Alex Lang

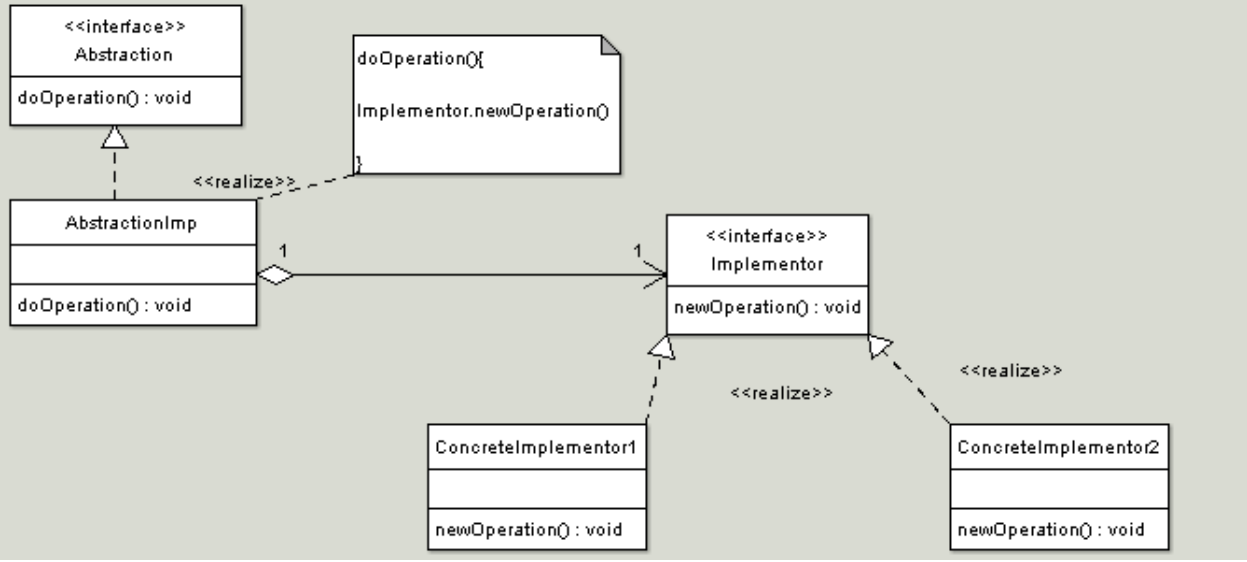
The Bridge Pattern

Design Patterns

11/8/16

For this assignment, we were told to do the Bridge Design Pattern. The Bridge Pattern decouples abstraction from implementation so that the two can vary independently. Basically, this pattern takes advantage of polymorphism, and implements my abstraction in a few different ways.

Here is the UML diagram from OODesign on what the Bridge Pattern consists of and its implementation.



For my application, I decided to make a typical grocery store checkout machine. In this application, the user has three buttons to choose from, cash, debit and credit, some of the most common ways to pay. When the user selects one of the payment options, the dialog box is updated and explains which button you have pressed and shows a brief description.

Starting off I created the Abstraction class. This class is my abstract Abstraction class.

```
public abstract class Abstraction
    {
        public abstract string selectPayment();
    }
```

This class is very brief, but does all it needs to do with the selectPayment method. The select payment method returns a string. I created selectPayment because the user MUST select a payment option when checking out.

Next, I created the AbstractionImp class.

```
public class AbstractionImp : Abstraction
    {
        public Payment payment;

        public AbstractionImp(Payment payment)
        {
            this.payment = payment;
        }
        public override string selectPayment()
        {
            return payment.completePayment();
        }
    }
```

This class inherits from abstraction, and implements selectPayment. First, I declare the Payment object. The Payment object is my abstract Implementor from the UML diagram. The AbstractionImp class is created when a Payment object is passed in through the constructor.

The selectPayment method calls the completePayment method from the abstract Payment object that is passed into it.

Next, I created the abstract Payment class.

```
public abstract class Payment
    {
        public abstract string completePayment();
    }
```

This class declares that there is a completePayment method that returns the string. This is here since you must complete a payment to check out at a store.

Finally, I created my concrete implementors which are Cash, Debit, and Credit.

```
public class Debit : Payment
    {
        public override string completePayment()
        {
            return "You pay using Debit. Thank you have a nice day.";
        }
    }

public class Cash : Payment
    {
        public override string completePayment()
        {
            return "You pay using Cash. Please insert cash below.";
        }
    }

public class Credit : Payment
    {
        public override string completePayment()
        {
            return "You pay using Credit. Thank you have a nice day.";
        }
    }
```

