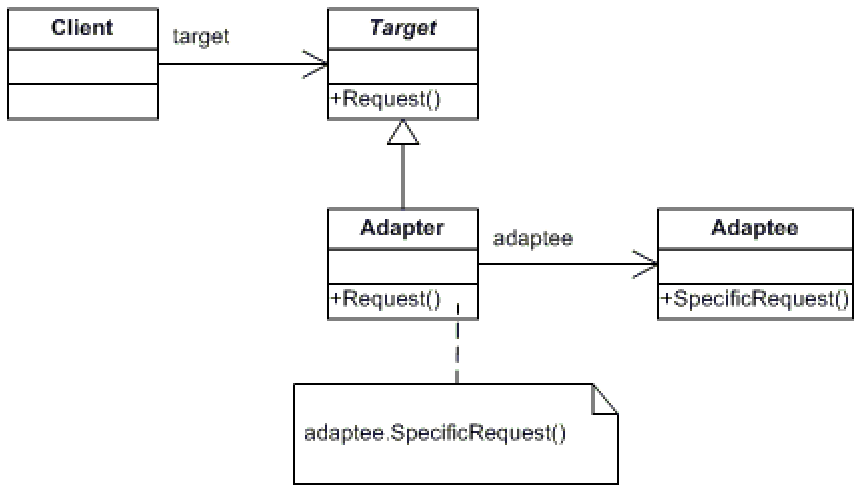
The Command and Adapter Patterns
Design Patterns
Andreya Grzegorzewski – Ohio Northern University
Fall 2016

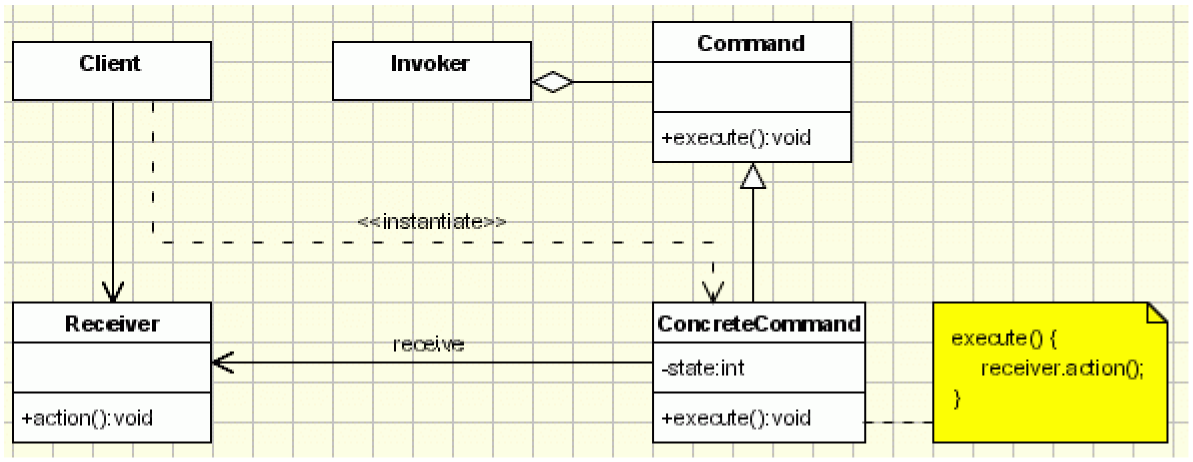
Introduction

This assignment requires the implementation of two design patterns: the command pattern and the adapter pattern. The command pattern uses command objects to perform actions so that the actions may be repeated or unexecuted. The adapter pattern transforms an unwanted interface into the interface that the client is expecting. My submission for this assignment is yet another human simulator, where the human can perform actions and then undo or redo those actions using the command pattern. The effects of the actions on the person's happiness and health are shown via method calls to a person class. The adapter pattern is implemented when "legacy code" for the person class is adapted into a usable interface via an adapter class.

The UML Diagrams



The UML diagram for the adapter pattern, provided at left courtesy of dofactory.com, shows the classes necessary in order to implement the adapter pattern. The adaptee class contains an interface that is not expected by the client, so the adapter class adapts the adaptee interface to the target interface so that the client can easily perform the desired operations.



