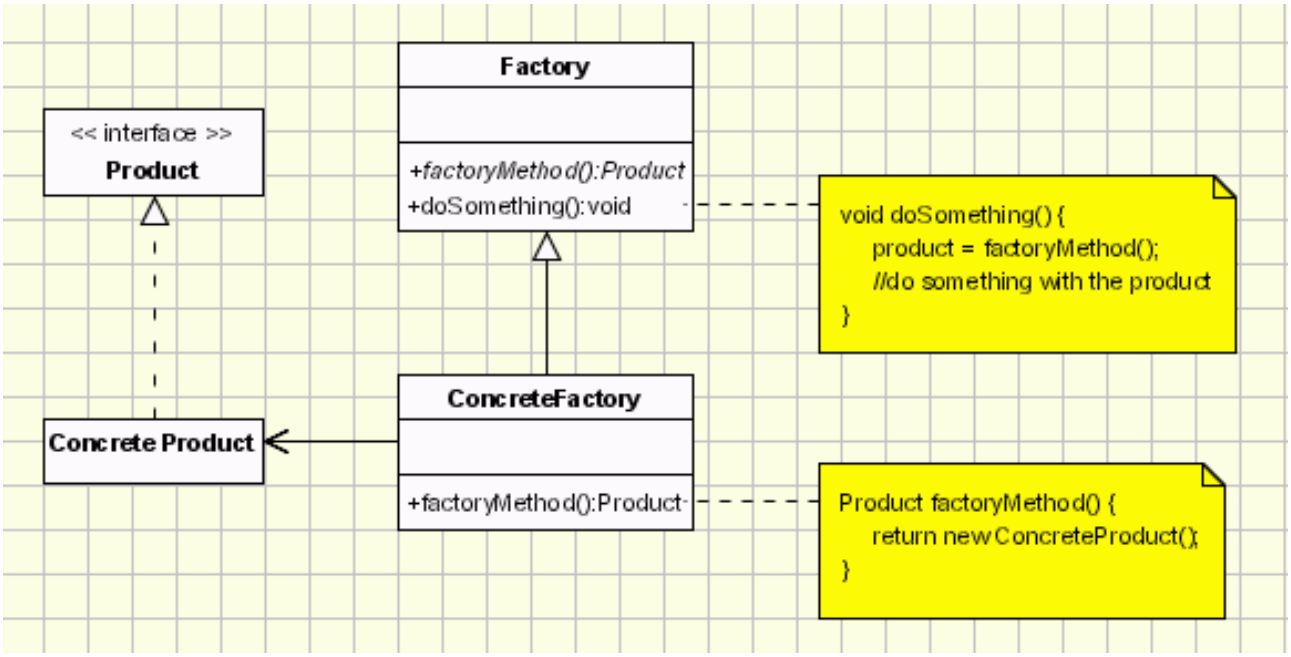
The Façade and Factory Method Patterns
Design Patterns
Andreya Grzegorzewski – Ohio Northern University
Fall 2016

Introduction

My submission for this assignment combines and implements both the façade pattern and the factory method pattern. My application simulates a Walmart store. The store buys products from a factory and sells them to a user at an increased price to make a profit. A form displays a list of all products available for purchases, along with their prices, and the current money in the bank is listed at the right side of the window. The form also contains two buttons; one restocks the store, and one allows the user to buy an item. A Finances class manages the store's money, and an Inventory class manages the store's inventory.

The UML Diagram for Factory Method



The UML diagram for the factory method pattern, shown above courtesy of oodesign.com, shows how the classes in the Factory Method pattern relate to one another. The table below shows how I implemented each class in my application.

Product	I used an interface called Product for the Product requirement. It provided function signatures but not implementations.

Concrete Product	I used a class called ConcreteProduct to implement the Product interface. Products in this case are generic products that might be found at a Walmart store, such as chairs, shirts, and food.
Factory	I used an abstract class called WalmartFactory for this class, with only one abstract method, called factoryMethod. When implemented, this method creates a concrete product.
Concrete Factory	I used a class called ConcreteFactory to implement the factory method. Given an ID number, the factory method returns a concrete product.

