Nathan Bemus

Factory Method Pattern

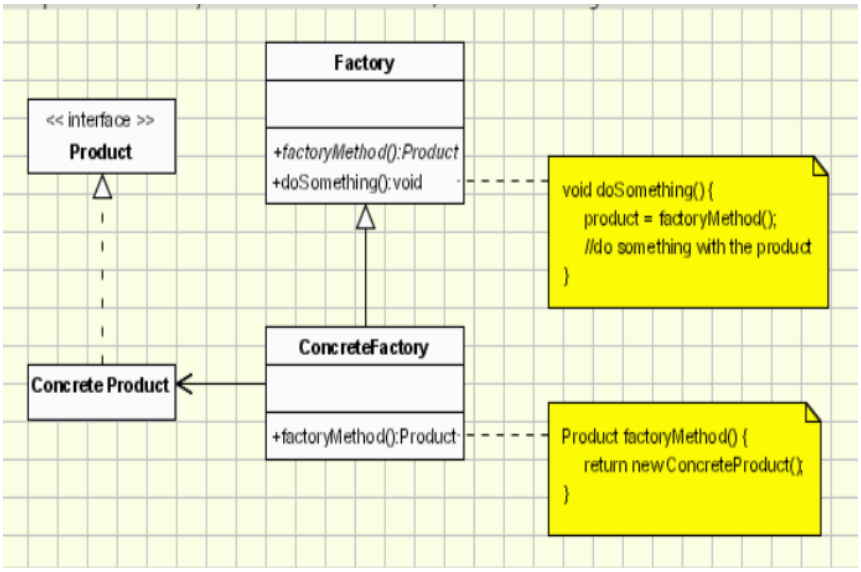
Design Patterns

9/17/16

Factory Method Pattern

Introduction

The purpose of this pattern is to provide an interface for creating objects, but letting subclasses decide which class to instantiate. This particular program demonstrates the factory method pattern by creating a Pokémon maker app. The program follows the UML diagram.



UML Diagram Analysis

The diagram shows that the factory method pattern is divided into four different classes. In it lies the Factory Class, a ConcreteFactory Class, the ConcreteProduct Class, and the Product Class. The client accesses the ConcreteFactory class and that class inherits functions from the Factory Class. The ConcreteFactory Class decides which product to create.

Program Walkthrough

Factory Class

This Class shows what is required to create a Pokémon no matter what type of Pokémon it is.

```
public abstract class Factory
    {
        public void createPokemon(string type, string name, string
ability)
        {
            Pokemon pokemon = createNewPokemon(type, name, ability);
            pokemon.createPokemon();
        }

        public abstract Pokemon createNewPokemon(string type, string
name, string ability);
    }
```

Pokémon Class

The Pokemon Class is used to represent a Pokemon without specifications.

```
public interface Pokemon
{
    string getPokemonName();
    string getPokemonType();
    string getPokemonAbility();
    void createPokemon();
}
```

ConcreteFactory

The Concrete Factory class determines what kind of Pokémon to create based off of the type of Pokémon. If there isn't a type the method returns null. The class inherits the methods from the Factory Class.

```
class ConcreteFactory : Factory
    {
        public override Pokemon createNewPokemon(string type,
string name, string ability)
        {

            if (type == "Fire")
            {
                return new Fire(name);
            }
            else if (type == "Water")
            {
                return new Water(name);
            }
            else if (type == "Grass")
            {
                return new Grass(name);
            }

            return null;
        }
    }
```

Water Class (product class)

This is the class that creates a product. The Water class inherits methods from the Pokémon Class. The Water class is very similar to the other two product classes. The class instantiates three different variables that are passed to create the Pokémon (product).

```
class Water : Pokemon
    {

        private string name;
        private string pokemonType = "Water";
        private string ability = "Torrent";

        public Water(string name)
        {
            this.name = name;
        }

        public void createPokemon()
        {
            MessageBox.Show("Pokémon Name: " + name + '\n' + "Pokémon
Type: " + pokemonType +
                  '\n' + "Pokémon Ability: " + ability, "Pokémon
Created");
        }

        public string getPokemonAbility()
        {
            return ability;
        }

        public string getPokemonType()
        {
            return pokemonType;
        }

        public string getPokemonName()
        {
            return name;
        }
    }
```

Form1

This is the main form where the program executes. The form creates a concrete factory object and calls it pokemonCF.When the application runs, the user types into the text box on the interface to instantiate the name. The user then has to select a Pokémon type and clicks on the create button. A message box appears with the newly created Pokémon with all of the specifications the user asked for.

```
public partial class Form1 : Form
    {

        public string PokemonName;
        public string PokemonType;
        public string PokemonAbility;

        ConcreteFactory pokemonCF = new ConcreteFactory();

        public Form1()
        {
            InitializeComponent();
        }

        private void rbnFireType_CheckedChanged(object sender,
EventArgs e)
        {
            rbnGrassType.Checked = false;
            rbnWaterType.Checked = false;

            PokemonType = "Fire";
            PokemonAbility = "Flash Fire";
        }

        private void rbnWaterType_CheckedChanged(object sender,
EventArgs e)
        {
            rbnFireType.Checked = false;
            rbnGrassType.Checked = false;
            PokemonType = "Water";
            PokemonAbility = "Torrent";

        }

        private void rbnGrassType_CheckedChanged(object sender,
EventArgs e)
        {
            rbnFireType.Checked = false;
            rbnWaterType.Checked = false;
            PokemonType = "Grass";
            PokemonAbility = "Overgrow";

        }

        private void btnCreatePokemon_Click(object sender,
EventArgs e)
        {
            pokemonCF.createPokemon(PokemonType,
tbxPokemonName.Text, PokemonAbility);
        }

    }
```

Reflection

The pattern uses a factory class to create a product determined by the subclasses. This is shown in the form of an application that creates Pokémon.

Conclusion

This particular assignment was initially frustrating to program, but now that it is complete isn't as bad to comprehend. I still have issues with some of the concepts and even how this could be used in a real life scenario. Overall it

was a project that I wasn’t excited for but I am not disappointed that I completed the project.