Abstract Factory

Design Patterns

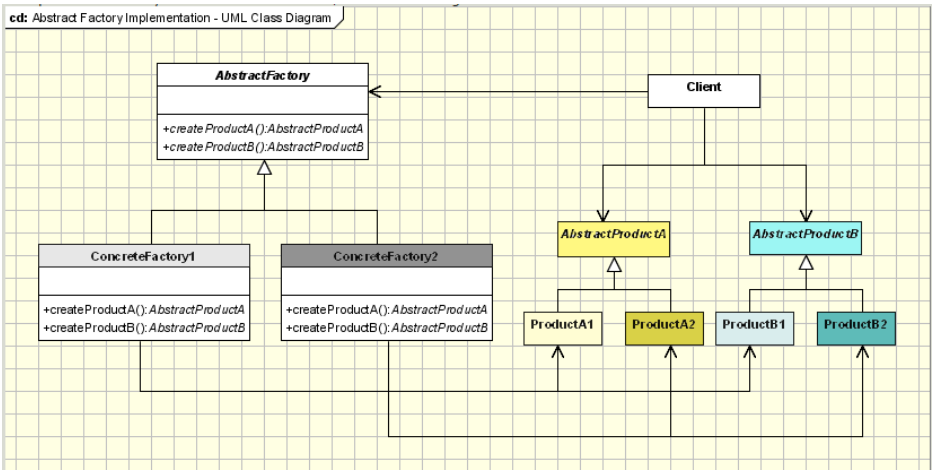
Christopher Doseck

9/28/2016

Introduction

This assignment is an application that I created to show how the factory method pattern works. In this application I am using a comic book character to represent the abstract product and create a comic book character for the abstract factory. I used heroes and villains as concrete products, and creating heroes and villains as concrete factories.

The UML Diagram for Abstract Factory



The UML diagram for the abstract factory pattern, shown on the right, shows the classes that are needed to have the requirements. I had the comic character as the product interface, the hero and villain as two concrete products, the create comic character class as the abstract factory interface, and the create hero and create villain as the concrete factory. The table below shows how each of the classes were used.

ComicCharacter	This is the abstract product interface. It had the basic setup for what every comic character would need.
Hero	This is a concrete product. It represents what characteristics a hero would have.
Villain	This is a concrete product. It represents what characteristics a villain would have.

CreateComicCharacter	This is an abstract factory interface. It contains what the basis is to create any ComicCharacter.
CreateHero	This is the class that creates an object of the Hero class.
CreateVillain	This is the class that creates an object of the villain class.
Form1	This is the GUI that shows the Abstract Factory Pattern in action.

Narrative

```
public interface ComicCharacter
{
    string getName();
    string getPowers();
    string getSummary();
}
```

This is the abstract product. All comic characters must have a method to get the name and powers of the character, along with a method to get the summary of the character.

```
public class Hero : ComicCharacter
{
    private string name;
    private string powers;
    private string city;
    private string origin;

    public Hero(string name, string powers, string city, string origin)
    {
        this.name = name;
        this.powers = powers;
        this.city = city;
        this.origin = origin;
    }

    public string getName()
    {
        return name;
    }

    public string getPowers()
    {
        return powers;
    }

    public string getCity()
    {
        return city;
    }

    public string getOrigin()
    {
        return origin;
    }
```

This is the Hero class. It is a concrete product that is a child of ComicCharacter. All of the information is passed through the constructor to it. It has methods that return all of the values that are used. getSummary is used to create a summary of the character in the form of a sentence.

```
    public string getSummary()
    {
        return "* " + getName() + " is a superhero from " + getCity() + " who got " +
getPowers() + " when " + getOrigin() + ".";
    }
}


class Villain : ComicCharacter
{
    private string name;
    private string powers;
    private string motive;
    private string rival;

    public Villain(string name, string powers, string motive, string rival)
    {
        this.name = name;
        this.powers = powers;
        this.motive = motive;
        this.rival = rival;
    }

    public string getName()
    {
        return name;
    }

    public string getPowers()
    {
        return powers;
    }

    public string getMotive()
    {
        return motive;
    }

    public string getRival()
    {
        return rival;
    }

    public string getSummary()
    {
        return "* " + getName() + " is a villain of " + getRival() + " who can " +
getPowers() + ". The reason they do this is because " + getMotive() + ".";
    }
}

interface CreateComicCharacter
{
    ComicCharacter createCharacter();
}
```

This is the Villain class. It is a concrete product that is a child of ComicCharacter. All of the information is passed through the constructor to it. It has methods that return all of the values that are used. getSummary is used to create a summary of the character in the form of a sentence.

This is the interface to create a comic character. It has the method createCharacter which return a ComicCharacter.