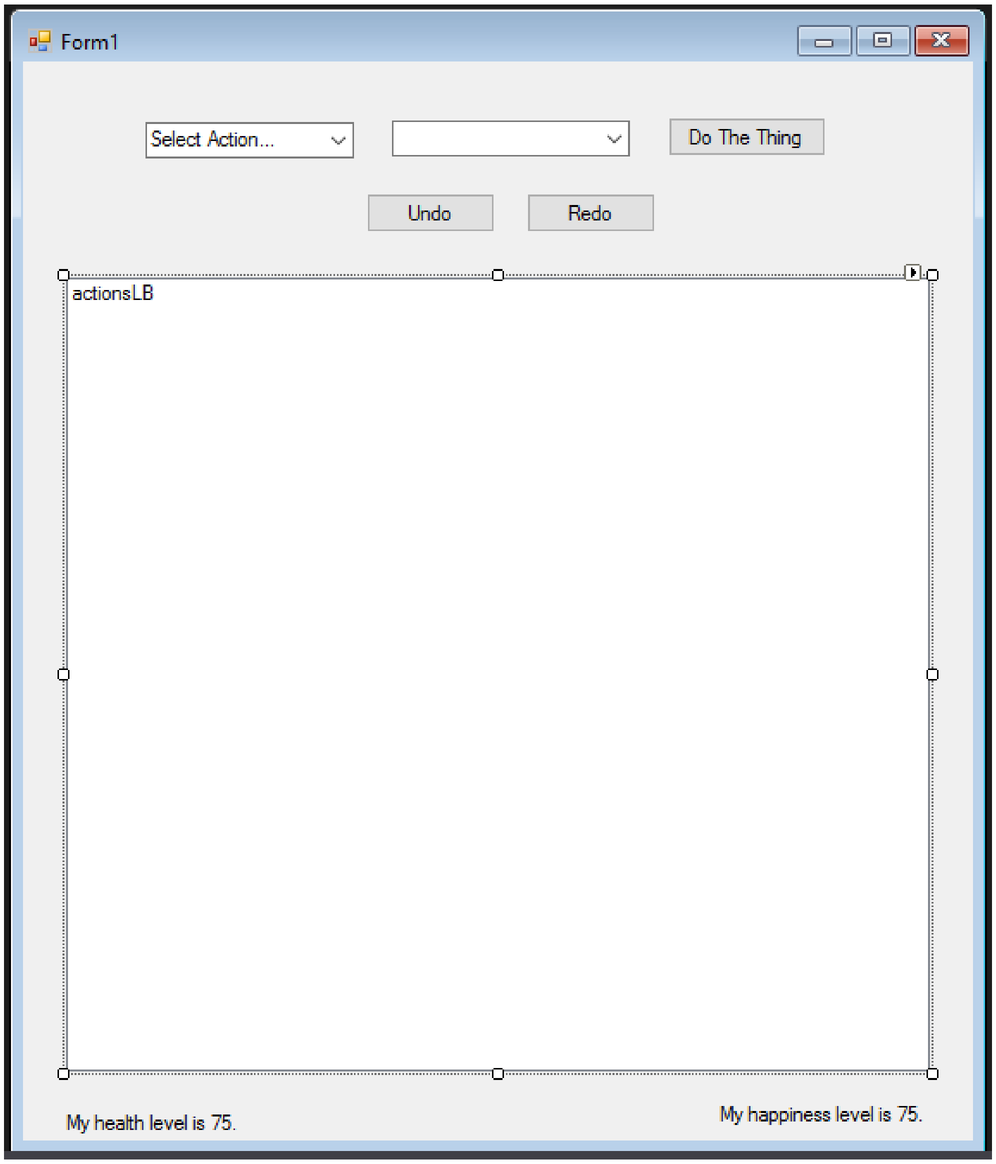
The UML diagram for the command class, shown above courtesy of oodesign.com, shows the classes necessary to implement the command pattern. The invoker class asks an object of the command class to carry out a request. The command class defines an interface for the concrete command class, which is actually capable of asking an object of the receiver class to perform some action. The client is used to create the concrete command object and set its receiver. The table below summarizes the classes I used to implement both of these patterns.

Client (Adapter)	My adapter pattern has two clients. One is the ConcreteCommand class from my command pattern. It needs to access the *Target* interface described below. The other is the code for my form, which calls two methods from the *Target* interface several times throughout the program.
Adaptee	My adaptee class is called LegacyPerson. It represents legacy code that is functional, but does not have the interface required by the ConcreteCommand class.
Adapter	My adapter class is called PersonAdapter. It creates a LegacyPerson object and uses functions defined in the LegacyPerson class to implement the methods provided in the *Target* class.
Target	My abstract *Target* class is called *NewPerson*. It defines methods called by the ConcreteCommand class but does not fully implement them. Most of the implementation for this class is taken care of in the Adapter class.
Invoker	The invoker class for my command pattern is my form. It creates ConcreteCommand objects when prompted by the user’s actions within the form.
Command	My abstract *Command* class is simply called *Command*. It provides two function prototypes for execute and unexecute methods.
ConcreteCommand	My ConcreteCommand class, called ConcreteCommand, implements the two functions defined by prototypes in the *Command* class.

