Bridge Pattern Assignment

Design Patterns

Jacob Hartman

6 November 2016

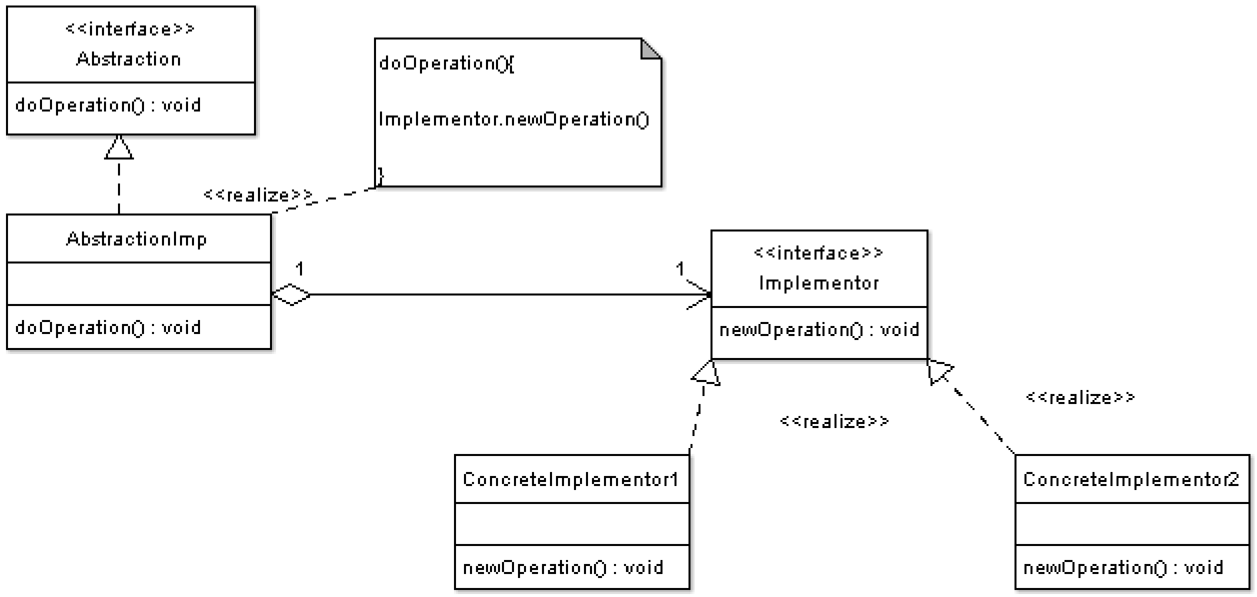
Introduction

This assignment requires the creation of an application that demonstrates the bridge pattern. The bridge pattern is used when an abstraction needs to have more than one implementation. Meaning, a particular structure for a class can be the same, but the actual content of its methods needs to be different.

The application I wrote for this assignment is a simulator for a medieval army. The general may assign to a contingent of soldiers either a raid captain or a guard captain. These captains will take different actions when commanded to advance.

UML Diagram

image taken from http://www.oodesign.com/bridge-pattern.html



Above is the UML Diagram for the bridge pattern. The first item it requires is an interface called Abstraction, with a method called doOperation. A class called AbstractionImp implements this interface, and it contains an Implementor object. The AbstractionImp class's doOperation method must execute the Implementor's newOperation method. The Implementor is an interface with a newOperation method. Two classes, ConcreteImplementor1 and ConcreteImplementor2 are required to implement the Implementor interface.

I created an Abstraction interface called General, and my AbstractionImp class is FieldGeneral. For the Implementor interface, I wrote the Captain class. RaidCaptain and GuardCaptain are my ConcreteImplementor classes. The doOperation method is command, and the newOperation method is followOrders.