Narrative and Code for Factory Method

The first class I created was the Product class, because it has no dependencies on other classes. The code is as follows:

```
public interface Product // This is the product interface
{
    double getPrice();
    string getDescription();
    int getID();
    double getSalePrice();
}
```

Here, I provide signatures, but do not provide any implementation for the methods. After I created the Product class, I moved on to the ConcreteProduct class, which derives from the Product class:

```
public class ConcreteProduct : Product
```

I began by defining the variables I would need to describe a product.

```
private string category; // The type of product - i.e., shirt, chair, etc.
private int id;          // numerical representation of the type of product
private string color;
private string material;
private bool isDangerous = false;
private int initPurchaseCost; // The amount of money that the Walmart Store can buy this
       product for

private string[] categories = { "shirt", "shoes", "chair", "book", "blanket", "food",
       "gizmo", "gadget", "whatchamacallit", "lipstick" }; // potential types of product
private string[] colors = { "black", "white", "gray", "red", "orange", "yellow", "green",
       "blue", "purple", "pink", "brown", "chartreuse" }; // potential colors
private string[] materials = { "cotton", "silk", "wood", "canvas", "brass", "glass",
       "paper", "plastic", "potatoes", "12M sulfuric acid" }; // potential materials each
       product could be made of
```

Then, I defined the constructor. The comments describe and explain each segment of code briefly.

```
public ConcreteProduct(int id)
{
    Random rng = new Random();

    // If no valid ID is passed, we come up with a random ID
    if (id == -1)
        id = rng.Next(categories.Length);
    else
        this.id = id;

    // Assign the category appropriately, and pick a random color and material
    category = categories[id];
    color = colors[rng.Next(colors.Length)];
    material = materials[rng.Next(materials.Length)];

    // Assign danger value based on the type of product created.
    if (category == "food" || material == "12M sulfuric acid")
        isDangerous = true;

    else if (category == "gizmo" || category == "whatchamacallit" || category ==
       "gadget")
    {
    int danger = rng.Next(2);
    if (danger == 1)
        isDangerous = true;
    }

    // It's unlikely that other types of things are dangerous...
    else
    {
        int danger = rng.Next(100);
        if (danger >= 95)
           isDangerous = true;
    }

    // Generate an initial cost between 5 and 15 dollars
    initPurchaseCost = rng.Next(5, 16);
}
```

I then implemented each of the methods defined in the Product interface, which were all simple getters.

```
// Gets the price that the product was purchased for originally
double Product.getPrice()
{
    return initPurchaseCost;
}

// Gets the description, including a warning if the product is dangerous
string Product.getDescription()
{
   string desc = color + " " + category + " made of " + material;
```

```
   if (isDangerous)
       return (desc + " - keep away from small children!");
   return desc;
}

// Gets the ID
int Product.getID()
{
    return id;
}

// Gets the price the product should be sold to customers for
public double getSalePrice()
{
    return (initPurchaseCost * 2);
}
```

After the classes defining the products were created, I could create the factory classes. My abstract Factory class, which provides the signature of the factory method, is as follows:

```
public abstract class WalmartFactory // This is the factory for the factory method
       pattern
{
    public abstract Product factoryMethod(int id);
}
```

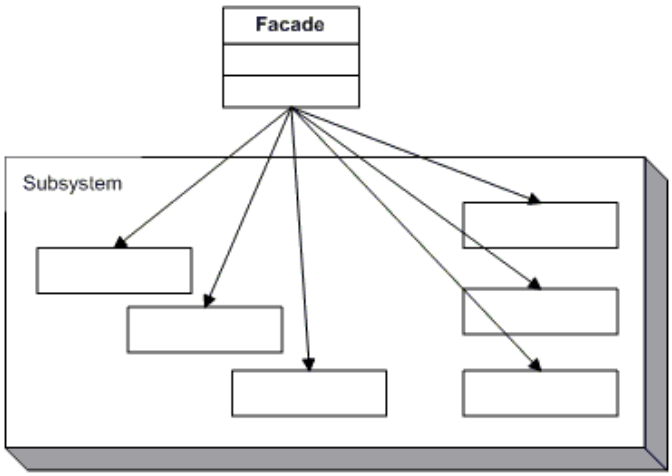
The implementation of this abstract class is almost as simple:

```
public class ConcreteFactory : WalmartFactory // This is the concrete factory as well as
       a subsystem for the facade pattern
{
    public override Product factoryMethod(int id)
    {
        return new ConcreteProduct(id);
    }
}
```

These are the four classes that make up the factory method pattern. The factory method is not called from within this pattern, but rather it is called from an outside source. The façade class from the façade pattern calls the factory method; in fact, the ConcreteFactory class is a subsystem of the façade class, which will be described in the following section.