Receiver	My receiver class is the *NewPerson* class, which is also the *Target* class described above. It receives commands and executes them by making use of the adapter class.
Client (Command)	The client of my command pattern is the code of my form. When my form is created, it creates pointers to new receiver and concrete command objects, and when the user performs certain actions, the concrete command object is instantiated and used.

Narrative and Code for the Form

I started my project by designing the form. I did this so that I would have a better conceptualization of what my program needed to accomplish moving forward. A snip of the form from the design view of Visual Studio is shown below.



The first combo box, which says “Select Action…,” allows the user to select one of three actions to perform. When that action is selected, the second combo box is populated with three different options regarding the action to be performed. For example, the user could select “Eat” from the first combo box and “Cake” from the second. When the user clicks the “Do The Thing” button, the thing is done and displayed in the actions list box. Additionally, the health and happiness labels at the bottom of the form are updated. Then, the undo button is enabled, and the command can be undone. Once at least one command has been undone, the redo button is also enabled. Once I knew what I wanted my form to do, I moved on to writing some code.

Narrative and Code

I started coding with the command pattern. I did this because it defines the interface required for the *Target* class of the adapter pattern, and starting by creating a required interface is the most true to the purpose of the adapter pattern. I started creating my command pattern by making the *Command* class, which provided the prototypes for two methods: execute and unexecute.

```
    public abstract class Command // This is the abstract command class
    {
        public abstract string execute();
        public abstract string unexecute();
    }
```

I then created my ConcreteCommand class:

```
   class ConcreteCommand : Command // This is the concrete command class
```

I created some variables that would be required for the implementation of the execute and unexecuted methods, and then created a constructor. The NewPerson object is the receiver of the commands; happy and healthy describe whether or not the action that the person is taking causes happiness and health; and text provides a textual description of the action.

```
    NewPerson person;
    bool healthy;
    bool happy;
    string text;

    public ConcreteCommand(NewPerson person, bool healthy, bool happy, string text)
    {
        this.person = person;
        this.healthy = healthy;
        this.happy = happy;
        this.text = text;
    }
```

After that, I implemented the execute and unexecuted methods, which are quite similar to one another. I did this by overriding the abstract methods provided in the *Command* class.

```
        public override string execute()
        {
            person.doThing(healthy, happy);
            text = getNewText();
            return text;
        }

        public override string unexecute()
        {
            person.doThing(!healthy, !happy);
            text = getNewText();
            return text;
        }
```

The doThing method, which will be described in more detail later in the adapter pattern code, has the person object do something which either increases or decreases the happiness and health levels of the person. While I was writing the code for these two methods, I realized that I needed a getNewText method to change the text when the action is undone or redone. For example, the person could choose to eat food, then later undo that action by regurgitating the food. The function is shown below:

```
        private string getNewText()
        {
            string oldText = this.text;
            string newText = "newText";

            if (oldText.Contains(" eat "))
                newText = oldText.Replace(" eat ", " regurgitate ");
            else if (oldText.Contains(" regurgitate "))
                newText = oldText.Replace(" regurgitate ", " eat ");
            else if (oldText.Contains(" do "))
                newText = oldText.Replace(" do ", " undo ");
            else if (oldText.Contains(" undo "))
                newText = oldText.Replace(" undo ", " do ");
            else if (oldText.Contains(" go to "))
                newText = oldText.Replace(" go to ", " return from ");
            else if (oldText.Contains(" return from "))
                newText = oldText.Replace(" return from ", " go to ");

            this.text = newText;
            return newText;
        }
```

This was the end of my code for the ConcreteCommand class. I then moved on to the NewPerson class, which is both the receiver for my command pattern method and the *Target* class for my adapter pattern.

```
public abstract class NewPerson // This is the receiver class for the command pattern and
       the target class for the adapter pattern
```

This class is short and straightforward. First, I defined a prototype for the doThing method that is called from the ConcreteCommand class:

```
public abstract int[] doThing(bool healthy, bool happy);
```

Then, I created two virtual methods for getting the descriptions of the person's happiness and health. The methods are virtual so that they can have some implementation while still being altered later on by the adapter class. The methods are shown below.

```
        public virtual string getHappyText()
        {
            return "My happiness level is ";
        }

        public virtual string getHealthyText()
        {
            return "My health level is ";
        }
```

