a member
    - **Member** – represents a library member
    - **Program** – a console application which creates some test objects
    - **Shelf** – represents the location within the library where copies of a title are stored
    - **Title** – represents an item in the library catalog

4. Build the solution.

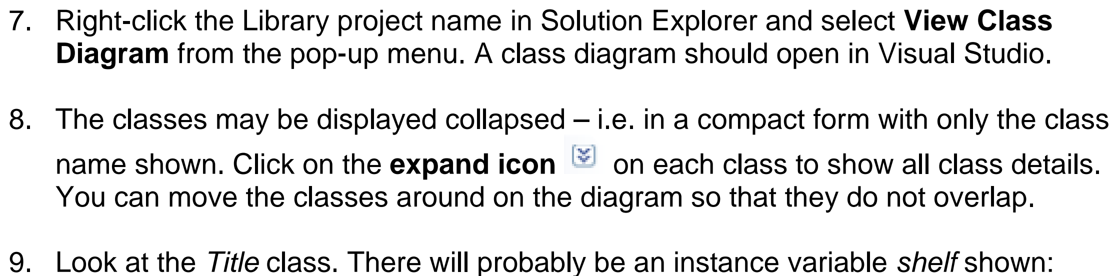
Drawing the class diagram:

5. Review the code in each of the classes. Note in particular the **instance variables** (also known as **fields**) and their types.

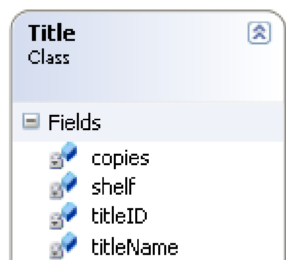
6. Draw a class diagram which represents the classes, their instance variables and the relationships between them. You do not need to include methods or properties in the class diagram. Don't include *Program* as it is not part of the model.

Visual Studio can create a class diagram automatically from the class definitions. You will now do this – you can **compare the result with the diagram you have drawn**.

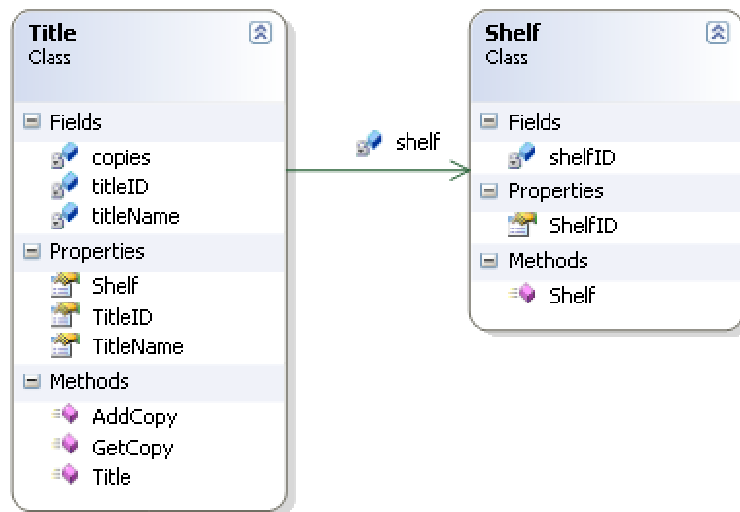
7. Right-click the Library project name in Solution Explorer and select **View Class Diagram** from the pop-up menu. A class diagram should open in Visual Studio.

8. The classes may be displayed collapsed – i.e. in a compact form with only the class name shown. Click on the **expand icon**  on each class to show all class details. You can move the classes around on the diagram so that they do not overlap.

9. Look at the *Title* class. There will probably be an instance variable *shelf* shown:



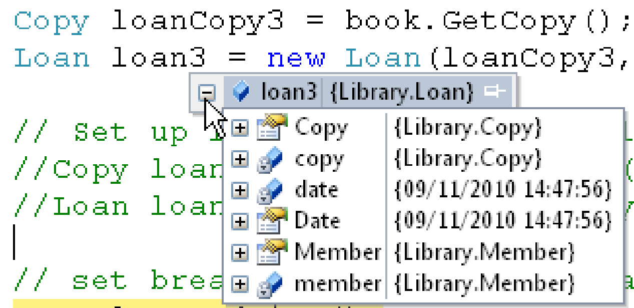
The purpose of the field *shelf* is to implement an association with the *Shelf* class. You can get Visual Studio to show this association. Right-click on the *shelf* field and select **Show as Association** from the pop-up menu. The association between the *Title* and *Shelf* classes should now be shown, and the *shelf* field in *Title* is no longer shown as its existence is implied by the association:



10. Repeat step 9 for all other fields in all classes which are there to implement associations. Note that when a field implements a one-to-many relationship, you need to choose **Show as Collection Association**. If you find Visual Studio displaying a *List* class or a *string* type in the class diagram then you have made the wrong choice of association type or field – choose Undo and try again.

Testing:

11. Open *Program.cs.* Set a breakpoint at the last line of code (the *Console.ReadlLine* statement). Run the program in debug mode. When the execution halts, find a reference to the variable *loan3* and place the cursor over it to inspect it:



What other objects are associated with this *Loan* object?

12. Repeat for the variable *book*. How many *Copy* objects are associated with this *Book* object?

NOTE: The model as it is now supports the selection of a *Title*, and the creation of a *Loan* for the first available copy of that title to a specific *Member*. The *GetCopy* method of *Title* searches through the list of associated *Copy* objects and returns the first one whose *IsAvailable* property has a value of *true*. When a *Loan* is created for a particular *Copy*, the value of *IsAvailable* is set to *false*.

Task 2: Modifying a relationship

There is a new requirement to allow a member to return all borrowed items at once. The *Member* class will need to have a method *ReturnAllLoans* to do this. This will not work with the current model, as *Member* has no reference to any *Loan* objects. The relationship between *Loan* and *Member* is one-way. You will now modify the model classes to support this new requirement.

Modifying the Member class:

1. Add a new instance variable called *loans* of type *List<Loan>* to the *Member* class. Initialise it to an empty list in the constructor of *Member*.

2. Add a new method *AddLoan* to the *Member* class. This should take a *Loan* object as its only parameter, and add this object to the *loans* list.

3. Add the following method to the *Member* class – what do you think this will do?

```
public void ReturnAllLoans()
{
      foreach (Loan loan in loans)
      {
            loan.Return();
      }
      loans.Clear();
}
```

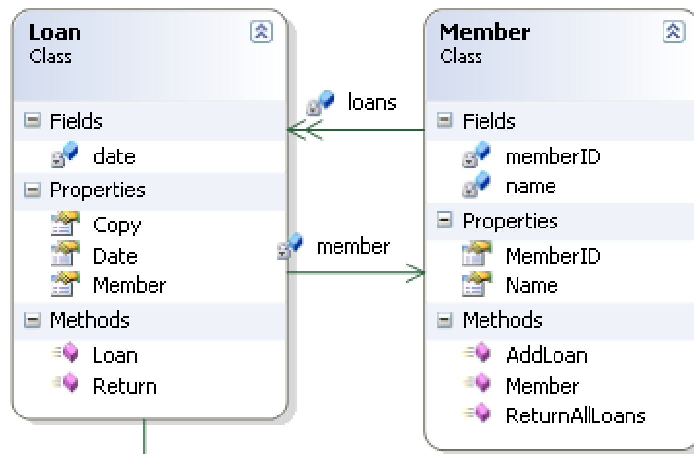
Modifying the Loan class:

The *Loan* class will need to be modified so that when a loan is created for a member, the new loan is added to the member's *loans* list.

4. Add the following code to the constructor of *Loan*:

```
this.member.AddLoan(this);
```

5. Go to the class diagram in Visual Studio and choose to show the *loans* field in *Member* as a collection association. You should now see a bidirectional association between *Loan* and *Member*:



Testing:

6. Add the following line of code to *Program*, just before *Console.ReadLine:*

```
member.ReturnAllLoans();
```

7. Set a breakpoint on the new line. Run the program in debug mode. When the execution halts, place the cursor over the reference to *member* to inspect the object. How many *Loan* objects are associated with this *Member* object?

8. Choose **Debug > Step Over** (or press the F10 key) to move to the next line, so that the *ReturnAllLoans* method is called. Inspect the member reference again - how many *Loan* objects are now associated with this *Member* object?

