Tori Lentz

Dave Retterer

Design Patterns

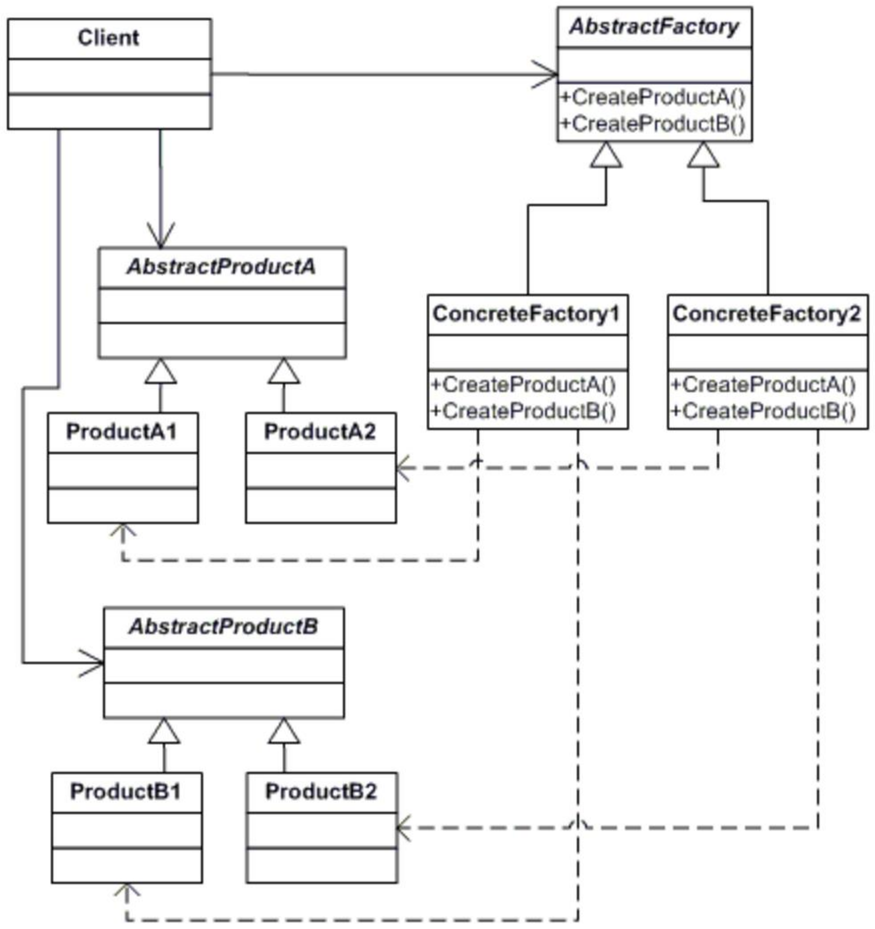
November 21, 2016

Abstract Factory Pattern

Introduction

The purpose of this assignment is to implement the Abstract Factory pattern. This pattern is typically used to avoid having to add more code to existing classes. By having a pattern that works this way, one can just add the new code as new classes which will then match up to its corresponding concrete factory.

UML Diagram



Abstract Factory	This class declares an interface for operations that create the abstract products.
Concrete Factory	This class implements the operations to create concrete product objects.
Abstract Product	These classes declare an interface for a type of product object.
Product	These classes declare which product object is to be created.

Client	The client uses the interfaces declared by the Abstract classes.
http://www.dofactory.com/net/abstract-factory-design-pattern	

The abstract factory method is a very complex pattern that includes multiple parts. The first part is the client which, in my case, will be my form. This client class uses the interfaces that have been declared by the Abstract classes, both Factory and Product. Next, is the Abstract Factory class that declares the interface for the operations that create the Abstract Products for this pattern. There are multiple Concrete factories which implement the operations to create concrete product objects. The Abstract Products define the interface for which type of product object someone would want to create. Finally, are the Product classes that define which product object is to be created by the corresponding Concrete Factory and, also implements the Abstract Product interface.

Code and Description

Client

```
public class PokemonWorld
{
    private Wild _wild;
    private Captured _captured;

    public PokemonWorld(PokemonFactory factory)
    {
        _wild = factory.CreateWildPokemon();
        _captured = factory.CreateCapturedPokemon();
    }

    public string Battle()
    {
        return _captured.Battle(_wild);
    }
}
```

This is my client class. This class uses the abstract class or interface that was created in my abstract factory and abstract product classes. For this pattern, my abstract factory is called PokemonWorld and my method that I call is Battle.

Abstract Factory

```
public abstract class PokemonFactory
{
    public abstract Wild CreateWildPokemon();
    public abstract Captured CreateCapturedPokemon();
}
```

This is my abstract factory class and, within this class, I create two methods that will be defined in my Wild and Captured classes.

Concrete Factory 1

```
public class FirstGenerationFactory : PokemonFactory
{
    public override Wild CreateWildPokemon()
    {
        return new Dratini();
    }

    public override Captured CreateCapturedPokemon()
    {
        return new Charamander();
    }
```

This is my first concrete factory which inherits from PokemonFactory and creates two first generation pokemon, one of either type, wild and captured.

```
    }
```

Concrete Factory 2

```
    public class SecondGenerationFactory : PokemonFactory
    {
        public override Wild CreateWildPokemon()
        {
            return new Mareep();
        }

        public override Captured CreateCapturedPokemon()
        {
            return new Cindaquill();
        }
    }
```

This is my other concrete factory which created second generation pokemon of both types, wild and captured.