Once I created the interface for my NewPerson class, I had to create the adaptee class, called LegacyPerson. In this class, I used many simple methods that could easily be condensed into one more helpful method. I did this so that the adapter class could later use these methods to provide the implementation of the *Target* class's interface in a straightforward manner. This is the class header:

```
class LegacyPerson // This is the adaptee class, which represents the poorly written
       legacy code of a person class.
```

I started by creating the variables and constants required for this class. I made variables to keep track of the person's happiness and health, and two constants to describe the amount that health and happiness are changed by when good and bad decisions are made.

```
int health = 75;
int happiness = 75;

const int GOOD_DECISION_ADDITION = 5;
const int BAD_DECISION_DETRACTION = -5;
```

Then, I quickly implemented several self-explanatory methods that were easy to write but provided a poor interface. I also provided getters for the health and happiness variables.

```
        public void doHealthyThing()
        {
            health += GOOD_DECISION_ADDITION;
        }

        public void doUnhealthyThing()
        {
            health += BAD_DECISION_DETRACTION;
        }

        public void doHappyThing()
        {
            happiness += GOOD_DECISION_ADDITION;
        }

        public void doUnhappyThing()
        {
            happiness += BAD_DECISION_DETRACTION;
        }

        public int getHealth()
        {
            return health;
        }

        public int getHappiness()
        {
            return happiness;
        }
```

With my adaptee and target classes implemented, I began to work on my adapter class, called PersonAdapter.

```
class PersonAdapter : NewPerson // This is the adapter class
```

I provided a LegacyPerson variable which I used to access the methods of the LegacyPerson class:

```
LegacyPerson lp = new LegacyPerson();
```

Then, I began to override the functions provided in the *NewPerson* interface. I started with the doThing method. This method returns the person’s health and happiness after the thing has been done.

```
public override int[] doThing(bool healthy, bool happy)
{
    if (healthy)
        lp.doHealthyThing();
    else
        lp.doUnhealthyThing();
    if (happy)
        lp.doHappyThing();
    else
        lp.doUnhappyThing();

    int[] healthyHappy = new int[2];
    healthyHappy[0] = lp.getHealth();
    healthyHappy[1] = lp.getHappiness();

    return healthyHappy;
}
```

Then, I finished the getters that I started in the *NewPerson* class. I used base to access the getter defined in the *NewPerson* class and then added to it by getting the appropriate values from the LegacyPerson class.

```
public override string getHealthyText()
{
    return base.getHealthyText() + lp.getHealth() + ".";
}

public override string getHappyText()
{
    return base.getHappyText() + lp.getHappiness() + ".";
}
```

With this code written, the only code I had left to write was the code for the form. I started writing the form by creating several variables. First, I made stacks to keep track of commands that have been executed.

```
Stack<ConcreteCommand> undoStack = new Stack<ConcreteCommand>();
Stack<ConcreteCommand> redoStack = new Stack<ConcreteCommand>();
```

Then, I created a pointer to a NewPerson and instantiated it as a PersonAdapter. I also created a pointer to a ConcreteCommand object which will be reused several times throughout the code.

```
NewPerson person = new PersonAdapter();
ConcreteCommand cc;
```

Finally, I made a two-dimensional array of Booleans. The values of these Booleans came from the options described in the combo boxes. For example, the first option is

“Eat cake,” which is not healthy, but is happy. Therefore, the values in the first set of curly braces are false and true.

```
bool[,] healthyHappy = new bool[,]
{
    { false, true }, { true, false }, { true, true },
    { true, false }, { false, true }, { true, true },
    { true, true },  { true, false }, { false, true }
};
```

With my variables created, I started to define what actions to take when events were fired in the forms. First, I implemented the method called when the user selects a new item from the action combo box. This method clears the items currently in the option combo box and replaces them with the values appropriate for the newly selected action. Then it ensures that the user can access the option combo box.

```
private void actionCB_SelectedIndexChanged(object sender, EventArgs e)
{
    int index = actionCB.SelectedIndex;
    optionCB.Items.Clear();

    switch (index)
    {
        case 0: // The user selected "Eat"
            optionCB.Text = "Select food...";
            optionCB.Items.Add("Cake");
            optionCB.Items.Add("Salad");
            optionCB.Items.Add("Strawberries");
            break;
        case 1: // The user selected "Do"
            optionCB.Text = "Select action to do...";
            optionCB.Items.Add("A workout");
            optionCB.Items.Add("Nothing all day");
            optionCB.Items.Add("A barrel roll");
            break;
        case 2: // The user selected "Go to"
            optionCB.Text = "Select a place to go...";
            optionCB.Items.Add("Sleep");
            optionCB.Items.Add("The dentist");
            optionCB.Items.Add("An ice cream shop");
            break;
        default:
            MessageBox.Show("actionCB.SelectedIndex = " + index + " in actionCB
                event method.");
            break;
    }
    optionCB.AllowDrop = true;
}
```

