Decorator Pattern Assignment

Design Patterns

Jacob Hartman

17 November 2016

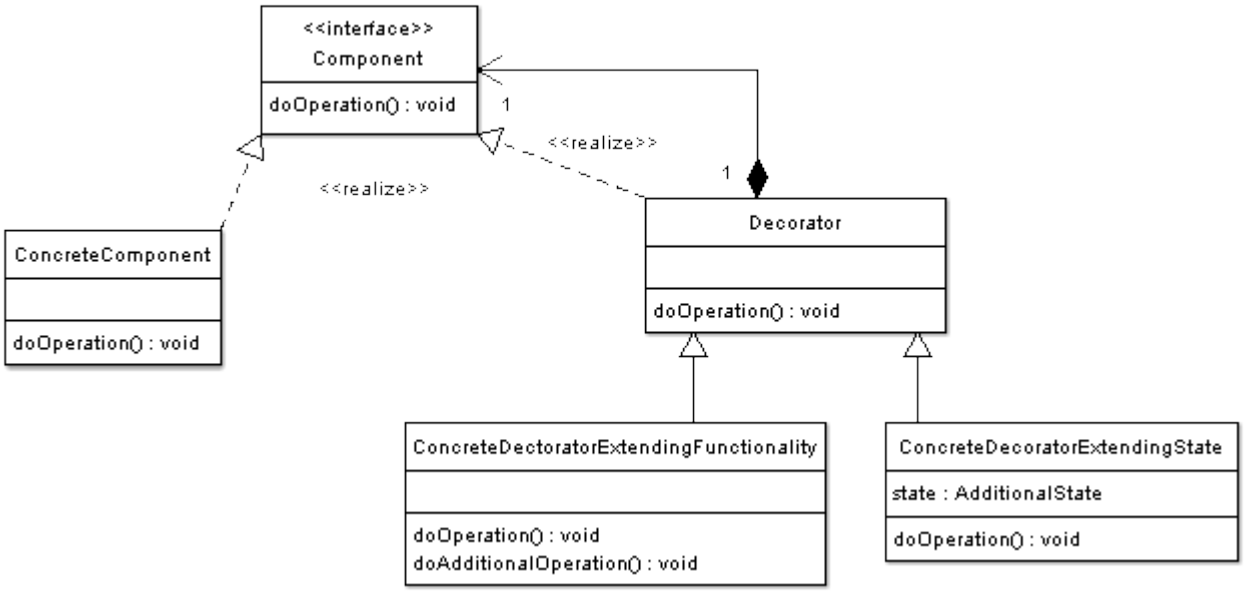
Introduction

This assignment requires the creation of an application that demonstrates the decorator pattern. The decorator pattern allows additional states and functionality to be added to an object after it is created.

the application I wrote for this assignment revolves around outfitting knights for battle. A recruited knight starts with a sword, but can be given a shield. The sword can also be replaced with a spear. Both cases alter the tactics available to the knight.

UML Diagram

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



Above is the UML diagram for the decorator pattern. It shows the need for a Component interface and two classes which implement it. The ConcreteComponent class implements the interface. The Decorator class also implements the interface, and it contains a Component object. The two ConcreteDecorator classes implement the functionality of the Decorator class.

My Component interface is called Soldier. My ConcreteComponent class is Swordsman. The ArmedKnight class fills the role of the Decorator class. Spearman is the ConcreteDecoratorExtendingFunctionality, and ShieldBearer is the ConcreteDecoratorExtendingState.

Narrative

I began by writing the Soldier interface.

```
public interface Soldier // this is the Component interface
{
    string attack();
}
```

The interface contains a signature for a single method called attack. This is the doOperation method. I then created the Swordsman class.

```
public class Swordsman : Soldier // this is the ConcreteComponent class
{
    public string attack()
    {
        return "The knight stabs forward with his weapon.";
    }
}
```

The Swordsman class implements the Soldier interface. It represents the default knight, one carrying a sword. The attack method returns a string stating the action he has taken. I then wrote the ArmedKnight class.

```
public abstract class ArmedKnight : Soldier // this is the Decorator class
{
    public Soldier fighter = new Swordsman();
    public string attack()
    {
        return fighter.attack();
    }
}
```