The UML Diagram for Façade

The façade pattern is a pattern that allows one class, the façade class, to provide one interface to a set of subsystems. The façade class makes each subsystem easier to use. Instead of the interacting with a series of subsystems and unifying those results, a



client can interact only with the façade pattern, which interacts with subsystems discretely. The UML diagram for this pattern is shown to the left, courtesy of dofactory.com, and a table describing my classes and the roles they fulfill is shown below.

Façade	My façade class is the WalmartStore class, which gathers and uses information from the three subsystems to provide the form with a unified interface.
Subsystem 1	My first subsystem is the ConcreteFactory class, which was described in the previous section. This class created the products to be sold at the store.
Subsystem 2	The second subsystem is the ConcreteProduct class, also described in the previous section. This class describes the individual products that are purchased by the Walmart store.
Subsystem 3	The third subsystem is the Inventory class, which keeps a record of the number of products of each category currently in the inventory.
Subsystem 4	My final subsystem is the Finances class, which keeps track of the money available to the store and records purchases and sales.

Narrative and Code for Façade

I started working on my application by first writing the subsystems, then writing the unifying façade class. The first subsystems that I wrote were the ConcreteProduct and ConcreteFactory classes, which were described along with the rest of the factory method pattern in the previous section. After I finished that, I moved on to the Inventory class. It contains only one variable:

```
private int[] categoryCount = new int[10]; // Keeps track of the number of products of
       each category
```

There are ten total categories, which were listed in the ConcreteProduct class in the previous section. There is a method for removing a product from the inventory, and another for adding a product to the inventory. These methods both take one argument in the form of an ID number, which represents the category of a product. The methods are as follows:

```
public void addProduct(int id)
{
	categoryCount[id]++;
}

public void removeProduct(int id)
{
	categoryCount[id]--;
}
```

Finally, there is a method for determining which category of product is understocked. It returns the ID number of the type of product that is present in the inventory the smallest number of times.

```
public int getLeastCommonCategory()
{
    int least = 0;
    for (int i = 1; i < 10; i++)
    {
        if (categoryCount[i] < categoryCount[least])
            least = i;
    }
    return least;
}
```

This comprises the Inventory class. The last subsystem of the façade class is the Finances class. This class is very simple; it has a variable for the amount of money in the bank which is updated when a purchase or sale is made. There is also a getter for the money variable. The class in its entirety is shown below:

```
public class Finances // This is a subsystem in the facade pattern
{
    double money = 250.00;  // Start with $250 in the bank
    // Gets the amount of money that the store can safely spend on new products
    public double getMoney()
    {
        return money;
    }

    // Updates the store's money when it purchases new products
    public void buy(double cost)
    {
        money -= cost;
    }

    // Updates the store's money when it makes a sale
    public void recordSale(double saleAmt)
    {
        money += saleAmt;
    }
}
```

Finally, once all the subsystems were written, I began to work on the façade class, WalmartStore. This class unified the three subsystems to perform the basic operations of a store. I began by declaring the variables, which make use of each of the subsystems:

```
private ConcreteFactory cf;
private Inventory i;
private Finances f;

List<Product> products = new List<Product>();
```

Then, I created a simple constructor for the WalmartStore class.

```
public WalmartStore(ConcreteFactory factory, Inventory inventory, Finances finances)
{
    cf = factory;
    i = inventory;
    f = finances;
}
```

I then wrote a method to initialize the store with 25 random products already in it.

```
// Used only once at the beginning of the program to get some products in the store
// Therefore we don't actually need to buy anything, and the finances variable isn't
used.
public string[] initializeStore()
{
    string[] productsAndCosts = new string[50]; // Only buy up to 25 things at once

    for (int index = 0; index < 50; index += 2)
    {
        // Create a product
        Product thing = cf.factoryMethod(-1);

        // Account for the product
        products.Add(thing);
        i.addProduct(thing.getID());

        // Even indices are descriptions of products; odd indices are costs
        productsAndCosts[index] = thing.getDescription();
        productsAndCosts[index + 1] = thing.getSalePrice().ToString();

        Thread.Sleep(15); // So the random number generator used in the ConcreteProduct
              constructor generates different numbers every time
    }
    return productsAndCosts;
}
```

