CSC207: Project Update

Tat Yin Sen, Terry1Alpha, and Yan Nowaczek

November 15, 2021

1 Overview

Missile Mayhem is a time-limited game where a user moves a character to avoid incoming missiles.

2 Design

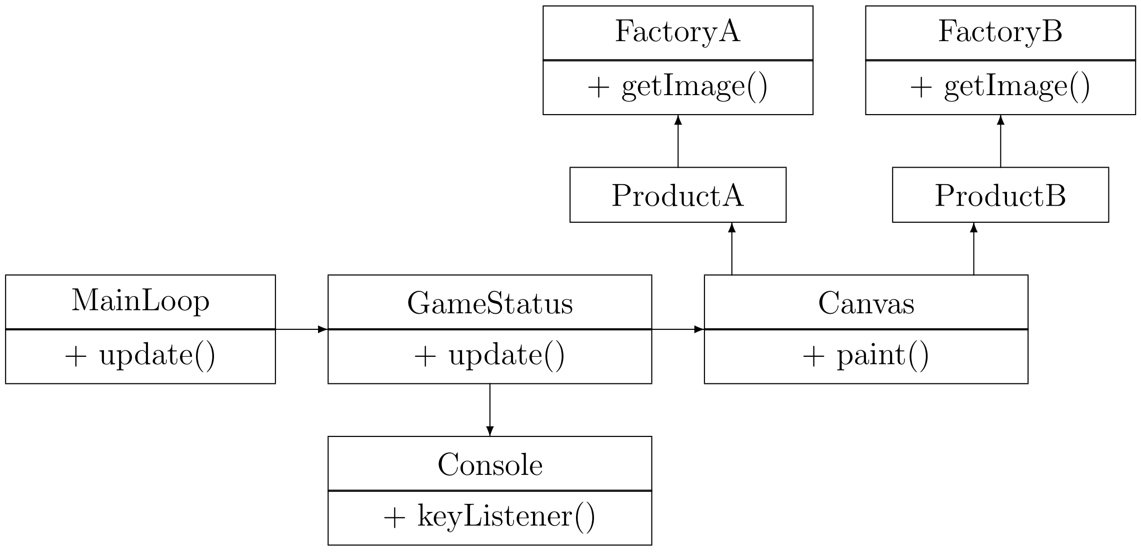

Figure 1: Missile Mayhem: calling dependency.

3 Class details

Overall, the design relies on a small number of classes, each of which contains a small code that deals with a single concern.

TODO Related classes perhaps need to be put in separate folders.

3.1 MainLoop

Class **MainLoop** is very simple. It contains only `main()` that repeatedly calls on class **GameStatus** to update itself and asks if it is time to exit the loop.

TODO Due to its simplicity there is nothing to add to this class.

3.2 GameStatus

Class **GameStatus** keeps track of the game stage and the position of the image that the user can control. It calls on class **Console** to get user input and decides on the stage of the game, e.g., 0 - in progress, 1 - pause, 9 - exit. It passes arrow keys values to class **Canvas** and calls on class **Canvas** to paint the screen.

According to this method the products are created by subclasses of an interface that describes generic behaviour of the subclasses such as getProduct.

TODO Stages in the game progress need to implement. So far there are only three stages: 0 (in progress), 1 (pause) and 9 (exit).

Implement the game clock, game score, target position. (Displaying a text box has failed. It is not possible to control the position of the text box.)

3.3 Console

Class **Console** captures the integer values of keys pressed by the user.

TODO Nothing. The code inside the class is very simple. Perhaps in future there will be a need to capture the information about a button that the user clicked or the text that the user entered into a text box.

3.4 Canvas

Class **Canvas** paints images (products) according to the stage of the game. Initially, this class creates a set of products which during the game, can be added to or remove from the canvas. This class also calls on products to update their positions if appropriate.

TODO Create a list of images (products) so adding to canvas can be accomplished with one line of code.

Figure out how to control the position of text on the canvas. It seems that it can be done with layers and layer managers.

3.5 ProductA, ProductB ...

Class **ProductX** is in the format and contains information that allow class **Canvas** to add them into canvas without any modification. When appropriate **ProductX** contains methods that calculate the image position on the screen.

TODO Apply the **Strategy Design Pattern** to the methods that calculate trajectories. These algorithms are similar to one another and quite complex, which means they require many lines of code that make **ProductX** bulky.

3.6 FactoryA, FactoryB ...

Class **FactoryX** follows the **Factory Design Pattern** - see details below. Essentially, they import images and set default x and y coordinates. This class allow the creation of multiple identical products. Note that some products such as text information are too simple and are not created in factories.

TODO To fully comply with the **Factory Design Pattern** a super class **Factory** needs to be created.