Next, I implemented the method called when the user clicks the Do The Thing button. This method instantiates the ConcreteCommand object created earlier with a command describing the action just taken by the person. It then executes the new command, puts it in the undo stack, adds a statement describing the command to the actions list box, and updates the health and happiness labels.

```
private void doActionButton_Click(object sender, EventArgs e)
{
    cc = null;
    string actionText = actionCB.GetItemText(actionCB.SelectedItem).ToLower();

    if (actionText == "eat")
        actionText = "regurgitate";
    else if (actionText == "go to")
        actionText = "return from";
    else if (actionText == "do")
        actionText = "undo";

    string commandText = "I will " + actionText + " " +
        optionCB.GetItemText(optionCB.SelectedItem).ToLower() + ".";
    undoButton.Enabled = true;

    switch (optionCB.GetItemText(optionCB.SelectedItem))
    {
        case "Cake":
            cc = new ConcreteCommand(person, healthyHappy[0, 0], healthyHappy[0, 1],
                commandText);
            break;
        case "Salad":
            cc = new ConcreteCommand(person, healthyHappy[1, 0], healthyHappy[1, 1],
                commandText);
            break;
        case "Strawberries":
            cc = new ConcreteCommand(person, healthyHappy[2, 0], healthyHappy[2, 1],
                commandText);
            break;
        case "A workout":
            cc = new ConcreteCommand(person, healthyHappy[3, 0], healthyHappy[3, 1],
                commandText);
            break;
        case "Nothing all day":
            cc = new ConcreteCommand(person, healthyHappy[4, 0], healthyHappy[4, 1],
                commandText);
            break;
        case "A barrel roll":
            cc = new ConcreteCommand(person, healthyHappy[5, 0], healthyHappy[5, 1],
                commandText);
            break;
        case "Sleep":
            cc = new ConcreteCommand(person, healthyHappy[6, 0], healthyHappy[6, 1],
                commandText);
            break;
        case "The dentist":
            cc = new ConcreteCommand(person, healthyHappy[7, 0], healthyHappy[7, 1],
                commandText);
            break;
        case "An ice cream shop":
            cc = new ConcreteCommand(person, healthyHappy[8, 0], healthyHappy[8, 1],
                commandText);
            break;
        default:
            MessageBox.Show(optionCB.SelectedValue.ToString());
            break;
    }

    if (cc != null)
    {
        cc.execute();
        undoStack.Push(cc);
        actionsLB.Items.Add("I will " +
                actionCB.GetItemText(actionCB.SelectedItem).ToLower() + " " +
                optionCB.GetItemText(optionCB.SelectedItem).ToLower() + ".");
        healthLabel.Text = person.getHealthyText();
        happinessLabel.Text = person.getHappyText();
    }
}
```

Next, I provided code for the undo button and redo button. These two methods are very similar. They both pop the top command off the appropriate stack, execute it, and push it on to the other stack. Then they display the results of these actions in the form.

```
        private void undoButton_Click(object sender, EventArgs e)
        {
            redoButton.Enabled = true;
            cc = undoStack.Pop();
            string nextText = cc.unexecute();
            redoStack.Push(cc);

            actionsLB.Items.Add(nextText);
            healthLabel.Text = person.getHealthyText();
            happinessLabel.Text = person.getHappyText();

            if (undoStack.Count == 0)
                undoButton.Enabled = false;
        }

        private void redoButton_Click(object sender, EventArgs e)
        {
            undoButton.Enabled = true;
            cc = redoStack.Pop();
            string nextText = cc.execute();
            undoStack.Push(cc);

            actionsLB.Items.Add(nextText);
            healthLabel.Text = person.getHealthyText();
            happinessLabel.Text = person.getHappyText();

            if (redoStack.Count == 0)
                redoButton.Enabled = false;
        }
```