Task 3: Exceptions

Finding a "bug" and catching an exception:

1. Uncomment the following lines in *Program* so that they are included in the code:

```
//Copy loanCopy4 = book.GetCopy();
//Loan loan4 = new Loan(loanCopy4, member);
```

2. Remove any breakpoints and run the program in debug mode. An exception should occur.

   When there are no copies of a title with *IsAvailable* = *true*, the *GetAllCopies* method in *Title* returns null. What happens when we try to create a loan for a null *Copy*?

   This exception has "crashed" the program. We need to handle the exception so that the program can continue.

3. Enclose the section of code where the loans are created with a try-catch block:

```
try
{
        ...

}
catch (Exception e)
{
        Console.WriteLine(e.Message);
}
```

4. Run the code again. The program should not crash, and you should see an error message in the console.

Throwing an exception:

The try-catch block does handle the error condition, but it doesn't provide much information about the error. It is usually a good idea to make sure that we can handle **specific types of exception** so that we can deal with or report the situation in specific ways depending on what has gone wrong.

The problem is actually caused by the failure of a *Title* object to find an available *Copy* object. We can choose to **throw a specific type of exception** at the point where the problem happens, and include information which *Program* can use to display a more helpful error message.

5. Add a new class to the project called *TitleNotAvailableException*. Use the following code to define the class – note that it is derived from the class *Exception*.

```
public class TitleNotAvailableException : Exception
{
    private string titleName;
    private int copiesOnLoan;

    public string TitleName
    {
        get { return titleName; }
    }

    public int CopiesOnLoan
    {
        get { return copiesOnLoan; }
    }


    public TitleNotAvailableException(string titleName,
         int copiesOnLoan)
    {
        this.titleName = titleName;
        this.copiesOnLoan = copiesOnLoan;
    }
}
```

6. Add the following code just before the *return* statement in the *GetCopy* method in *Title*:

```
if (!found)
   throw new TitleNotAvailableException(titleName,
        copies.Count);
```

Note that when *Title* can't find a copy, it now **throws an exception** object which contains information about the title and how many copies are already on loan.

7. Change the catch block in *Program* to the following:

```
catch (TitleNotAvailableException e)
{
     Console.WriteLine(
        "Title {0} not available, {1} copies on loan",
        e.TitleName, e.CopiesOnLoan);
}
```

8. Run the code again. The program should not crash, and you should see **an informative error message** in the console.

Task 4: Unit testing

In a real project we would create a set of unit tests to test all functionality of the classes in the system, and run these tests every time one part of the system changes during the development process.

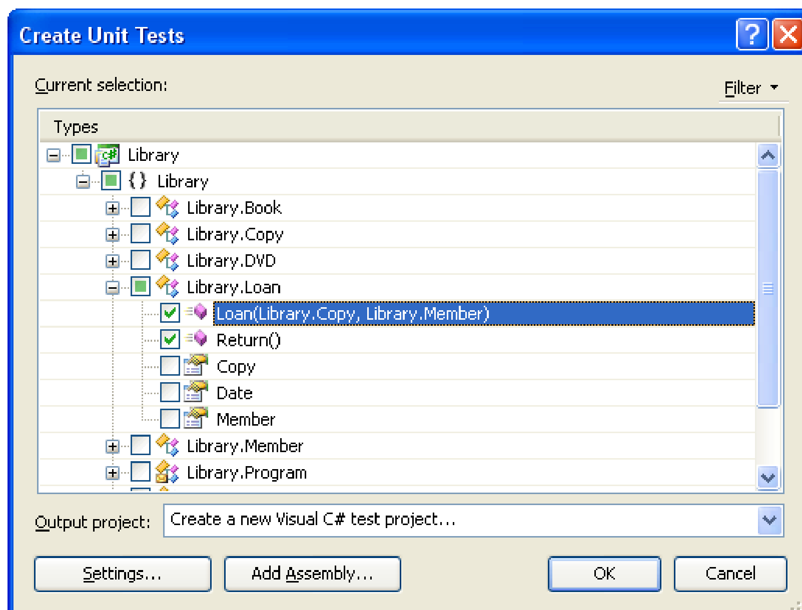
To get a flavour of this approach to testing, you will create a test project and tests for some of the functionality of the *Loan* class. You will test that the following works correctly:

- When a *Loan* is created, the *Copy* is marked as "not available"
- When a *Loan* is returned, the *Copy* is marked as "available"

Creating a Test Project:

1. Open *Loan.cs*. Right-click anywhere in the white space in the class and select **Create Unit Tests**... from the pop-up menu.