Abstract Product A

```
    public abstract class Captured
    {
        public abstract string Battle(Wild w);
    }
```

Abstract Product B

```
    public abstract class Wild
    {
    }
```

Here are my abstract classes. Within my Captured class is a method battle that will activate and, whichever wild Pokemon it is paired with, it will battle that wild product. These are both abstract classes because they are meant to be used as bases for other things.

Product 1A

```
    public class Charamander : Captured
    {
        //need to make a public variable that can be used by the classes

        public override string Battle(Wild w)
        {
            return this.GetType().Name + " battles " + w.GetType().Name;
        }
    }
```

Product 2A

```
    public class Dratini : Wild
    {
    }
```

Product 1B

```
    public class Cindaquill : Captured
    {
        public override string Battle(Wild w)
        {
            return this.GetType().Name + " battles " + w.GetType().Name;
        }
    }
```

Here are my products, one for either of my concrete factories that cross with my abstract products, resulting in four total products. For both of my captured products, my method for Battle is further described and, the method itself is overridden.

Product 2B

```
public class Mareep : Wild
{
}
```

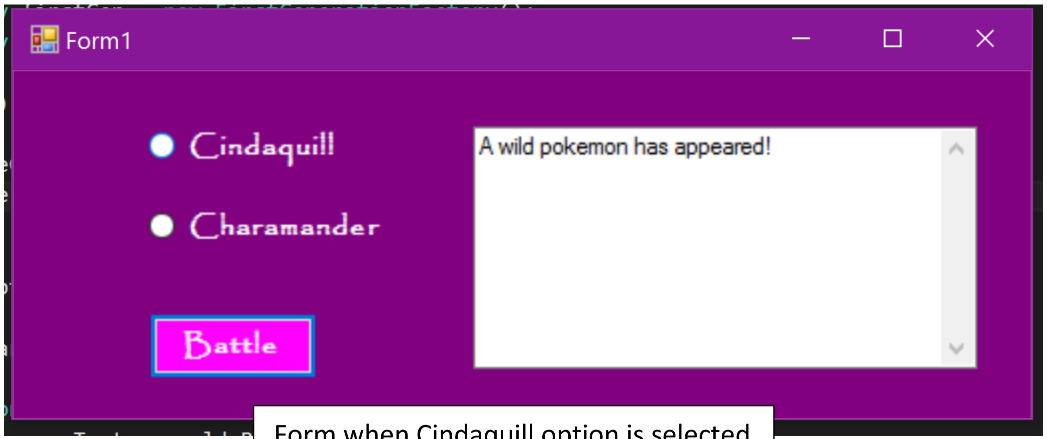
Form

```
public partial class Form1 : Form
{
    PokemonFactory firstGen = new FirstGenerationFactory();
    PokemonFactory secondGen = new SecondGenerationFactory();

    public Form1()
    {
        InitializeComponent();
        tb_Outcome.Text = "A wild pokemon has appeared!";
    }

    private void btn_Battle_Click(object sender, EventArgs e)
    {
        if (rb_Charamander.Checked == true)
        {
            PokemonWorld world = new PokemonWorld(firstGen);
            tb_Outcome.Text = world.Battle();
        }

        else if (rb_Cindaquill.Checked == true)
        {
            PokemonWorld world = new PokemonWorld(secondGen);
            tb_Outcome.Text = world.Battle();
        }

    }
}
```

This is my form class which first creates two new factories, one for First Generation and the other for the Second Generation. Then, when the form loads, it will say that a wild pokemon has appeared. Also, depending on which Pokemon you pick, the designated code will run and one will battle the wild Pokemon that has appeared.

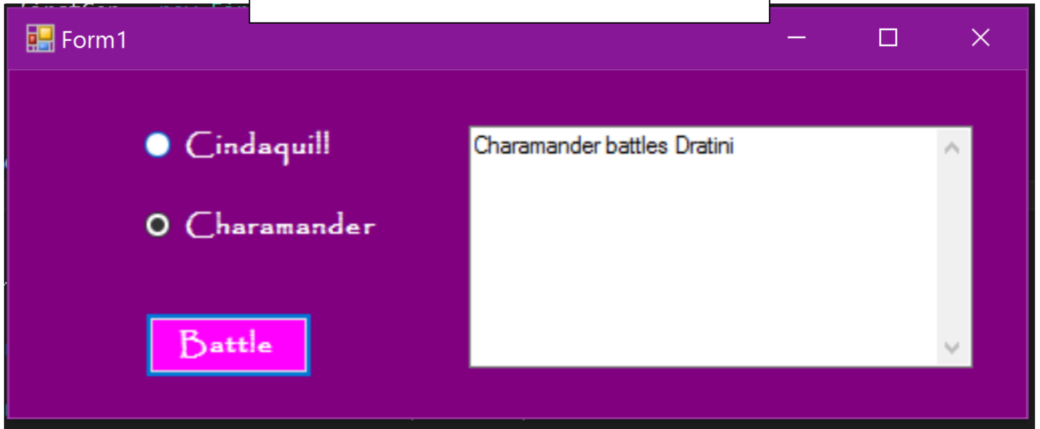
Screen Shots

Form on load



A wild pokemon has appeared!

Form when Cindaquill option is selected



Charamander battles Dratini

Observations

I thought this pattern went very well. I really liked seeing how this pattern worked and it was probably one of my favorites. The capabilities that this pattern has are endless and I think it would be really interesting to see other examples of it.