Next, I needed a restock method that the store could use. When a store restocks its inventory, it orders the things it needs first; therefore, my restock method orders 25 products at maximum, where each product is the product that the store currently has the least of.

```
public string[] restockStore()
{
    // We have to keep track of money this time
    double moneyToSpend = f.getMoney();
    string[] productsAndCosts = new string[50]; // Only buy up to 25 things at once

    for (int index = 0; index < 50; index += 2)
    {
        // Make a product of the least common category
        Product thing = cf.factoryMethod(i.getLeastCommonCategory());

        // If we can afford it...
        if (thing.getPrice() < moneyToSpend)
        {
            // Buy the product and add it to the list
            products.Add(thing);
            f.buy(thing.getPrice());
            moneyToSpend -= thing.getPrice();
            i.addProduct(thing.getID());

            // And document it
            productsAndCosts[index] = thing.getDescription();
            productsAndCosts[index + 1] = thing.getSalePrice().ToString();

            // Then wait so the random number generator will come up with a new product
            next time

            Thread.Sleep(15);
        }
        else break; // If we're broke, break
    }
    return productsAndCosts;
}
```

Finally, the Walmart store needed a way to sell a product to the user. This method records a sale and removes the product that was sold from the lists.

```
public void sellToUser(int index)
{
    Product thing = products[index];

    f.recordSale(thing.getSalePrice());
    i.removeProduct(thing.getID());

    products.Remove(thing);
}
```

The Form and the Deliverable

The only part of the code left after this is the code for the form, which the user interacts with directly. The form only calls methods from the WalmartStore class, which shows that it implements the façade pattern correctly. The partial class WalmartForm, which derives from Form, begins by creating variables for three of the subsystems, as well as the WalmartStore class, and setting up the form for use.

```
// Variables for subsystems and Walmart Store
ConcreteFactory factory;
Inventory inventory;
Finances finances;
WalmartStore wm;

public WalmartForm()
{
    InitializeComponent();

    // Sets up WalmartStore class
    factory = new ConcreteFactory();
    inventory = new Inventory();
    finances = new Finances();
    wm = new WalmartStore(factory, inventory, finances);


    // Displays the 25 products currently in the inventory
    string[] productsAndCosts = wm.initializeStore();

    for (int i = 0; i < productsAndCosts.Length; i += 2)
    {
        productsLB.Items.Add(productsAndCosts[i]);
        costsLB.Items.Add(productsAndCosts[i + 1];
    }
}
```

Next, I needed a method to show a new product in the list boxes that contain product descriptions and prices:

```
public void addProduct(Product thing)
{
    productsLB.Items.Add(thing.getDescription());
    costsLB.Items.Add(thing.getSalePrice());
}
```

Then, I needed a way to show the user how much money is currently in the bank. This method is called when actions affecting money are performed.