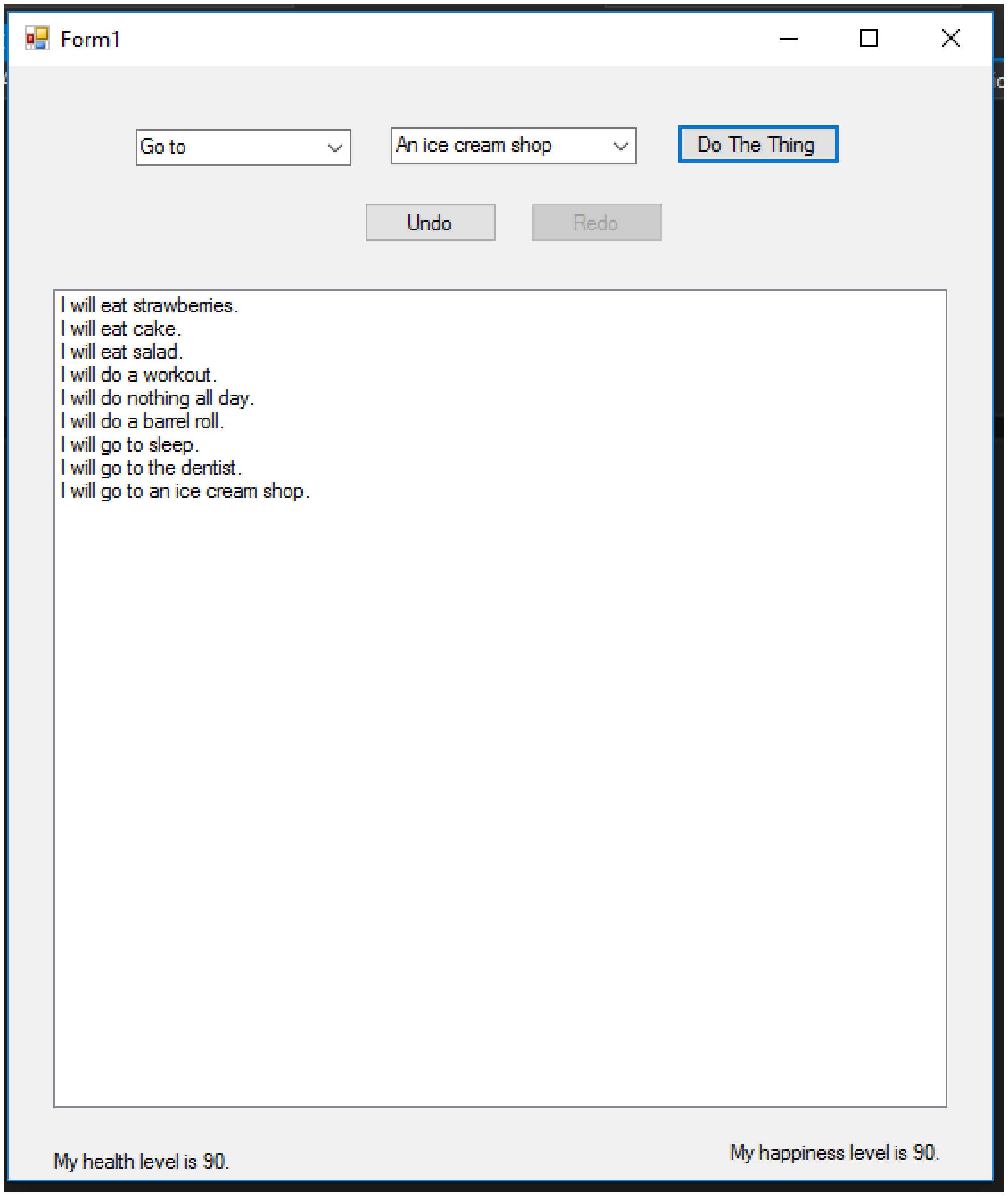
Finally, I provided a short method to enable the Do The Thing button once an action and option have both been selected.

```
private void optionCB_SelectedIndexChanged(object sender, EventArgs e)
{
    doActionButton.Enabled = true;
}
```

With this last addition of code, my program was finished.

The Deliverable

On the next three pages are three screenshots of my program in action.

