Labwork 3 Abstract Factory – Object Orientation with Design Patterns 2015

This lab is worth 5% or 50 points from the total 500 points for labwork this semester

Part 1: Extend the Gardening Abstract Factory (6 points)

Create a new project called **Lab3Part1**. Extend the Garden example from the lecture to handle a new subtype called **TreeGarden**. Return three different types of trees, e.g., "Fir", "Ash", "Redwood" (choose one of each type to be center, shade or border).

- Add the new subclass fully implemented in the hierarchy (3 points)
- Implement the new GUI that handles trees (3 points)

Part 2: Gardening Abstract Factory with Garden interface (6 points)

Create a new project called **Lab3Part2**. Re-implement the above Garden Abstract Factory example using an **interface** for the Garden abstract factory.

- Re-write the Garden as interface (2 points)
- Change subclass implementation for interface inheritance (4 points)

Part 3: Advanced product hierarchy in Abstract Factory (23 points)

Problem: Create a new Earth-like planet creation application which will allow many various life-forms and inanimate objects to be created and added to the list of 'things' without the client application needing to worry about the details of the subtypes (or 'things').

Possible solution: use the abstract factory design pattern to return objects belonging to groups of things to include in the creation of the new Earth-like planet.

Create a new project called **Lab3Part3**. In this project use the Abstract Factory Pattern to implement a creation hierarchy for all things on an Earth-like planet (either Living or Non-Living). Test the hierarchy by creating a vector of 1,000,000 things (subtypes of Thing) and print the name of all of the things in the vector and a feature of each thing on the vector, e.g., "Bird has feature wings". It might help to sketch a design of your system to understand the pattern prior to implementation. The Abstract Factory solution should include the following classes\interfaces:

1. An Abstract Factory called **EarthThingsAbstractFactory** (implement as an interface with a getThing() method to force a concrete factory to return a thing from the group of things they create)
2. Two concrete factories called **LivingThingFactory** and **NonLivingThingFactory** (implement a getThing method to return things belonging to the group of objects in some random way, e.g. LivingThingFactory should return an instance of a living thing when called, but may be a different thing each time using randomization)
3. An abstract **Thing** class (with attribute name and feature – feature is something that makes the thing unique, e.g., birds have wings)
4. Abstract subclasses of Thing called **LivingThing** and **NonLivingThing**
5. Concrete subclasses (at least two of each type, living and non-living) with features set, e.g., Bird class, subclass of LivingThing that has name "Bird" and feature "Wings")
6. A test class called **EarthMaker** which will create the vector of 1,000,000 things and prints them to the screen using the factory pattern (NO SUBCLASSES SHOULD BE USED IN THIS **EarthMaker** CLASS, <u>USE THE FACTORY CLASSES ONLY!</u>)

- Abstract factory (implement this as an interface) (3 points)
- Concrete factories (two, implement random getThing()) (4 points)
- Abstract product class (Thing) with generic attributes etc. (5 points)
- Concrete product classes (2 min. each of non-living\living) (6 points)
- EarthMaker (vector of 1,000,000 things in vector and output) (5 points)

Part 4: Abstract Factory with GUI implementation (15 points)

Problem: Write a GUI (in this case a battle game) that can return a group of objects relating to the type of game chosen (sea game or land game)

Possible solution: use the abstract factory design pattern to return the correct groups of related objects for the current game selection.

You have been supplied with a Project called **Lab3Part4 (in Lab3Part4.zip)**. This group of classes does not compile as it is missing three vital components of the abstract factory pattern used to implement it, namely the Abstract Factory and two concrete factories (for Land game and Sea game). Repair this system by writing the three missing classes. Note: the concrete factories must return a random object from the groups of objects available (put these factory classes in the factoryClasses package. It can help to sketch out a UML model of the system to understand the classes that are missing. When the game runs correctly it looks like the images below, i.e., when the "Sea Battle" selection is set in the comboBox only Sea related objects are returned, and when the "Land Battle" selection is set in the ComboBox only Land related objects are returned (Figure 1 and Figure 2):

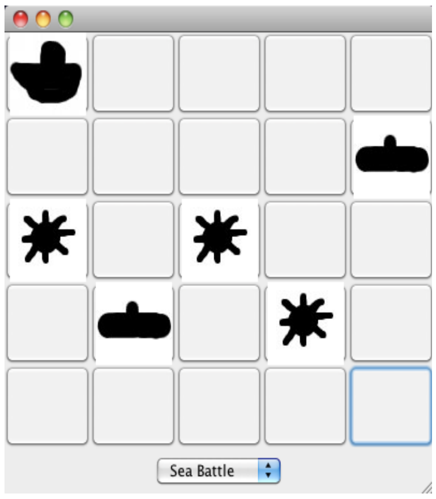

Figure 1: Sea Battle (Mines, Subs, Boats)

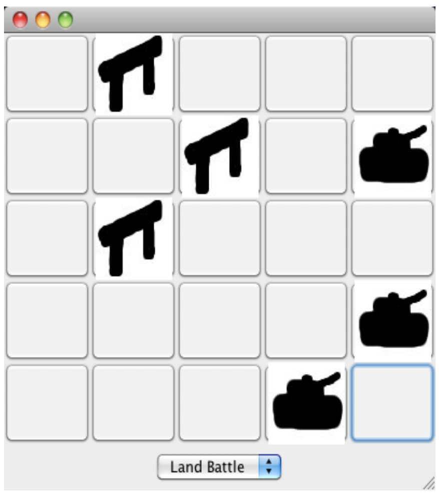

Figure 2: Land Battle (Tanks and Roadblock)

- Abstract Factory added (4 points)
- Land based concrete factory (4 points)
- Sea based concrete factory (4 points)
- Game fully repaired and working (Sea and Land) (3 points)