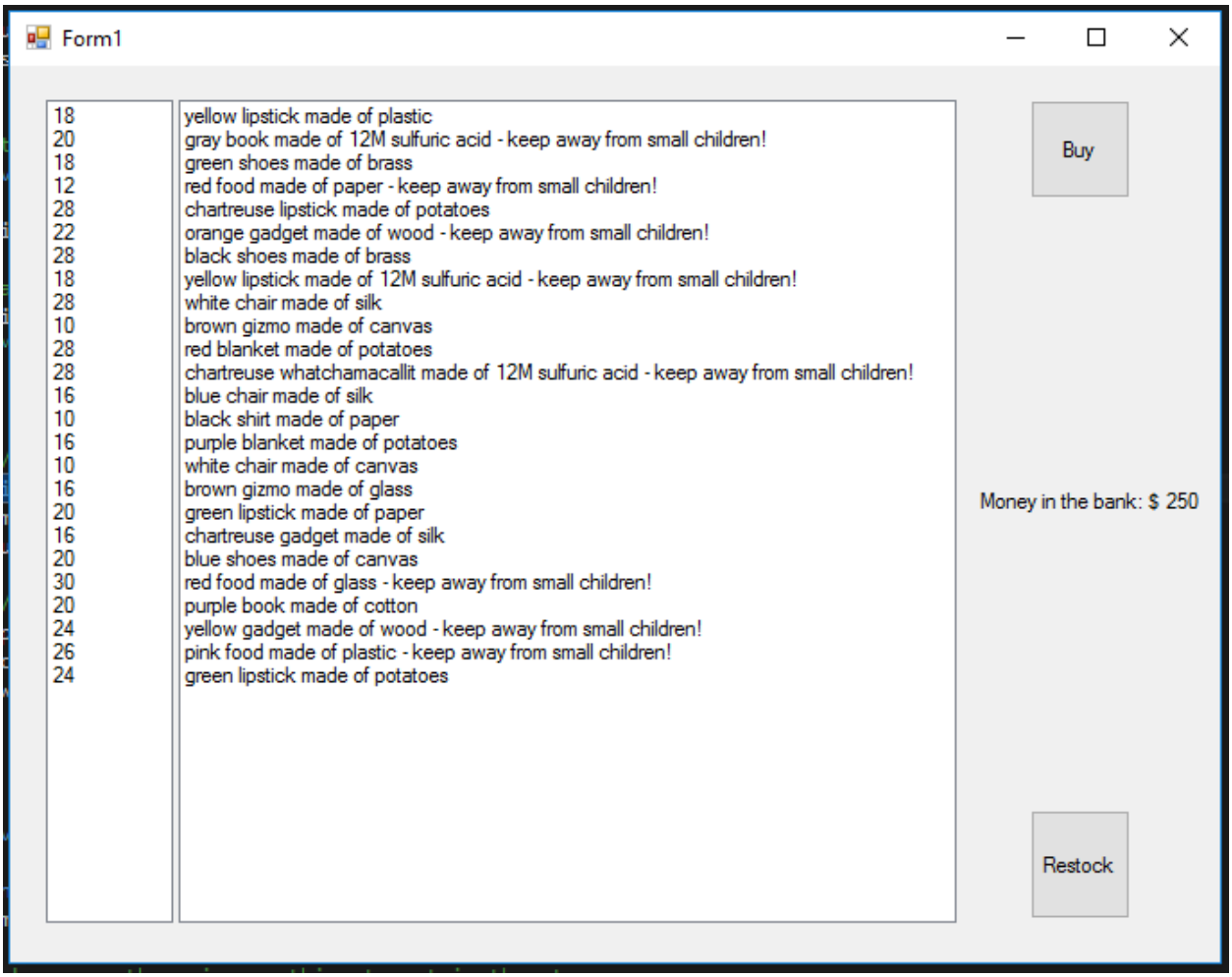
```
private void updateMoney(int money)
{
    moneyLabel.Text = "$" + money.ToString();
}
```

As mentioned previously, there are two buttons on the form: the buy button and the restock button. I needed one method for each of those buttons. The buy button removes the items from the list boxes and documents the purchase through the WalmartStore class, and the restock button buys new products for the store to display. These are the last two functions in the form, and with them, the coding of my application was completed.