This is the CreateHero class, which is a concrete factory class. This class is a child class of the CreateComicCharacter interface. In this class, it is passed all of the info needed to create a Hero object in the constructor. In the createCharacter method it returns a new object of the Hero class.

```
class CreateHero : CreateComicCharacter
{
    private string name;
    private string powers;
    private string city;
    private string origin;

    public CreateHero(string name, string powers, string city, string origin)
    {
        this.name = name;
        this.powers = powers;
        this.city = city;
        this.origin = origin;
    }

    public ComicCharacter createCharacter()
    {
        return new Hero(name, powers, city, origin);
    }
}
```

This is the CreateVillain class, which is a concrete factory class. This class is a child class of the CreateComicCharacter interface. In this class, it is passed all of the info needed to create a Villain object in the constructor. In the createCharacter method it returns a new object of the Villain class.

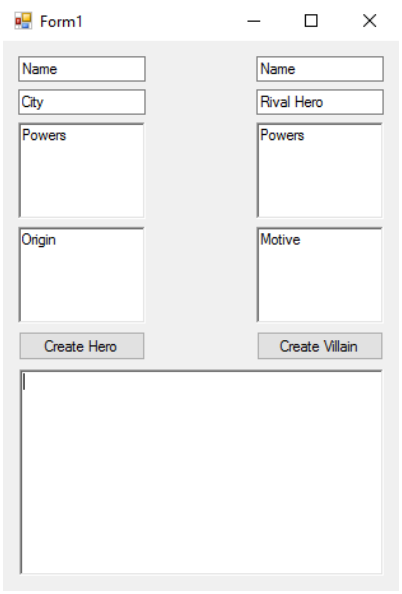
```
class CreateVillain : CreateComicCharacter
{
    private string name;
    private string powers;
    private string motive;
    private string rival;

    public CreateVillain(string name, string powers, string motive, string rival)
    {
        this.name = name;
        this.powers = powers;
        this.motive = motive;
        this.rival = rival;
    }

    public ComicCharacter createCharacter()
    {
        return new Villain(name, powers, motive, rival);
    }
}
```

This is the form that shows the Abstract Factory Pattern in action. It has a CreateComicCharacter variable named factory and a ComicCharacter variable named character. When the create hero button is clicked it creates a new CreateHero object which it assigns to factory. After this, the factory calls its createCharacter method and assigns that to character. The textbox at the bottom then adds the character summary to it.

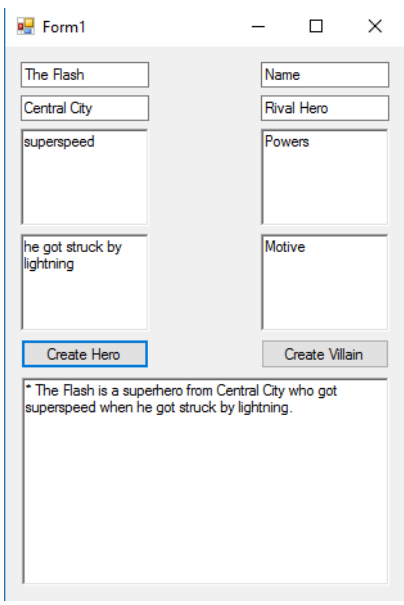
```
public partial class Form1 : Form
{
    private CreateComicCharacter factory;
    private ComicCharacter character;
    public Form1()
    {
        InitializeComponent();
    }

    private void btnCreateHero_Click(object sender, EventArgs e)
    {
        factory = new CreateHero(tbHeroName.Text, tbHeroPowers.Text, tbHeroCity.Text,
tbHeroOrigin.Text);
        character = factory.createCharacter();
        tbHeroesAndVillains.Text += character.getSummary() + "\n \n";
    }
```

```
    private void btnCreateVillain_Click(object sender, EventArgs e)
    {
        factory = new CreateVillain(tbVillainName.Text, tbVillainPowers.Text,
tbVillainMotive.Text, tbVillainRival.Text);
        character = factory.createCharacter();
        tbHeroesAndVillains.Text += character.getSummary() + "\n \n";
    }
}
```

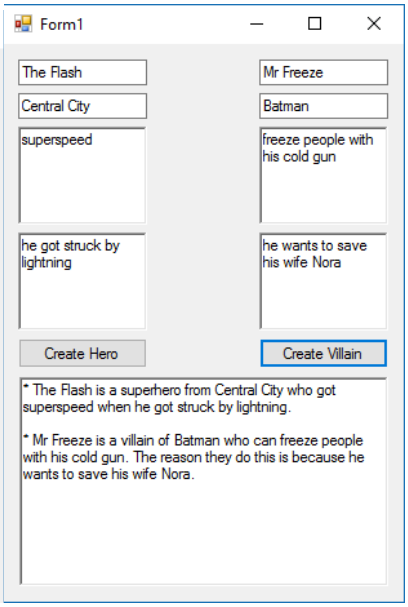
When the create villain button is clicked it does the same thing as the other one, but it uses the CreateVillain class instead of the CreateHero class.



This is the basic setup of the Form.



This is after information is put in for the flash, and create hero is pressed.



This is after information is put in for Mr. Freeze, and create villain is pressed.

Observations

This was a rather straightforward design pattern. I thought that it was rather easy to do. I can see that the purpose in this is creating related objects without specifying their classes.