Jackson Spencer

The Template Method Pattern

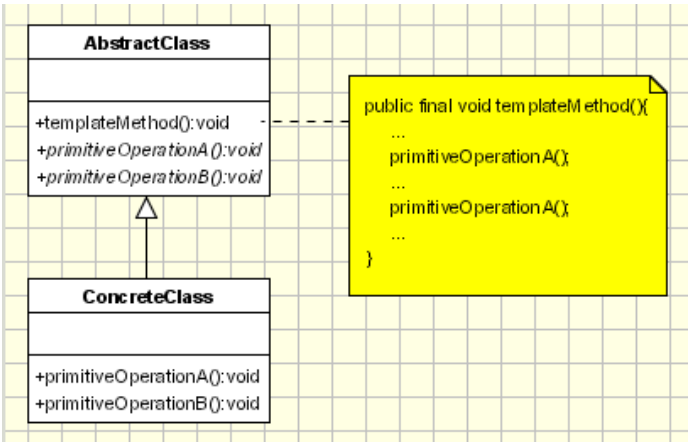
December 2, 2016

ECCS 2411: Design Patterns

Introduction

The purpose of this pattern is to define the skeleton of an algorithm in an operation, deferring some steps to subclasses. The subclasses are allowed to redefine certain steps of an algorithm without letting them change the structure of the algorithm. My application demonstrates this as selecting a very expensive vehicle from a leisure vehicle lot. The user can choose either a sportcar, helicopter, or speedboat.

UML Diagram



The UML diagram for the Template Method pattern shows the necessary components and functions. My application contains an AbstractClass class called Transport, and three ConcreteClasses called Boat, Chopper, and Sportcar.

Classes

The Abstract class contains the Template method, chooseVehicle(). It defines strings for the methods used in the concrete classes. The chooseVehicle() method returns the completed string using the strings returned from the concrete classes.

```
public abstract class Transport
    {
        public abstract string vehicleType();
        public abstract string vehicleEnv();
        public string vehicle;
        public string environment;

        public string chooseVehicle()
        {
            vehicle = vehicleType();
            environment = vehicleEnv();
            return vehicle + "Now you will commandeer it " + environment;
        }

    }
```

The Concrete classes are very similar to each other. Each one inherits from the Transport abstract class, using the abstract string methods to define the type of vehicle and the environment in which it will be operated.

Sportcar

```
public class Sportcar : Transport       // a concrete class
    {
        public override string vehicleType()
        {
            return "You have left the lot in a Ferrari 458. ";
        }

        public override string vehicleEnv()
        {
            return "across the greatest roads on the Earth!";
        }
    }
```

Boat

```
public class Boat: Transport        // a concrete class
    {
        public override string vehicleType()
        {
            return "You have left the lot with a large speedboat. ";
        }

        public override string vehicleEnv()
        {
            return "across the Great Lakes (weather permitting)!";
        }
    }
```

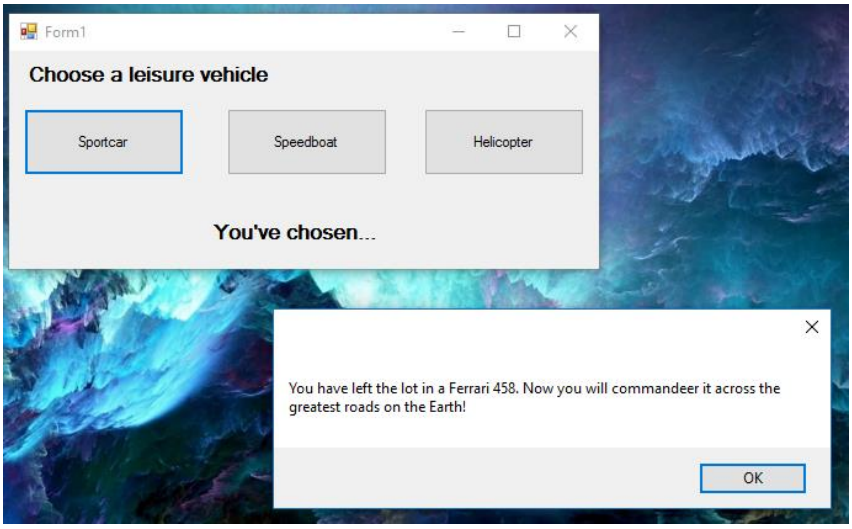
Chopper

```
public class Chopper: Transport
    {
        public override string vehicleType()
        {
            return "You have left the lot in a helicopter. ";
        }

        public override string vehicleEnv()
        {
            return " mightily across the skies!";
        }
    }
```