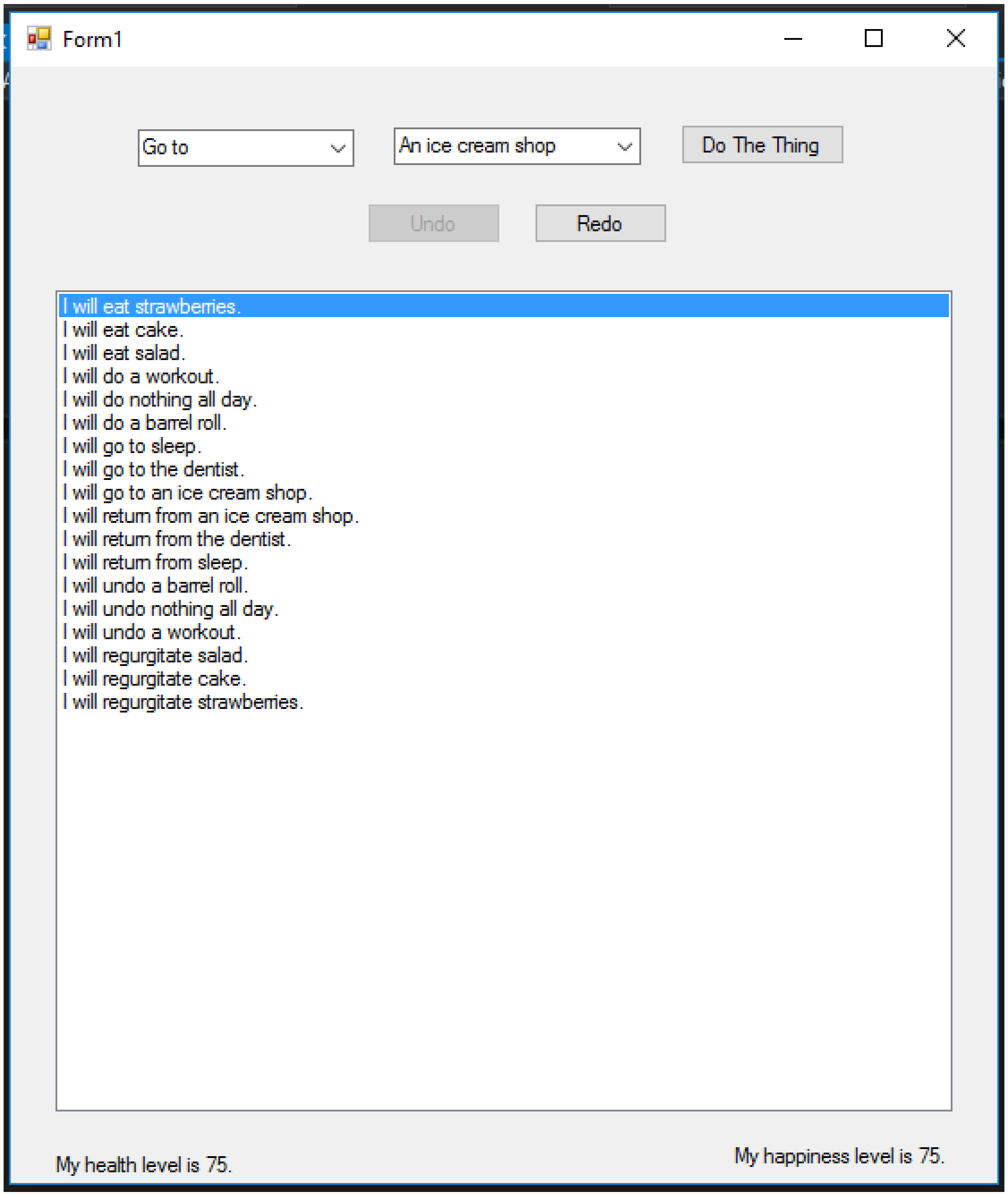


```
I will eat strawberries.
I will eat cake.
I will eat salad.
I will do a workout.
I will do nothing all day.
I will do a barrel roll.
I will go to sleep.
I will go to the dentist.
I will go to an ice cream shop.
```

My health level is 90.

My happiness level is 90.

Here, I performed each available action once. Note that the redo button is not enabled and my health and happiness are both at 90.