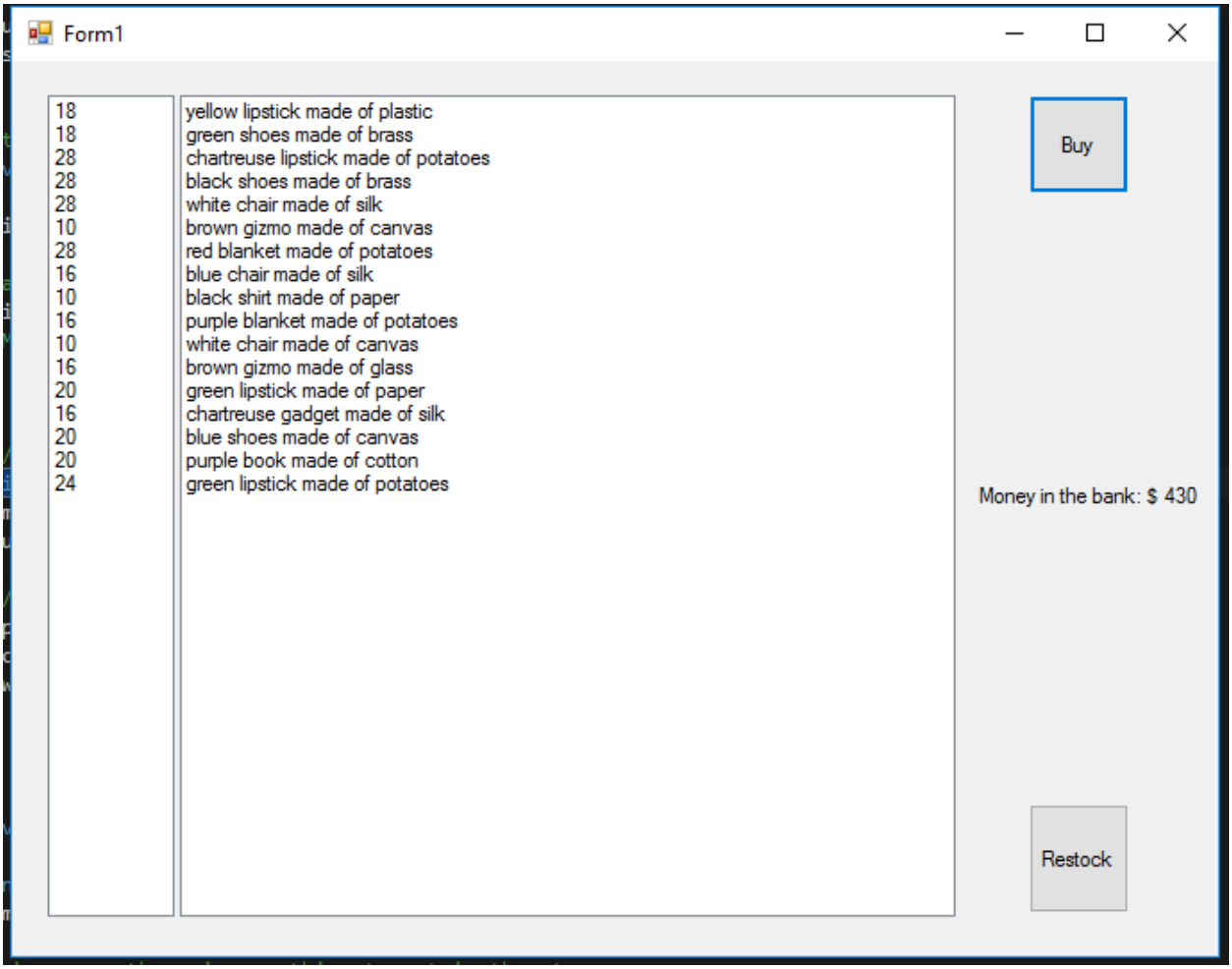
```
// Shows a new product in the list boxes
public void addProduct(Product thing)
{
```

```
    productsLB.Items.Add(thing.getDescription());
    costsLB.Items.Add(thing.getSalePrice());
}

// Lets the user buy a product
private void buyButton_Click(object sender, EventArgs e)
{
    int index = productsLB.SelectedIndex;

    // Makes sure something is selected
    if (index == -1)
        MessageBox.Show("Choose what you'd like to buy!");

    else
    {
        // Updates the money label
        int money = Convert.ToInt32(moneyLabel.Text);
        money += Convert.ToInt32(costsLB.Items[index]);
        updateMoney(money);

        // Removes the product from the list boxes and sells the product
        productsLB.Items.RemoveAt(index);
        costsLB.Items.RemoveAt(index);
        wm.sellToUser(index);
    }
}

// Makes new products available for purchase
private void restockButton_Click(object sender, EventArgs e)
{
    string[] productsAndCosts = wm.restockStore();
    int money = Convert.ToInt32(moneyLabel.Text);

    // Makes sure there is something to put in the store
    if (productsAndCosts[0] == null)
        MessageBox.Show("There isn't enough money in the bank to restock!");

    else
   {
        for (int i = 0; i < productsAndCosts.Length; i += 2)
        {
            // Makes sure there is an item to stock
            if (productsAndCosts[i] == null)
                break;

            // Adds the item and updates the money count
            productsLB.Items.Add(productsAndCosts[i]);
            costsLB.Items.Add(productsAndCosts[i + 1]);
            money -= (Convert.ToInt32(productsAndCosts[i + 1]) / 2);
        }
    }
    updateMoney(money); // Displays the money
}
```