ArmedKnight is an abstract class implementing the Soldier interface. it contains a Swordsman object called fighter. The attack method falls through to the fighter object by default. Next, I wrote the Spearman class.

```
public class Spearman : ArmedKnight // this is the ConcreteDecoratorExtendingFunctionality
class
{
    public string attack()
    {
        return fighter.attack();
    }

    public string charge()
    {
        return "The knight charges forward with his spear.";
    }
}
```

The Spearman class implements the ArmedKnight class. Its attack method falls through to the fighter object the class inherits. To extend the functionality, a Spearman can also charge. This is a different action from a simple attack. I then wrote the ShieldBearer class.

```
    public class ShieldBearer : ArmedKnight // this is the ConcreteDecoratorExtendingState
class
    {
        bool defense = false;
        public string attack()
        {
            return "";
        }

        public string defend()
        {
            defense = true;
            return "The knight raises his shield in defense.";
        }

        public string dropDefense()
        {
            defense = false;
            return "The knight lowers his shield to attack.";
        }

        public bool isDefending()
        {
            return defense;
        }
    }
```

The ShieldBearer class extends the state of a knight. The shield can either be raised or lowered, represented by the defense boolean. The knight can be told to defend or to drop his defense, and his state can be checked. The new methods in the class accomplish this functionality. The rest of the code is found in the main form.

```
    public partial class Form1 : Form
    {
        List<Soldier> knightList;
        Soldier knight;
        int index;
        public Form1()
        {
            InitializeComponent();
            knightList = new List<Soldier>();
        }
```

The main form contains a Soldier object representing the currently selected knight. It also includes a list of Soldier objects to contain all the knights. This list is called knightList. The index variable contains the index of the currently selected knight.

```
        private void btn_recruit_Click(object sender, EventArgs e)
        {
            string name = tb_name.Text;
            if (name == "")
            {
                MessageBox.Show("You haven't entered a name for the new recruit!");
```

```
        }
        else if (lb_knights.Items.Contains(name))
        {
            MessageBox.Show("That knight has already been recruited!");
            tb_name.Text = "";
        }
        else
        {
            lb_knights.Items.Add(name);
            knightList.Add(new Swordsman());
            tb_name.Text = "";
        }
    }
```

There is a recruit button on the form. The form asks for a name to be entered. If no name is entered or the entered name already exists, a message dialog will inform the user of the error. Otherwise, a new knight is added to the list.

```
    private void lb_knights_SelectedIndexChanged(object sender, EventArgs e)
    {
        index = lb_knights.SelectedIndex;
        knight = knightList[index];
        btn_attack.Enabled = true;
        if (knight is Spearman)
        {
            btn_charge.Enabled = true;
            btn_defend.Enabled = false;
            btn_dropdefense.Enabled = false;
            btn_spear.Enabled = false;
            btn_shield.Enabled = false;
        }
        else if (knight is ShieldBearer)
        {
            btn_charge.Enabled = false;
            btn_defend.Enabled = true;
            btn_dropdefense.Enabled = true;
            btn_spear.Enabled = false;
            btn_shield.Enabled = false;
        }
        else
        {
            btn_charge.Enabled = false;
            btn_defend.Enabled = false;
            btn_dropdefense.Enabled = false;
            btn_spear.Enabled = true;
            btn_shield.Enabled = true;
        }
    }
```

When the user selects a knight in the list, the index of that knight is stored, and that knight object becomes the currently selected knight. The GUI is then updated based on the knight's equipment to allow the appropriate actions.