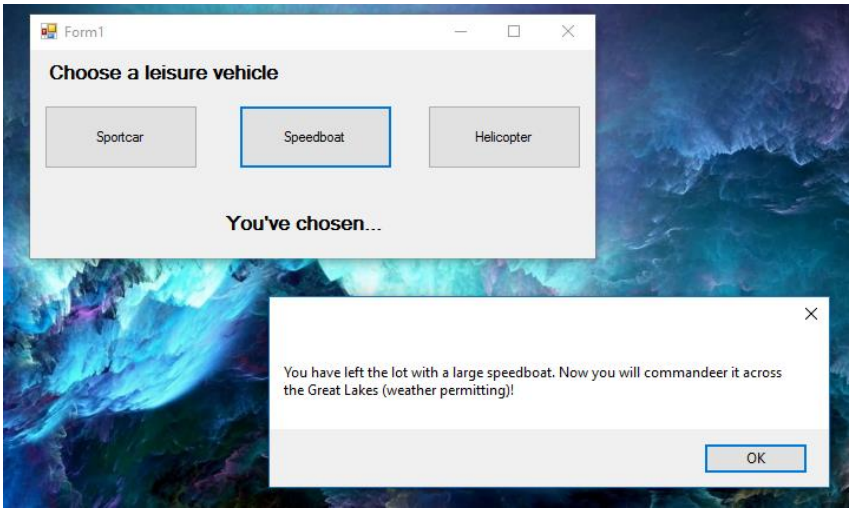
The form adds functionality to the controls. Each button uses the concrete class corresponding to the object it represents.

```
public partial class Form1 : Form
    {
        Transport vehicle;

        public Form1()
        {
            InitializeComponent();
        }

        private void btn_Car_Click(object sender, EventArgs e)
        {
            vehicle = new Sportcar();
            MessageBox.Show(vehicle.chooseVehicle());
        }

        private void btn_boat_Click(object sender, EventArgs e)
        {
            vehicle = new Boat();
            MessageBox.Show(vehicle.chooseVehicle());
        }

        private void btn_Chopper_Click(object sender, EventArgs e)
        {
            vehicle = new Chopper();
            MessageBox.Show(vehicle.chooseVehicle());
        }
    }
```

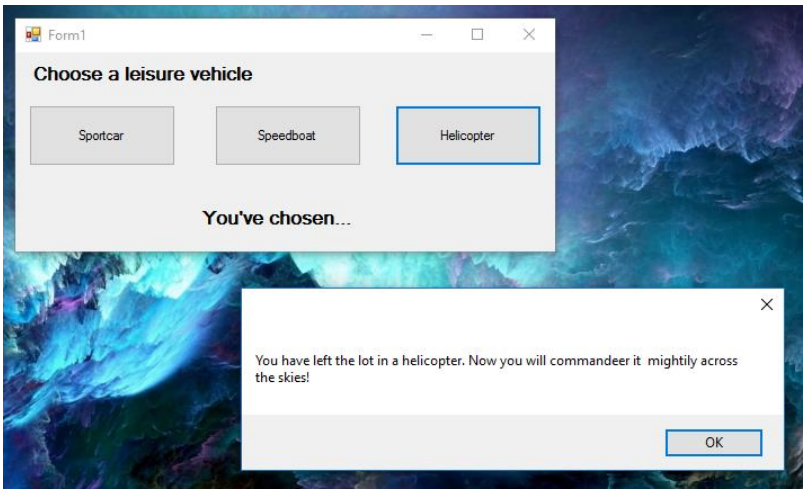
Sportcar



Boat



Helicopter



Reflection

The most difficult part of this pattern for me was to come up with an idea for a demo. The development of the application was simple. The UML diagram was very easy to understand, which I believe helped immensely.