# IN710 OOAD 2014 Practical 5.2 - Factory Patterns

*Checkpoint. Due Wednesday at 9.00 am*

1. The factories in our Animal World only created a single object (an animal). But the Factory Pattern can easily be extended to situations where you need to create a group of objects (as shown in the classic diagram of the Abstract Factory pattern). Imagine, for example, that you were writing code to run (or simulate) a manufacturing process for different models of car. To make each type of car you would need not a single concrete object, but several: doors, wheels, windows, etc. Your abstract factory (implemented as an interface) might look like:

public interface IAutomobileMaker

{

Door makeDoor();

Wheel makeWheel();

Window makeWindow();

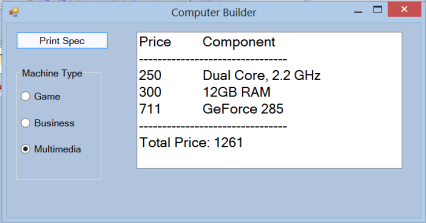
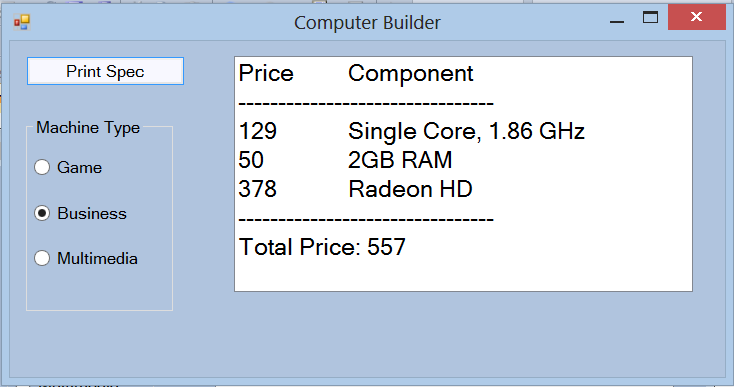
}

You could then create any number of different factory classes, each of which implemented IAutomobileMaker. For example, you might have one for family cars, one for compact cars, one for SUVs, etc. Each of those would have to provide implementations of the makeDoor(), makeWheel() and makeWindow() methods.

The Door, Wheel and Window objects these methods returned would be concrete object instances. These objects would probably be members of their own class hierarchies. For example, there would be a base class Door, from which you would descend a door for a family car, a door for a compact car and a door for an SUV. Wheels and windows would have a similar class hierarchy. The IAutomobileMaker for compact cars would implement a makeDoor() method that made doors for compact cars. It would implement a makeWheel() method that made wheels for compact cars. It would implement a makeWindow() method that made windows for compact cars. These methods would simply need to call the correct constructors to create the desired concrete object instances.

With this architecture, the code to manage the manufacturing process is very clean and general. Given an object that implements IAutomobileMaker, the managing class just calls the makeDoor(), makeWheel() and makeWindow() methods, and can be confident that the correct parts will be produced to make a complete car.

Using this pattern, write an application that creates specs for different types of custom-built PCs. Your application will print out a list of components, and the total cost. For this exercise, you will need to support three types of machine: a high-end gaming machine, a standard business machine and a multimedia machine. Your specification will show a CPU, a graphics card and the required quantity of RAM. Please build your application with (quasi) realistic data values. Here are screenshots of my solution, showing the displays for the Multimedia and Business machines:

Before you start building this application, consider carefully what class and interface hierarchies you need. If you get this wrong, your architecture will not be extensible. As an example, my solution contains the following class definition. Make sure you understand why ***before*** you start coding. (If you need more help getting started, see the end of this handout for further useful code fragments.)

public class GameMemory : Memory

{

public GameMemory()

{

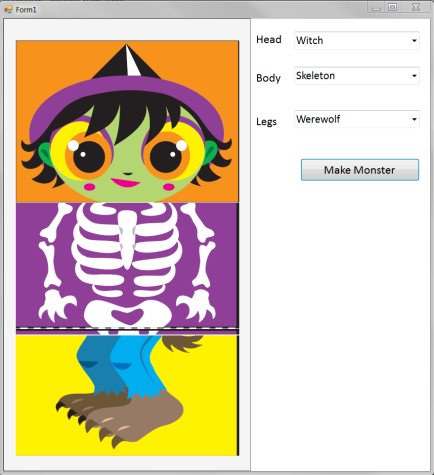
name = "8GB RAM";

price = 200.00;

}

}

1. After your Computer Builder application is running, add a new component to your specification (perhaps a monitor). Add a new type of computer (perhaps a laptop). What classes and class hierarchies did you have to modify? Does your architecture follow the Open/Closed principle?
2. You are developing an application to build model airplane kits using a 3D printer. The user will select from a number of options for wing types, fuselage, wheels, etc. The necessary data & instructions will be passed to the printer, which will create the parts. As a proof of concept, you first want to demonstrate the functionality using 2D images. To do this, build an image “mix and match” application, as shown. There is a demo on the I: drive with prepared .png files, if you wish to use them.



The Form needs to have the necessary input controls (I have used ComboBoxes) and three PictureBoxes. For the purposes of this exercise, you can think of the Form as playing the role of the 3D printer. Instead of the complex data the printer would require, the Form needs to be provided with the three appropriate Bitmap objects to put in its three PictureBox controls.

Before you begin to build this application, think very carefully about your architecture. How can you make the interface respond gracefully to the addition of new character types? What interfaces and/or class hierarchies do you require to minimise coupling between the user interface, the application logic and the “printer”? Remember that you are trying to build an extensible architecture, not just throwing together some code to make a simple mix-and-match game. Try to build the application so that you can easily add new character types, and could easily replace the “provide a Bitmap” functionality with “provide complex printer data”.

## Useful Code Fragments

## 

## 

## 

## 