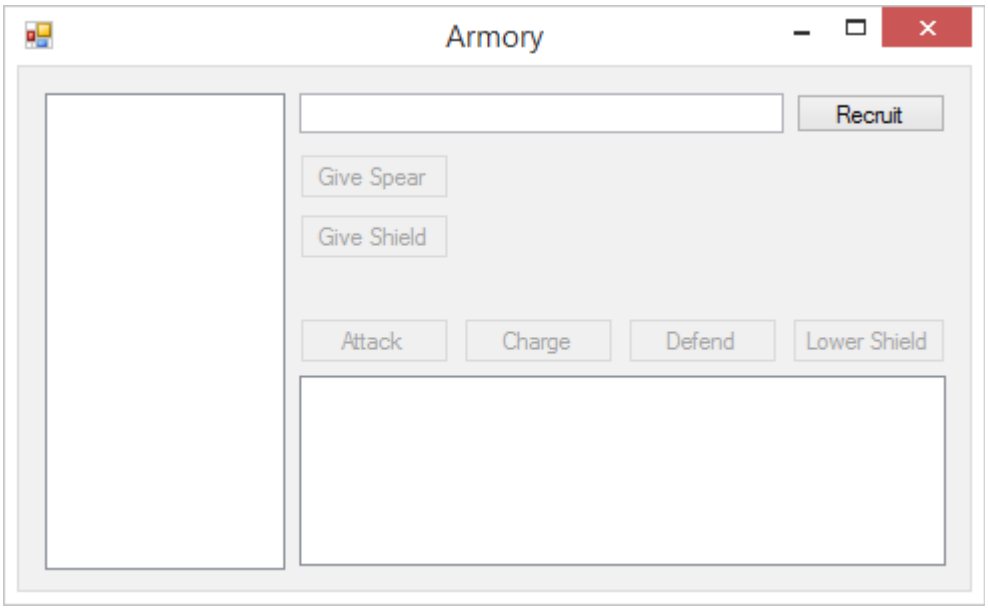
```
    private void btn_spear_Click(object sender, EventArgs e)
    {
        knightList[index] = new Spearman();
        lb_knights_SelectedIndexChanged(sender, e);
    }
```

```
        private void btn_shield_Click(object sender, EventArgs e)
        {
            knightList[index] = new ShieldBearer();
            lb_knights_SelectedIndexChanged(sender, e);
        }
```

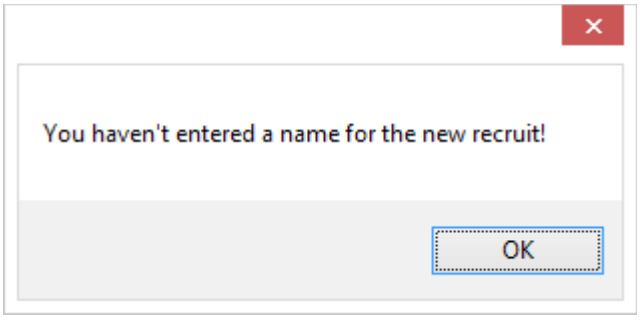
Two buttons allow the user to give a knight either a spear or a shield. The knight is decorated appropriately when a button is chosen, and the application reselects the knight to update the GUI.

```
        private void btn_attack_Click(object sender, EventArgs e)
        {
            lb_feedback.Items.Insert(0, knight.attack());
        }

        private void btn_charge_Click(object sender, EventArgs e)
        {
            Spearman spearman = (Spearman)knight;
            lb_feedback.Items.Insert(0, spearman.charge());
        }

        private void btn_defend_Click(object sender, EventArgs e)
        {
            ShieldBearer defender = (ShieldBearer)knight;
            lb_feedback.Items.Insert(0, defender.defend());
        }

        private void btn_dropdefense_Click(object sender, EventArgs e)
        {
            ShieldBearer defender = (ShieldBearer)knight;
            lb_feedback.Items.Insert(0, defender.dropDefense());
        }
    }
```

