Decorator Pattern

Design Patterns

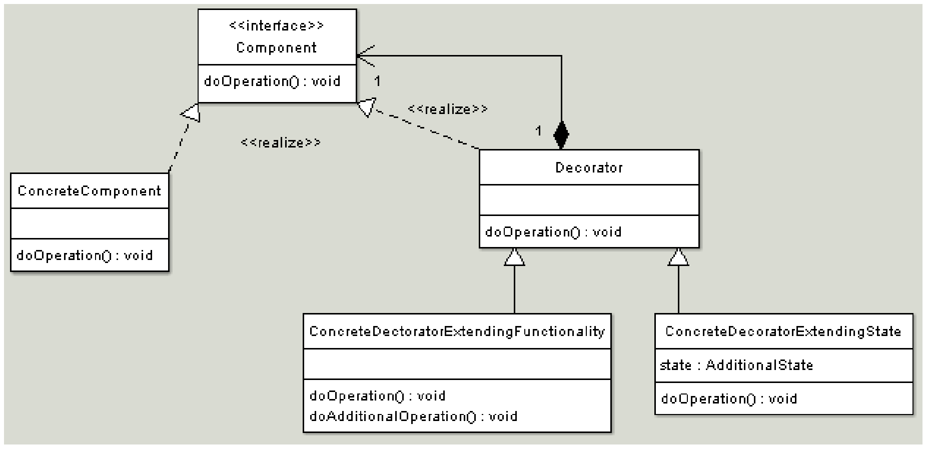
Christopher Doseck

11/17/2016

Introduction

This assignment is an application that I created to show how the decorator pattern works. In this application I am creating superheroes to demonstrate the bridge pattern. I use each superpower as a concrete decorator.

The UML Diagram for Decorator



The UML Diagram for the decorator pattern, shown on the right, shows the classes that are needed to have the requirements. The Hero abstract class represents the component interface. The HeroDecorator is the decorator, and the SimpleHero is the concrete component. The strong hero, fast hero, flying hero, and other hero, are all concrete decorators. The table below shows how all of the classes were used.

Hero	This is the abstract class that represents the Component interface.
SimpleHero	This is a simple hero with no powers. It represents the ConcreteComponent class.
HeroDecorator	This is the class that is used to set a base for decorating other Hero classes.
StrongHero	This is a hero with super strength. It represents ConcreteDecorator class.
FastHero	This is a hero with super speed. It represents ConcreteDecorator class.
FlyingHero	This is a hero with the ability to fly. It represents ConcreteDecorator class.
OtherHero	This is a hero with other super powers. It represents ConcreteDecorator class.
Form1	This is the client that shows the decorator pattern in action

Narrative

```
public abstract class Hero
{
    protected string name;
    protected string level;
    public string getName()
    {
        return name;
    }
    public abstract string getPowers();
}
```

This is the abstract Hero class. It has a name and level, which are both protected so that the children classes can access them. The getName() method is always the same. The getPowers() method is different for each class, so it is marked as abstract.

```
public class SimpleHero : Hero
{
    public SimpleHero(string name)
    {
        base.name = name;
    }

    public override string getPowers()
    {
        return null;
    }
}
```

This is the SimpleHero class. It demonstrates a superhero without any powers. In the constructor it gets passed a name. The getPowers() method returns null because there are no powers.

```
public abstract class HeroDecorator : Hero
{
    public Hero hero;

    public HeroDecorator(Hero hero)
    {
        this.hero = hero;
        this.name = hero.getName();
    }

    public override string getPowers()
    {
        return hero.getPowers();
    }
}
```

This is the HeroDecorator class. It gets passed a Hero object in its constructor, and passes assigns it to a hero object in it. It also stores the name in it.

The getPowers() method returns what hero.getPowers() returns.

```
public class StrongHero : HeroDecorator
{
    public StrongHero(Hero hero, string level) : base(hero)
    {
        this.level = level;
    }

    public override string getPowers()
    {
        return base.getPowers() + ", is strong enough to " + level;
    }
}
```

This is the StrongHero class. In its constructor, it is passed a Hero object and a string object. It stores the string object within itself, and passes the hero to the HeroDecorator class.

The getPowers() method returns what the getPowers() method of HeroDecorator returns, which is the Hero object that it is passed.getPowers() method, and some additional text related to the hero.

The rest of the classes that inherit from HeroDecorator are not included because they are similar and would just be redundant.