<u>Narrative</u>

I began by writing the Captain interface to fill the role of the Implementor.

```
public interface Captain // this is the Implementor interface
{
    string followOrder(); // this is the newOperation method
}
```

This interface merely contains a signature for the followOrder method, which returns a string. I then created the ConcreteImplementors, which implement the Captain interface.

```
public class RaidCaptain : Captain // this is the ConcreteImplementor1 class
{
    public string followOrder()
    {
        return "We are advancing to take more territory.";
    }
}
```

The RaidCaptain class represents a captain ready to move forward and attack the enemy. Its followOrder method returns a string pertaining to that action. In a real world scenario, the contingent would advance when this method is called.

```
public class GuardCaptain : Captain // this is the ConcreteImplementor2 class
{
    public string followOrder()
    {
        return "We are advancing to defend this area.";
    }
}
```

The GuardCaptain class represents a captain prepared to defend the army's current holdings. Again, calling the followOrder method in a real scenario would cause the contingent to advance to a location to defend. Once these were created, I wrote the General interface.

```
public interface General // this is the Abstraction interface
{
    void callRaider();
    void callGuard();
    string command(); // this is the doOperation method
}
```

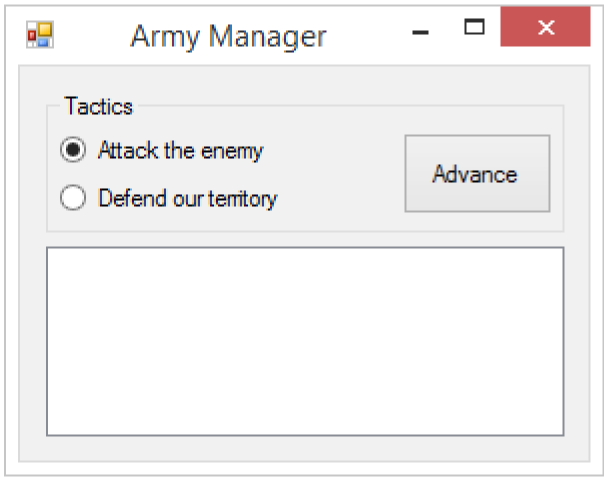
The General interface declares the signatures for three methods named callRaider, callGuard, and command. The command method is the doOperation method. This interface is implemented by the FieldGeneral class.

```
    public class FieldGeneral : General // this is the AbstractionImp class
    {
        Captain captain;

        public FieldGeneral()
        {
            captain = new RaidCaptain();
        }

        public void callRaider()
        {
            captain = new RaidCaptain();
        }

        public void callGuard()
        {
            captain = new GuardCaptain();
        }

        public string command()
        {
            return captain.followOrder();
        }
    }
```

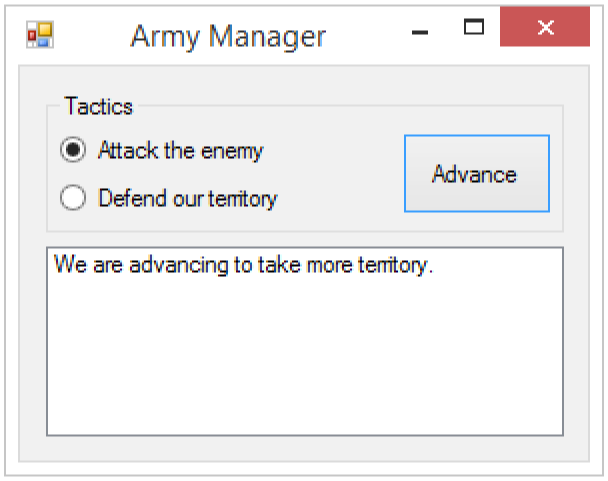
The FieldGeneral class represents a General who is out on the field, in a position where he can easily order troops. This class contains a Captain object. When the class is instantiated, a new RaidCaptain is created as the captain object. Calling the callRaider method will place a RaidCaptain in the Captain object, and calling the callGuard method will place a GuardCaptain in the Captain object. The command method will execute the Captain object's followOrder method and return the captain's words on the issue. This code is utilized by the main form.

```
    public partial class Form1 : Form
    {
        General general;
        public Form1()
        {
            InitializeComponent();
            general = new FieldGeneral();
        }

        private void rb_attack_CheckedChanged(object sender, EventArgs e)
        {
            if (rb_attack.Checked)
            {
                general.callRaider();
            }
        }

        private void rb_defend_CheckedChanged(object sender, EventArgs e)
        {
            if (rb_defend.Checked)
            {
                general.callGuard();
            }
        }

        private void button1_Click(object sender, EventArgs e)
        {
            lb_output.Items.Insert(0, general.command());
        }
    }
```