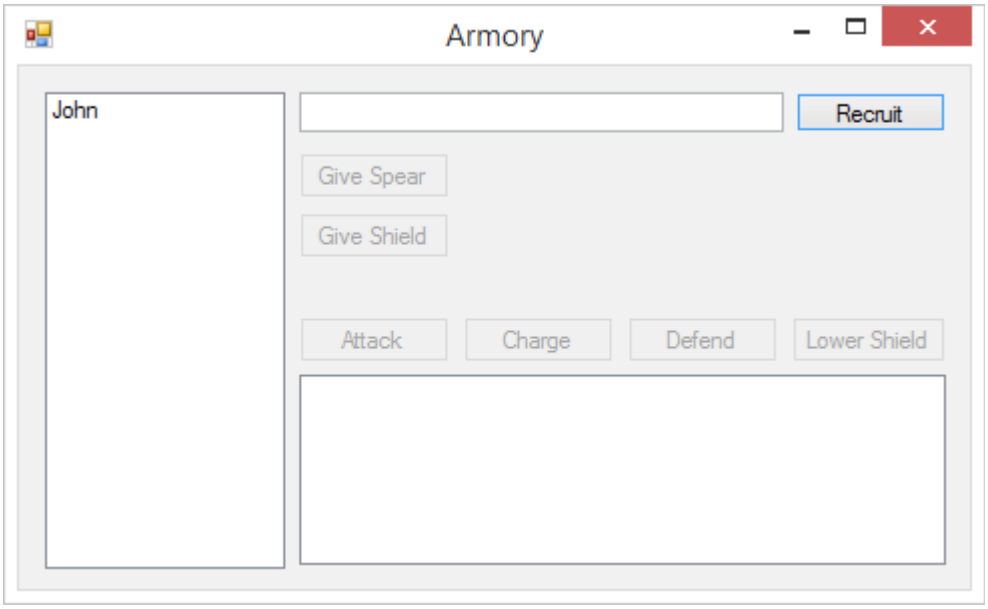
The rest of the buttons allow the user to command the selected knight to perform an action. Feedback from these actions is provided in a list box at the bottom of the form. Below are some screenshots of the program in use.



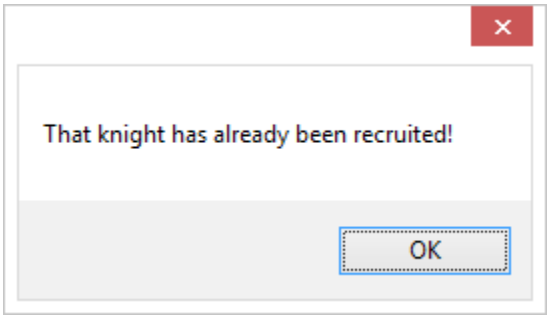
The initial view of the application.



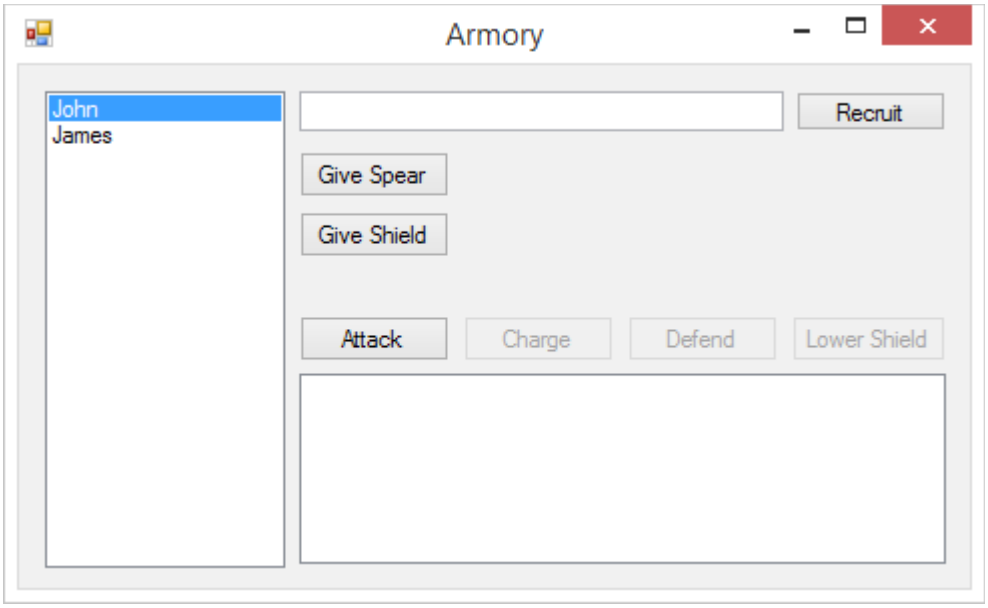
If the user clicks recruit without entering a name, this message is shown.