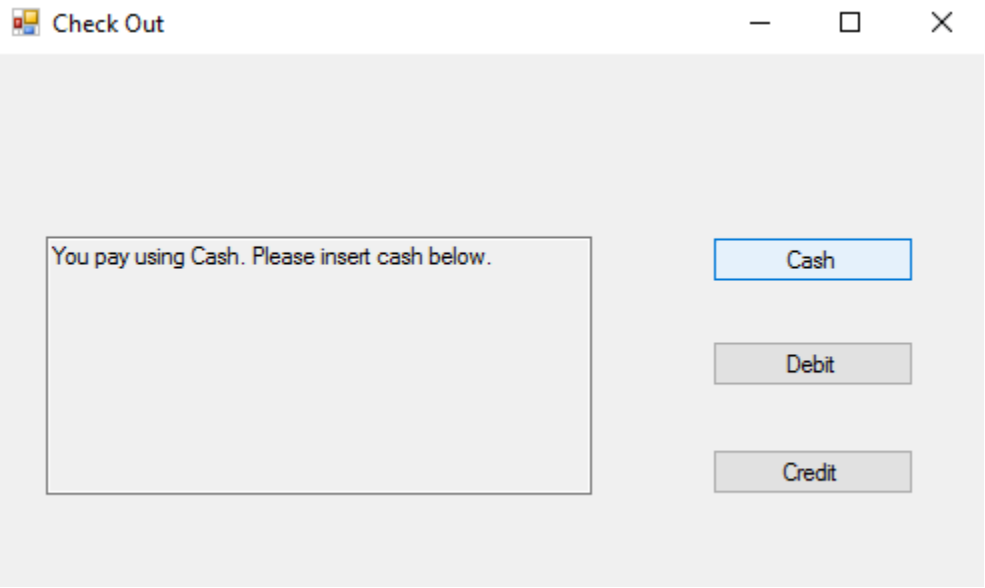
These classes inherit from Payment, and implement completePayment. Complete payment returns the string that best describes these payment options.

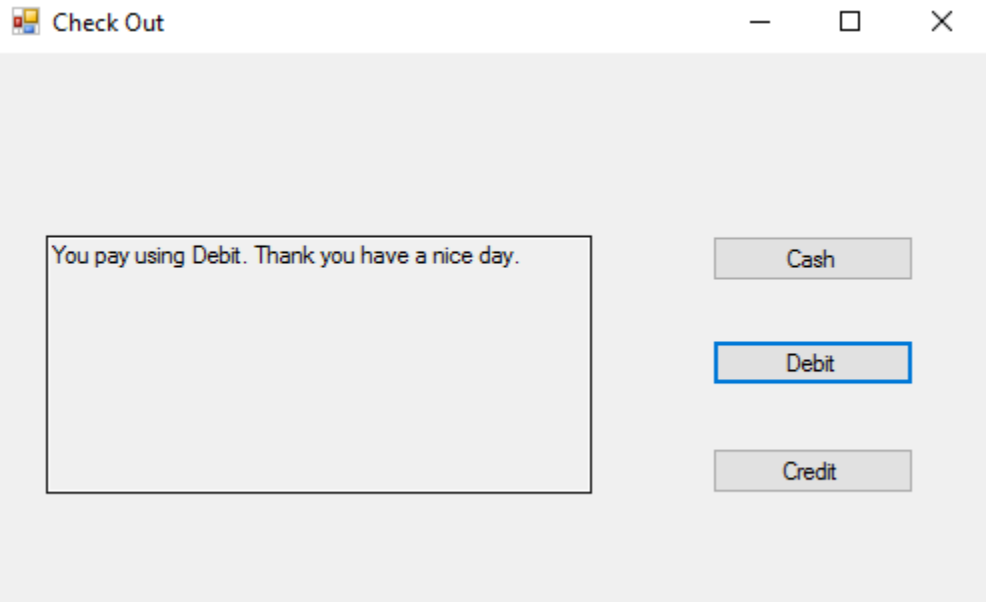
Finally, I put everything together in my form.

```
public partial class Form1 : Form
    {
        Abstraction NewChoice;

        public Form1()
        {
            InitializeComponent();
        }

        private void cashBtn_Click(object sender, EventArgs e)
        {
            NewChoice = new AbstractionImp(new Cash());
            displayBox.Text = NewChoice.selectPayment();
        }

        private void debitBtn_Click(object sender, EventArgs e)
        {
            NewChoice = new AbstractionImp(new Debit());
            displayBox.Text = NewChoice.selectPayment();
        }

        private void creditBtn_Click(object sender, EventArgs e)
        {
            NewChoice = new AbstractionImp(new Credit());
            displayBox.Text = NewChoice.selectPayment();
        }
    }
```

First I created my Abstraction object, NewChoice. It is named NewChoice since every time you click a button, you are making some new form of a choice, that can be defined in several different ways. When cashBtn is clicked, NewChoice is a new abstractionImp defined with the Cash characteristics. DebitBtn and creditBtn are the same with their respective objects. After that, the displayBox is updated to display the text corresponding to its text.

Screenshots of the app running:





Conclusion: This pattern was definitely pretty tricky thinking of an idea to implement it. Once I thought of my idea, I just had to think of a way to use it effectively. I can see the use of this pattern, but I really didn't come to appreciate it so much since I think it was overkill for this specific application. I'm sure I could find myself using it in plenty of instances in the future.