Tutorial 7: Introducing Testing plus Better OO in C#

Purpose: To introduce use case-based testing and to learn some more advanced features of OO programming in C#.

If done already: Peer assessment for Assignment 1 (~10 minutes)

At any point during the tutorial complete the peer assessment form for Assignment 1 by having your team members read your description of your contributions and having them sign underneath. The form is available in the Case Study & Assignments section of the MyLO site. If you haven't already, allocate 100 points among your team members on the bottom of the peer assessment form. Submit the form to your tutor.

Outline

One way of generating system-level test cases is to derive them from the use cases that drove the software's design, which means you can begin work on them before any software is actually implemented. This week you'll draft your first test case for the RAP application to familiarise yourself with the test case template and process of filling it in using information from the RTM.

After making a start on your test cases you will improve the object-orientation of your solution from last week by exploring other options for defining a class's fields, and optionally discover how delegates (function types) work in C#, which is how event handlers work in WPF-based GUIs. **In today's exercise we will again use pair programming or, in the case of task 1, work in pairs or triples.**

1 Looking forward to testing your application (~30 minutes)

Download the test case template document from MyLO (available in both the Module 5 and Tutorial areas). Also make sure you have the RAP requirements document and RTM available to refer to. This task will be easier if you work in your new Assignment 2 groups.

Task: Follow the procedure described in Module 5 on *Creating a Test Plan from Use Cases* to design a test case for the first use case in the RAP RTM. Use one of your group's structured scenarios from Assignment 1 as a guide for the various steps a user could take in the (first) test method.

Start by establishing the *criteria* for the test case by looking at both the structured scenario's description and the software constraints (SWC entries in the RTM) that refer to that use case. Then work out the first *method*, or sequence of steps that a user could take to fulfil the use case. (Once you've implemented the system you can refine your methods to match your application's particular user interface.) With respect to the first use case in the RAP case study, you may find that a single method can test more than one way of the user interacting with the staff list, which is fine. But you will probably also be able to identify quite separate methods by which the user could interact with the software.

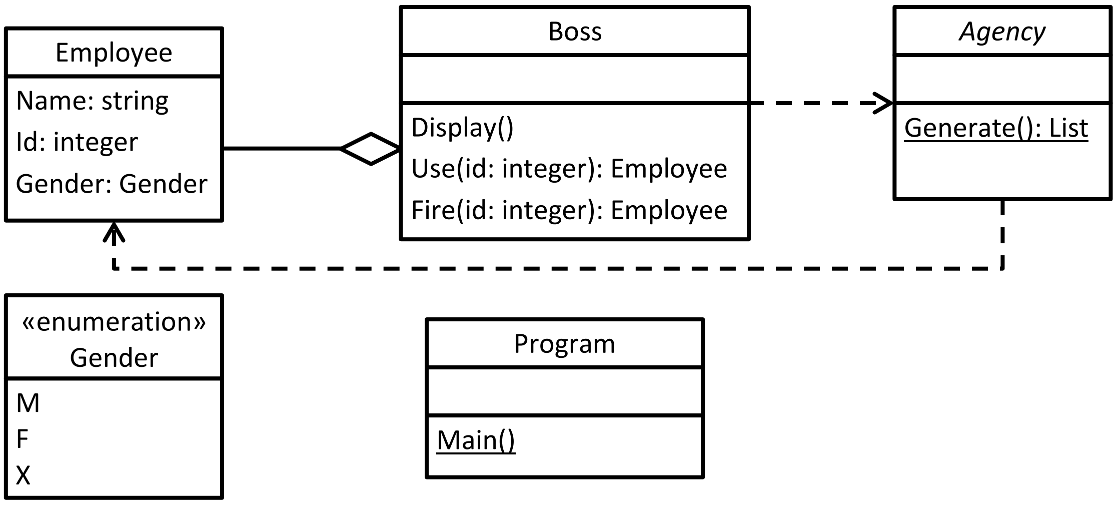
Outside of class, as you continue to work on your assignment, think about the following with regard to your test cases:

- Are the methods you define **black box** or **white box** tests? Once you've worked on the software behind the scenes, are you aware of any patterns of interaction that could (or did during development) lead to faults? (For instance, was it OK to filter the staff list by selecting a category, unless the 'category' was later changed back to 'All'?)
- Does your set of methods cover all the different ways a user could complete the use case?

2 Properties, constructors and delegates in C#

If you have a copy from last week, open (or ensure you have open) the simple employee management system project. If you don't, work through the first few steps from last week until you have a Console Application that includes an additional source file holding a (possibly empty) Employee class definition. We will be replacing the Main() method so move all of its contents into a new method so you don't lose it.

In today's task you will implement the following OO model:



- **Employee** is an entity class with (in this system) no operations apart from a ToString() method (not shown above)
- **Boss** is a *controller* that maintains a List of Employee objects and which can Display() the list of all Employees (on the Console), allows others to Use() an Employee based on his or her ID, and also allows an Employee to be Fire()d (removed from the list), also based on ID
- **Agency** is an abstract class with one class method Generate() that can create a List of Employee objects
- **Program** contains the Main console application program *and* the definition of a *delegate*

You may see some similarities between this model and part of the RAP application you will implement in the assignment.