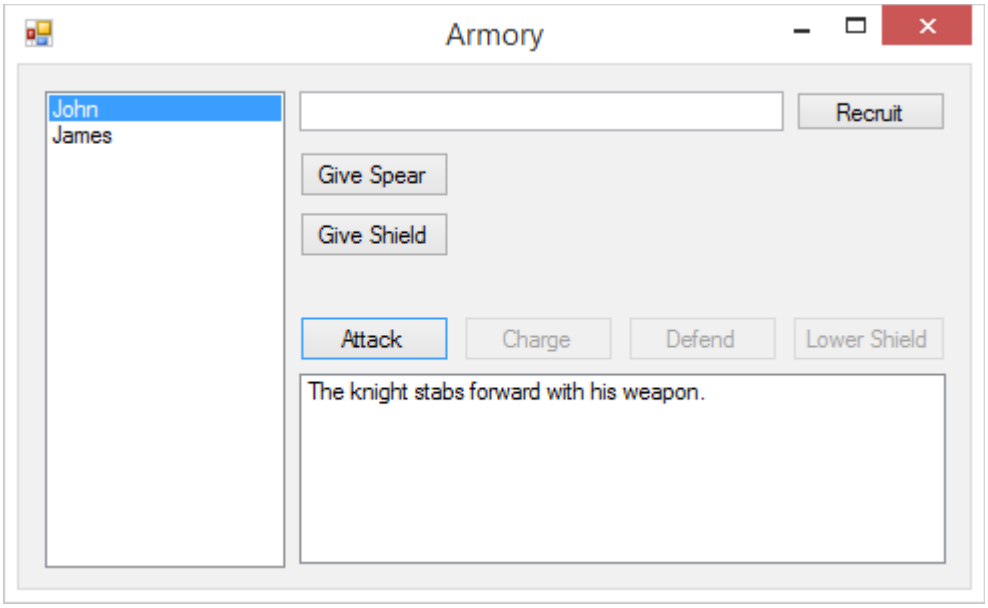
A knight named John is recruited.



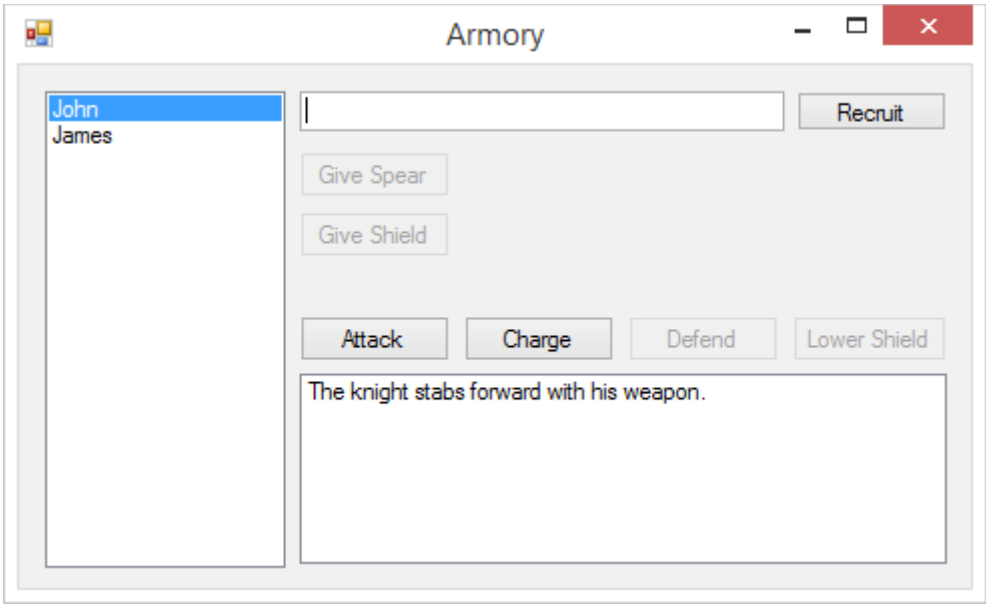
If a duplicate name is entered, this message is shown.