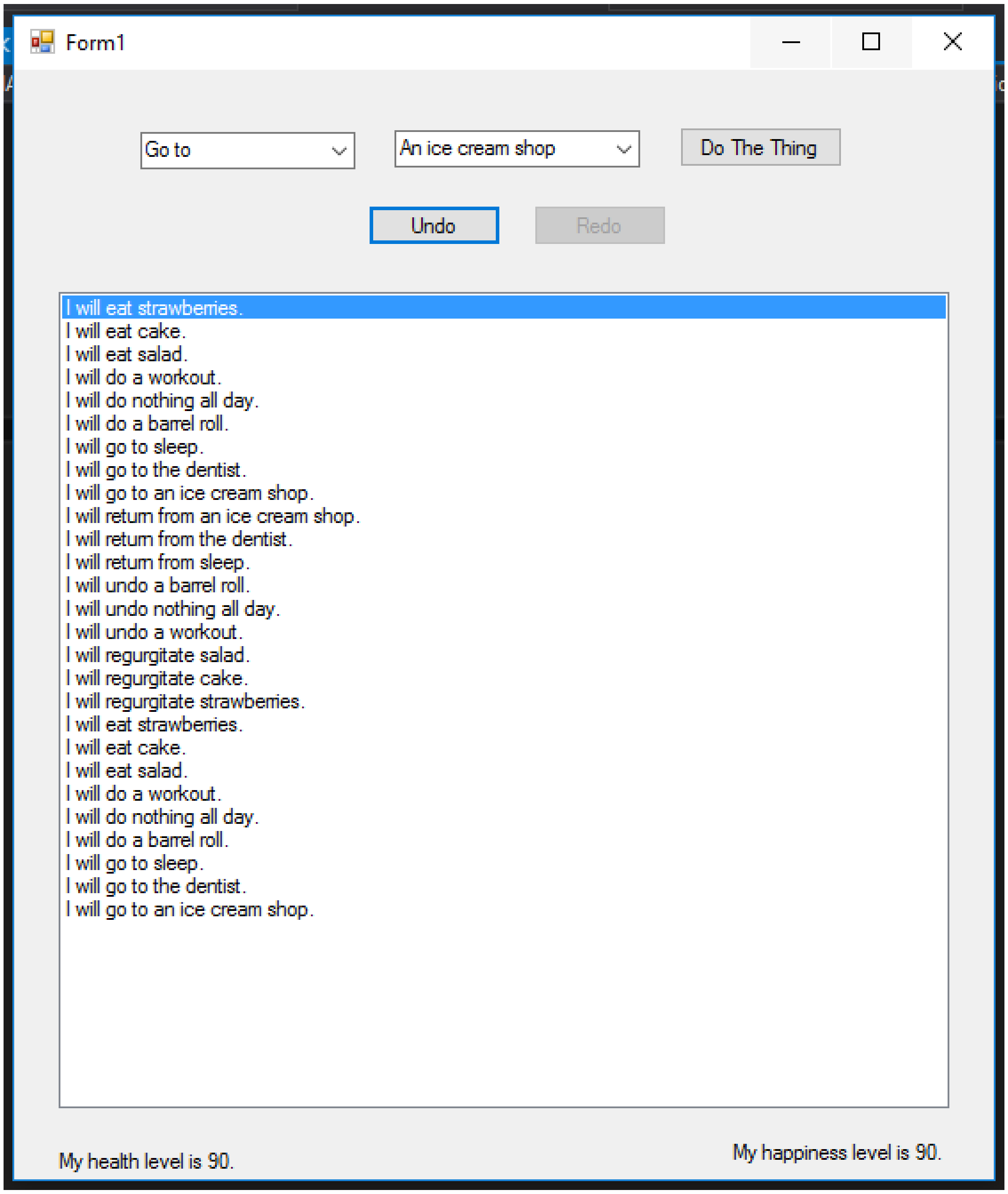


```
I will eat strawberries.
I will eat cake.
I will eat salad.
I will do a workout.
I will do nothing all day.
I will do a barrel roll.
I will go to sleep.
I will go to the dentist.
I will go to an ice cream shop.
I will return from an ice cream shop.
I will return from the dentist.
I will return from sleep.
I will undo a barrel roll.
I will undo nothing all day.
I will undo a workout.
I will regurgitate salad.
I will regurgitate cake.
I will regurgitate strawberries.
```

My health level is 75.

My happiness level is 75.

Here, I undid each of my actions. Now, the redo button is enabled and the undo button is not enabled. Also note that my health and happiness levels have returned to the defaults of 75 and 75.



```
I will eat strawberries.
I will eat cake.
I will eat salad.
I will do a workout.
I will do nothing all day.
I will do a barrel roll.
I will go to sleep.
I will go to the dentist.
I will go to an ice cream shop.
I will return from an ice cream shop.
I will return from the dentist.
I will return from sleep.
I will undo a barrel roll.
I will undo nothing all day.
I will undo a workout.
I will regurgitate salad.
I will regurgitate cake.
I will regurgitate strawberries.
I will eat strawberries.
I will eat cake.
I will eat salad.
I will do a workout.
I will do nothing all day.
I will do a barrel roll.
I will go to sleep.
I will go to the dentist.
I will go to an ice cream shop.
```

My health level is 90.

My happiness level is 90.

Here, I clicked redo nine times. The buttons are appropriately enabled and my health and happiness are both at 90.

<u>Observations</u>

I had a hard time coming up with a good idea for the adapter pattern, and what I finally settled on didn't involve nearly as much as my ideas usually do for this class. However, I think I got a good understanding of this pattern and its purpose, and I could see it being useful if I am ever asked to maintain poorly written legacy code at a future job.

The command pattern will be extremely useful to me in the future because undo and redo buttons, in some form, are necessary in most real software applications. Since no one died in the making of this application, I didn’t have quite as much fun as I usually do, but I think I have a good understanding of how both of these patterns work.