2.1 Properties and constructors

2.1.1 Converting name, ID and gender to properties

C# supports a range of mechanisms for hiding and exposing object data fields to other parts of a system. Last week we declared the three fields of Employee as public, which is normally poor OO design, since any other object could modify their values. This week we'll make a small improvement by changing them to properties, albeit with just as much access as before.

To declare a string-valued Name property with public read and write access, write:

```
public string Name { get; set; }
```

which, behind the scenes, will declare a simple string-valued field to hold the property's value and automatically create methods to get and set its value. Given an Employee object called e, we can then read and modify this value as if it were a public field:

```
e.Name = "Jane";
Console.WriteLine(e.Name);
```

Two variations on the above property declaration are to have a private set method, or to provide bodies for get and set. The first alternative means the value can only be written to from inside the class, and essentially *requires* that the class have an explicit constructor so that the property can be given an initial value when an object of that class is created. It looks like this:

```
public string Name { get; private set; }
```

The second alternative requires a separate declaration of the backing field, and more lines of code, but gives greater control over how new values are handled (or what concrete value is returned by get). Here's a simple example that only allows non-null values to be set:

```
private string name;
public string Name {
    get {
        return name;
    }
    set {
        if (value != null) {
            name = value;
        }
    }
```

Task: Pick an approach you'd like to use and modify the three fields in Employee to be properties. Adopt the C# naming conventions and give the properties names beginning with an upper case letter.

After the tutorial, if you haven't already, review the MSDN documentation on properties to see the various ways they may be declared.

2.1.2 Implementing the Agency

Create a new class called Agency. Mark it as **abstract**. Move or duplicate the code you wrote last week for generating a list of Employee objects in order to define the static Generate() method, which should return an object of type List<Employee>. Note that in addition to object initialisers (which you used last week), C# also supports collection initialisers, as in:

```
List<Employee> staff = new List<Employee>() {
    new Employee{ Name="Jane", ID=1, Gender=Gender.F },
    new Employee{ Name="John", ID=2, Gender=Gender.M }
};
```

2.1.3 Creating the Boss

Create a new class called Boss. Give it one private field: List<Employee> staff. Also give it a constructor that calls Agency.Generate() to obtain the list of Employee objects to store in the staff field.

Implement Display() by replicating or copy-pasting your code from last week to iterate over a list and print each object to the console. Implement Use() and Fire() as described above: Use() returns the identified object, while Fire() removes that identified Employee from the list (and returns it).

2.1.4 A running program

In Program.Main() declare a variable of type Boss and instantiate it using the **new** operator. Then have Main() call the object's Display() method. **Run it to see how it behaves.**

2.1.5 Using and firing employees

Modify Main() to call either Use() or Fire(), using your knowledge of valid ID numbers, and store the result in a temporary Employee variable. Print the value of that variable (make sure you override Employee.ToString() so you can see which employee was selected) and then call Display() again. **Run the program once calling Use() and once calling Fire().** The different behaviour should be obvious.

2.2 Introduction to delegates (optional for KIT506)

At the moment Main() calls the methods of Boss directly, but it is possible to add an additional level of indirection which could allow the Boss object to be declared and created elsewhere in the project and for Main() to not even know it is using a Boss object. (It still will for the time-being, but it will be possible to separate them further in the future.)

Delegates are data types for references to methods. When declaring a new delegate type you are describing the signature of a method (its parameters and return type). A variable of that delegate type can then be assigned a method name and can be used to invoke (call) that method. This will make more sense when you try it.

2.2.1 Declaring a delegate

Inside the namespace declaration but outside the Program class declaration add the following:

```
public delegate Employee ManageWorker(int id);
```

This defines a new delegate type that is compatible with any method whose return type is Employee and that takes in a single integer parameter. It doesn’t specify what the behaviour of the method must be, only what parameters and return type it must have.

2.2.2 Attaching delegates variables to actual methods

Now that the ManageWorker delegate has been defined, add the following declarations *inside* Main():

```
Action doSomething;
ManageWorker manage;
```

Action is an existing delegate in the System namespace. It is compatible with any method that has no return value (its return type is `void`) and that takes no parameters.

Adapt the code in Main() that called methods on the Boss object directly to be more like the following (which assumes that your Boss object is called boss):

```
doSomething = boss.Display;
manage = boss.Use;

doSomething();
Console.WriteLine("Dealing with {0}", manage(1)); #if 1 is an ID
doSomething();
```

Note that there are no parentheses after the method names in the two assignments; these statements assign those *methods* to the variables doSomething and manage, but do not execute the methods.

Run the program. What happens?

2.2.3 Changing program behaviour by reassigning the delegate

Change the line that assigns a value of `manage` to refer to `boss.Display` and try recompiling. **What’s the result? Why?**

Now change that line to assign the value of `boss.Fire`, recompile and run the program. **What’s the result now?**

3 For your assignment

Practice and apply some of the approaches above by developing the entity (model), controller and database adapter classes in your assignment application. Ensure your controller classes have methods that would be (ultimately) triggered by user actions.