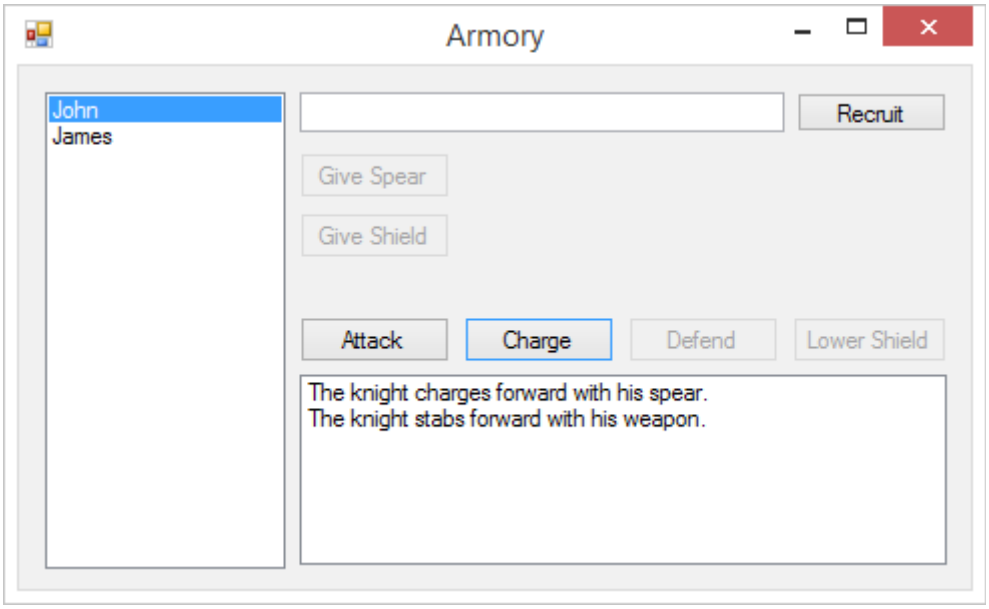
Multiple knights can be recruited. When a knight's name is clicked, the knight is selected and the proper buttons are enabled.



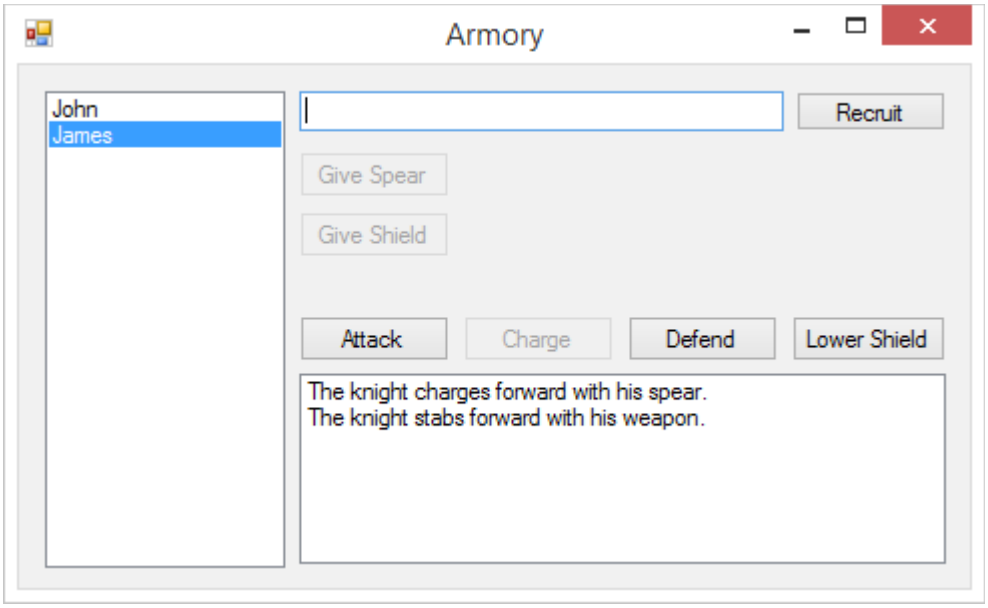
Clicking attack results in a simple attack.