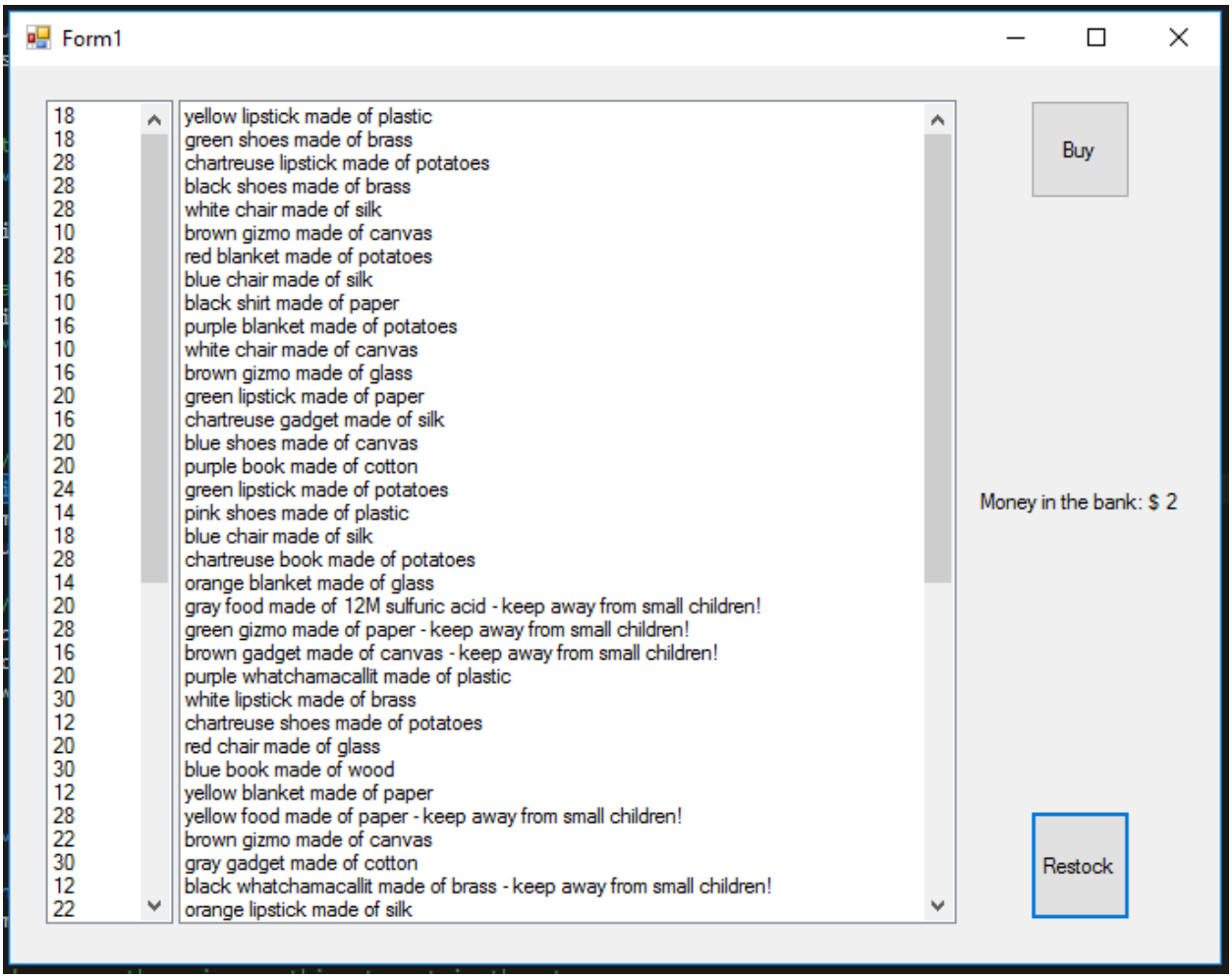
Screenshots of the application in action are shown below.



This was displayed when the program was first opened. There were $250 in the bank and 25 products in the store.



This is what was displayed after I purchased every dangerous object in the store. The money was updated with each purchase and reached $430.



This is what was displayed after I restocked the store two times. When the money in the bank reached $2, no more products could be purchased.

Observations

Writing this application was very helpful to me because I could see how two patterns worked together. I could see myself using the façade pattern a lot in the future, and the factory method pattern upon occasion. I like the façade pattern because it can apply to so many different types of situations, and because it can simplify the end result of the program significantly. Additionally, neither of these patterns have any convoluted elements that make them unapproachable to a new programmer, so they are ideal for where I am in my education. I really enjoyed writing this program, and I think that my application meets the requirements of the assignment. I used the factory method to create several products, and I used a façade class to manage the factory and the products, as well as two other subsystems. Only the façade class accessed the subsystems at any point in the program, and the form only communicated directly with the façade class.