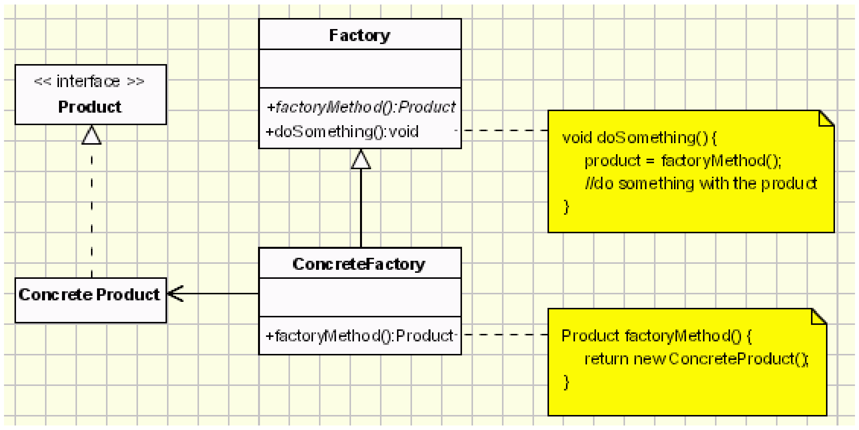
Jack Raney

Design Patterns

9/15/2016

The Factory Method Pattern

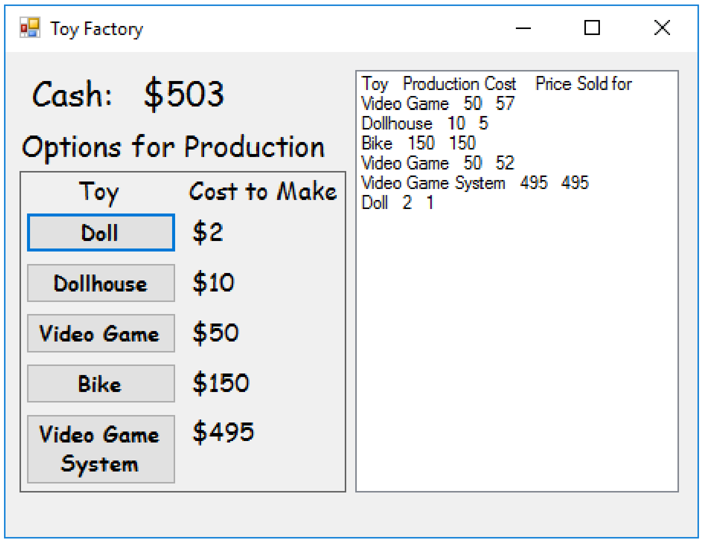
This paper will cover the Factory Method design pattern, a creational pattern used for, as oodesign.com states, "[defining] an interface for creating objects, but let subclasses… decide which class to instantiate" and "[refer] to the newly created object through a common interface". In other words, this pattern will generate objects based off an interface through a "factory" object. oodesign.com's UML diagram can be seen to the right.



```
void doSomething() {
    product = factoryMethod();
    //do something with the product
}
```

```
Product factoryMethod() {
    return new ConcreteProduct();
}
```

To show an example of this pattern, I have created a program called ToyFactory that uses a factory object to create toys, which are then sold.



Cash: $503

Toy	Cost to Make
Doll	$2
Dollhouse	$10
Video Game	$50
Bike	$150
Video Game System	$495

```
Toy   Production Cost   Price Sold for
Video Game   50   57
Dollhouse   10   5
Bike   150   150
Video Game   50   52
Video Game System   495   495
Doll   2   1
```

The toy factory itself is an object of the ToyFactory class, which is derived from the Factory abstract class.

```
public abstract class Factory     //This is the factory class
    {
        private Product product;

        public abstract Product produce();

        public void setProduct()
        {
            product = factoryMethod();
        }

        public Product getProduct()
        {
            return product;
        }
    }
```

Factory contains a Product (which we will talk about later), an abstract method for producing Products, a method that sets the Product object using produce, and a method that returns the Product object that can be used outside of the Factory class.

```
public class ToyFactory : Factory
    {
        public override Product produce()
        {
            return new Toy();
        }
    }
```

ToyFactory simply overrides produce to create Toy objects, a subclass of Product.

```
public interface Product     //This is the product interface
    {
        int sell();
        void setCost(int newCost);
        void setPrice(int newPrice);
    }
```

The Product class is an interface with a method for selling an object derived from it, setting the production cost, and setting the selling price.

```
public class Toy : Product
    {
        private int cost;
        private int price;

        public int sell()
        {
            Random rand = new Random();
            int sellPrice = price + ((rand.Next() % 11) - 8);
            if (sellPrice <= 0)
            {
                return 1;
            }
            return sellPrice;
        }
        public void setCost(int newCost)
            {
                cost = newCost;
            }

        public void setPrice(int newPrice)
        {
            price = newPrice;
        }
    }
```

Toy is derived from Product, and contains cost and price integers alongside the previously described methods. The sell function returns a random value so as to simulate a very simple changing economy.

Within the form there are several constant integers that determine the production cost and selling price for each of the 5 toys sold in the program. There is also a cash integer that keeps track of how much money the factory has to spend, a constant integer that determines how much money the factory starts with, and an initialization of a ToyFactory.

```
public partial class Form1 : Form
    {
        const int DOLL_COST       = 2;   //Cost constants are the cost to make items
        const int DOLLHOUSE_COST  = 10;
        const int VG_COST         = 50;  //VG stands for video game
        const int BIKE_COST       = 150;
        const int VGS_COST        = 495; //VGS stands for video game system

        const int DOLL_PRICE      = 3;   //Price constants are the base price items sell for
        const int DOLLHOUSE_PRICE = 12;
        const int VG_PRICE        = 55;
        const int BIKE_PRICE      = 155;
        const int VGS_PRICE       = 499;

        const int START_CASH      = 500;
        int cash;

        ToyFactory factory = new ToyFactory();
```

On the program's initialization, the listBox is populated with a header string that shows the user how to interpret the items that are read into the box. The starting cash is also set.

```
public Form1()
        {
            InitializeComponent();
            toyListBox.Items.Add("Toy   Production Cost    Price Sold for");

            cash = START_CASH;
        }
```

```
private void makeToy(string toyName, int toyCost, int toyPrice){
    factory.setProduct();
    Toy toy = (Toy) factory.getProduct();

    toy.setCost(toyCost);
    toy.setPrice(toyPrice);

    int marketPrice = toy.sell();

    toyListBox.Items.Add(toyName + "    " + toyCost + "    " + marketPrice);

    cash = cash - toyCost + marketPrice;
    cashLabel.Text = "$" + cash;
}

private void dollButton_Click(object sender, EventArgs e)
{
    makeToy(dollButton.Text, DOLL_COST, DOLL_PRICE);
}
```

A makeToy function within the form is called with each button. This function creates and receives a toy, and then sets its cost and price. Afterwards it determines its market price with the sell function, populates the listBox with the toy's name, cost, and market price, changes the factory's spending cash, and updates the cash label. Each button inputs their own respective toy information.

Reflection

Creating a program with this pattern was difficult for me, as my understanding of the pattern is not exactly thorough. I also struggled with time management and prioritization, so this program did not turn out as I originally envisioned that it would. This assignment has taught me that I need to change something in my current day-to-day planning and/or my priorities in general to have more time to understand the patterns I am assigned.