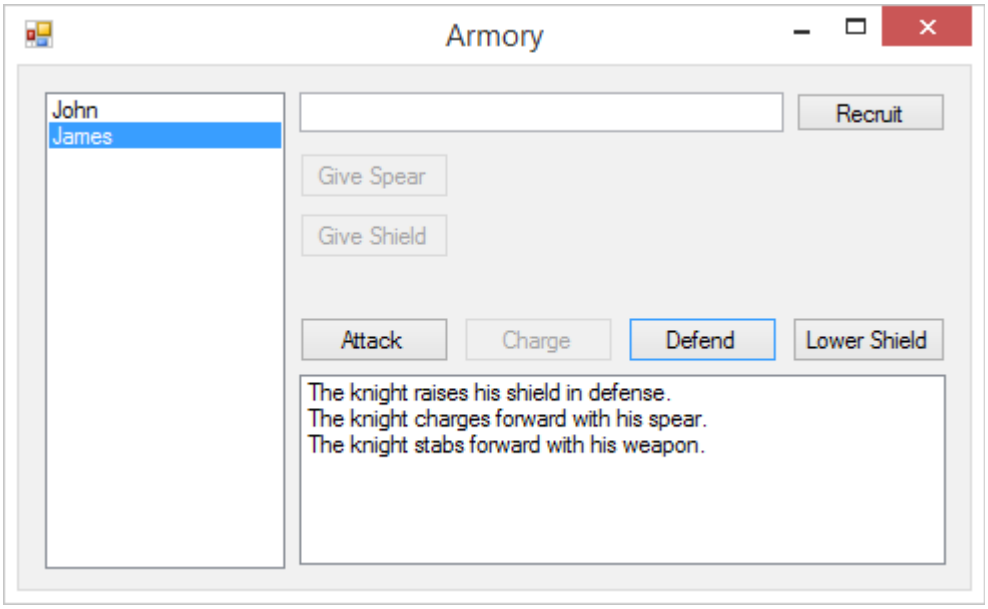
Once the knight is given a spear, the give buttons are disabled and the charge tactic becomes available.



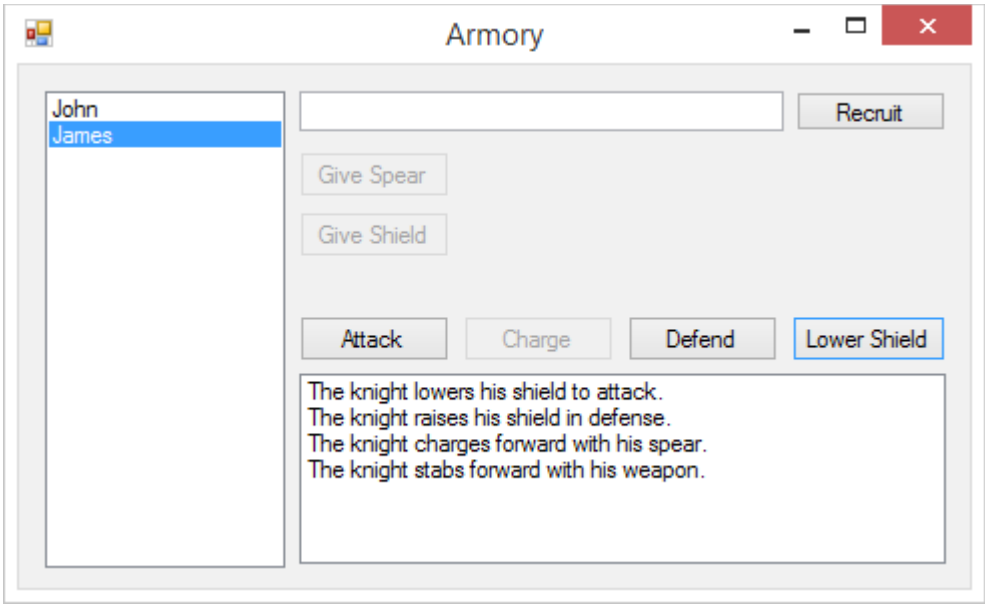
The charge tactic is an alternate attack.



The second knight is given a shield and can now defend.



Clicking defend makes the knight raise his shield.



Clicking lower shield tells the knight to stop defending.

Observations

The Decorator pattern is fairly similar to other patterns we've learned. It uses multiple layers of polymorphism, which is something I didn't expect. There are a lot of potential uses for it. The reuse of legacy code is one example, as the pattern can extend functionality or state to allow old code to be used in a new way. I feel my submission fits the assignment well. I followed the structure of the UML Diagram in the creation of my application. I think there's a good chance a pattern like this will be something I use in the future.