4 Factory Method (party implemented)

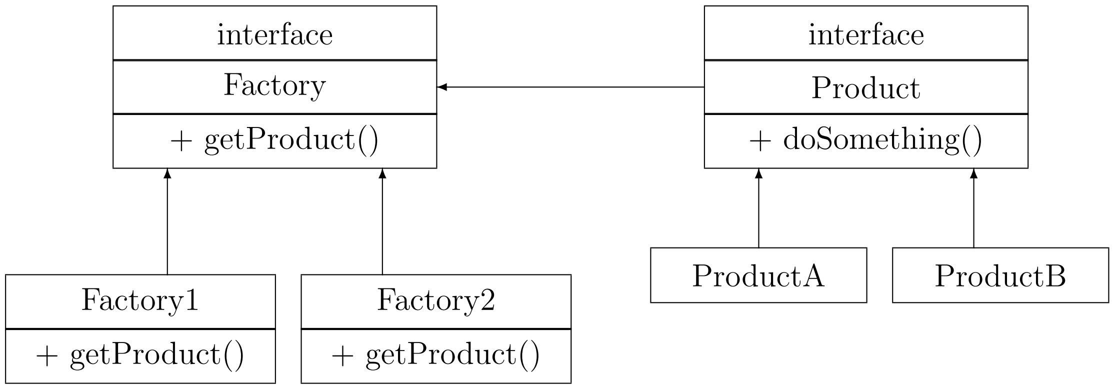

Figure 2: The factory method as a design pattern

The **Factory Method** is a design pattern for creating products (objects) of similar types.

According to this method the products are created by subclasses of an interface that describes generic behaviour of the subclasses such as getProduct.

The main() method in a separate class will actually return the product. The method examines the input ad determines which type of product needs to be created. Through a switch, it instantiates one of the subclasses that implement product creation. Finally, it returns whatever one of this subclasses returns.

The advantages of the Factory Method include the ability to modify how products of specific types are created by making changes to individual subclasses that create products without making changes to the main() method. It also possible and easy to add new product creating subclasses.

This method often features an interface for the product itself. This interface is then implemented by the product creating subclasses.

The anti-pattern, that is the code that benefits from the Factory Method, can be spotted the presence of multiple classes each of which creates products of similar types.

5 Relationship to our Project (Dated)

In the current implementation, out program contains a class called GameEngine. In this class, the main() method places shapes on a canvas simulating the movement of missiles. This is accomplished with many variables that specify the size, colour, direction, and current coordinates for each flying object. The simulation is a loop that starts by erasing the canvas. After a pause of a few milliseconds, the variables for each flying object in sequence are sent as input to a class that calculates the trajectory and the next position. Based on these calculations, the objects are painted on the canvas.

The factory method can be applied to improve the code. Each missile is a product, in this usage. The painting routine requires only two methods: getShape() and getCoordinates(). Therefore, we can have an interface for a product (Missile) and on the basis of this interface we can implement several different types of flying objects. The factory interface will have subclasses, each of which will produce a flying object of a specific type. Finally, the loop in the GameEngine will simply use two methods associated with each missile. These methods are getShape() and getCoordinates(). The figure below illustrates how the factory method can be applied.

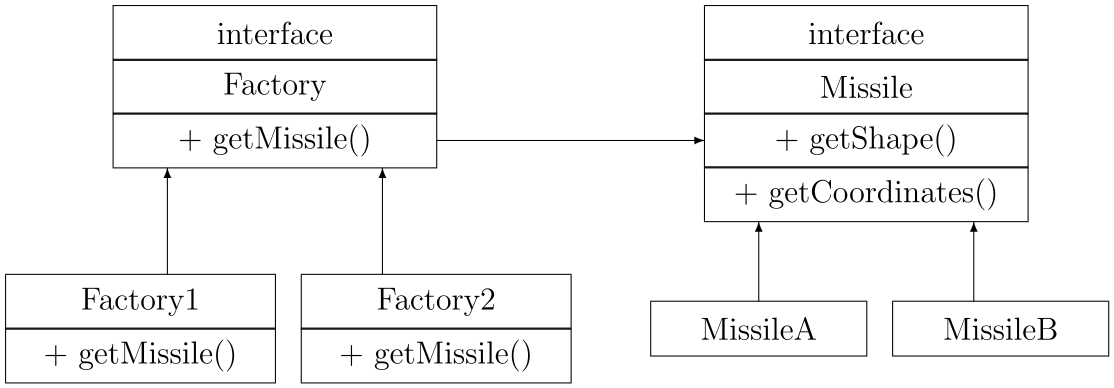

Figure 3: The factory method applied to our project

The code can be further refactored with the Strategy Method. According to the above figure, each product calculates its own trajectory or its next position. We have as many classes as many flying objects, each containing a variation of the algorithm to calculate its next position. Therefore, following the Strategy Method we can create a class or classes that calculate different trajectories and use these trajectory classes as a variable inside each product. In this way, modifications to the trajectory calculations can be done independently of products. Products do not need to know how their movement is calculated.