The form has a pair of radio buttons for selecting a tactic, a button to command the contingent to advance, and a list box for feedback from the captains. The class contains a General object containing a FieldGeneral, who can issue commands to the captains. When a radio button is selected, the proper method is called in the General object so that the appropriate captain is standing by. When the advance button is clicked, the general issues the order, and the feedback from the captain is placed in the list box at the top. Below are some screenshots of the application in use.

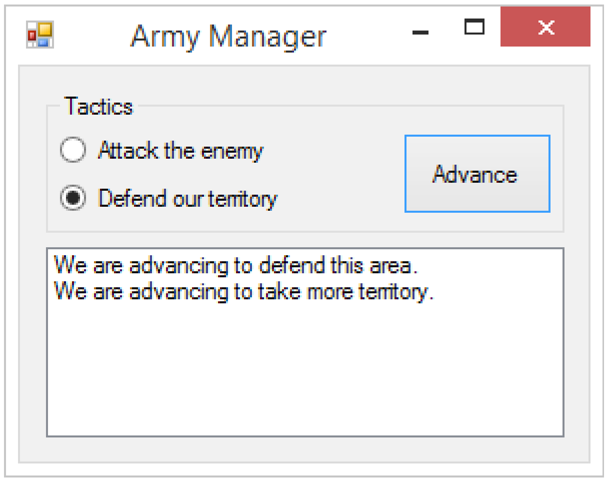


The initial view of the application.



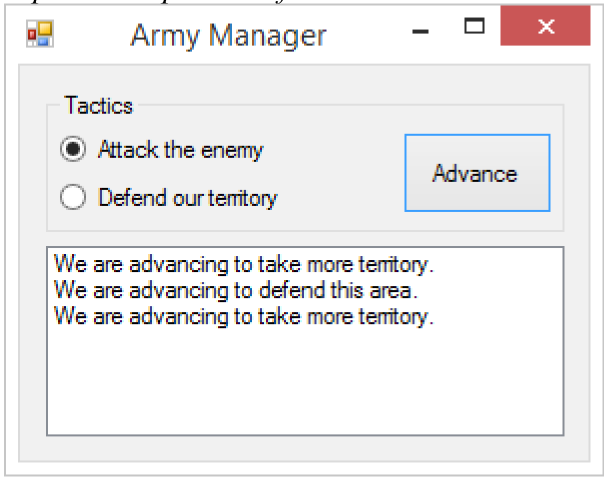
```
We are advancing to take more territory.
```

Hitting Advance issues the order to the captain and provides feedback below.



```
We are advancing to defend this area.
We are advancing to take more territory.
```

The defensive tactic can be selected with the radio buttons.



```
We are advancing to take more territory.
We are advancing to defend this area.
We are advancing to take more territory.
```

The offensive tactic can be reselected.

<u>Observations</u>

The bridge pattern is simple but intriguing. It allows class structures to be implemented in various ways, assisting in data encapsulation and promoting polymorphism. It could also potentially prove useful in the implementation of legacy code. I think that my submission fits the assignment well. I followed the UML Diagram closely in creating my application. The code displays a simple use of the bridge pattern. This assignment was fairly simple, but it reinforced the basic concept of polymorphism.