2. In the Create Unit Tests Dialog box, check only the constructor and Return method of Loan, and click OK. Make sure the **output project** is a new Visual C# test project. When prompted, call the test project *LibraryTests*.



A new test project should appear in Solution Explorer, with a test class called *LoanTest*.

3. Open *LoanTest* and look at the code. There should be two test methods called *LoanConstructorTest* and *ReturnTest*. Replace the code for these methods with the following:

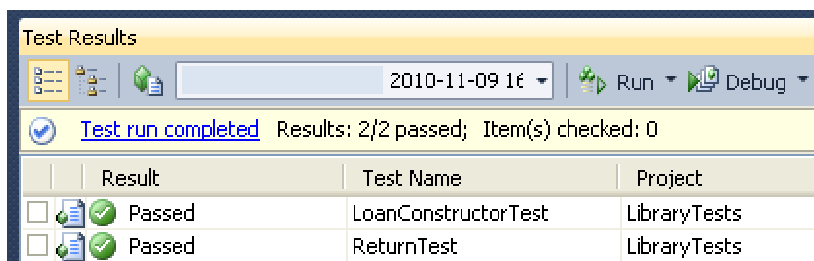
```
 [TestMethod()]
public void LoanConstructorTest()
{
    Copy copy = new Copy("C001");
    Member member = new Member("M001", "Michael");
    Loan target = new Loan(copy, member);
    Assert.AreEqual(copy.IsAvailable, false);
}

 [TestMethod()]
public void ReturnTest()
{
    Copy copy = new Copy("C001");
    Member member = new Member("M001", "Michael");
    Loan target = new Loan(copy, member);
    target.Return();
    Assert.AreEqual(copy.IsAvailable, true);
}
```

Each test method creates a *Copy* and a *Member* object and then creates a *Loan* for that *Copy* to that *Member*.

The constructor test then checks, using an **Assert**, that the copy has been marked as not available. If this is not working correctly, the Assert will cause the test to be failed.

The Return method test creates the Loan and returns it and checks that the copy is now marked as available.

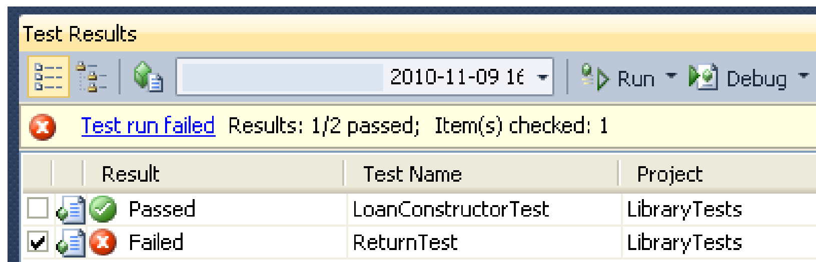
4. Build the solution. There should now be a **Test** item in the main menu. Select **Test > Run > Tests in Current Context.** The Test Results window in Visual Studio should now show the test results:



5. Now you will change something and re-run the tests. In the *Return* method in *Loan*, comment out the line:

```
this.copy.IsAvailable = true;
```

6. Build the solution and run the tests again. You should now see a failure – why?



7. Change your code back to the correct version, build and re-run the tests to check it works correctly again.

8. Finally, you will change something about the way the *Loan* class works, and **check that your change does not affect its behaviour.**

   The constructor and *Return* method of *Loan* contain lines of code, currently commented out, which use the *ToggleIsAvailable* method of *Copy* rather than setting values of the *IsAvailable* property directly, for example:

   ```
   //this.copy.ToggleIsAvailable();
   ```

   Uncomment these lines and comment out the lines which set *IsAvailable*.

9. Build the solution and run the tests again. The tests should now pass.

   Although the code works slightly differently, it **still performs its function as specified in the test.**