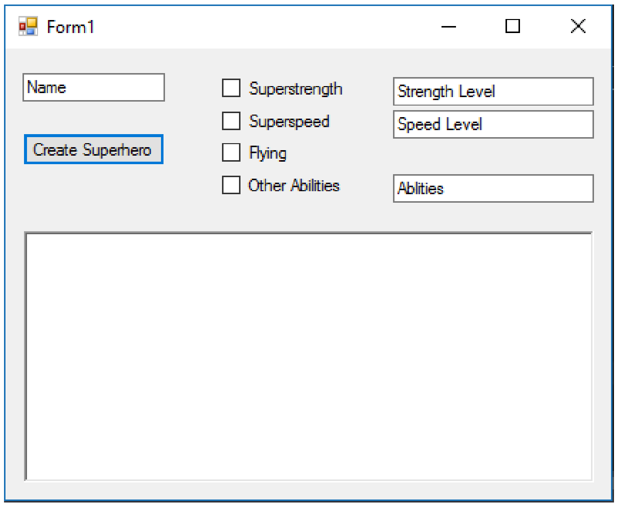
```
public partial class Form1 : Form
{
    private Hero hero;
    public Form1()
    {
        InitializeComponent();
    }

    private void btnCreate_Click(object sender, EventArgs e)
    {
        hero = new SimpleHero(tbName.Text);

        if (cbStrenght.Checked)
            hero = new StrongHero(hero, tbStrength.Text);
        if (cbSpeed.Checked)
            hero = new FastHero(hero, tbSpeed.Text);
        if (cbFlying.Checked)
            hero = new FlyingHero(hero, null);
        if (cbOther.Checked)
            hero = new OtherHero(hero, tbOther.Text);
        tbHeroes.Text += hero.getName() + hero.getPowers() + "\n";

    }
}
```

This is the form that demonstrates the decorator pattern.

When the btnCreate button is clicked, a new simple hero is created with the name from the tbName.

It then checks which textboxes are checked, and creates a new ConcreteDecorator for the boxes that are checked, passing it the former Hero object, and assigning it back to it.

It then prints out the hero name, and the powers that it has.


Form1

Name

☐ Superstrength — Strength Level

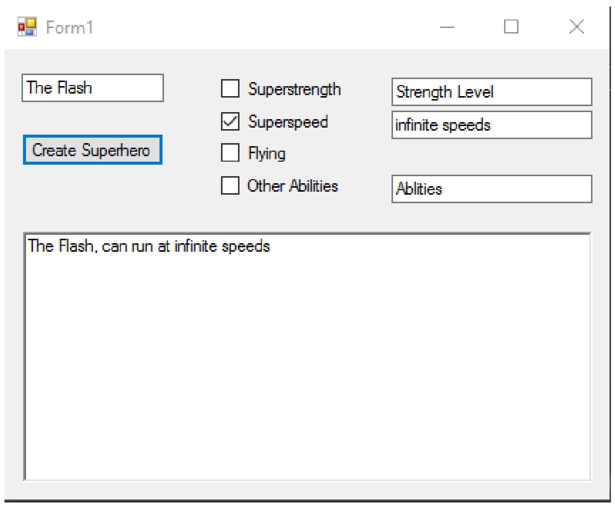
☐ Superspeed — Speed Level

☐ Flying

☐ Other Abilities — Ablities

Create Superhero

This is the basic setup of the GUI.


Form1

The Flash

☐ Superstrength — Strength Level

☑ Superspeed — infinite speeds

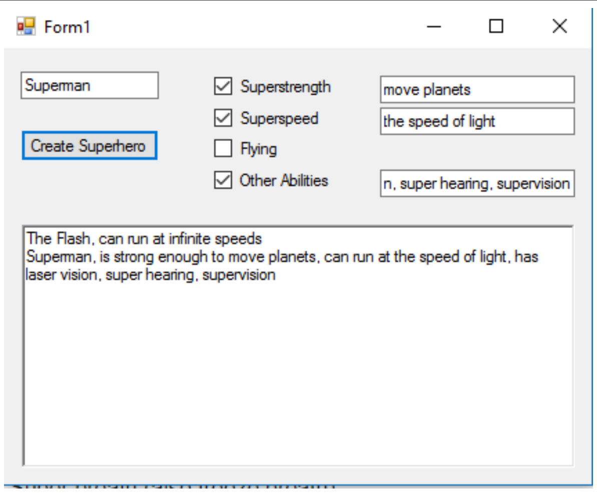
☐ Flying

☐ Other Abilities — Ablities

Create Superhero

```
The Flash, can run at infinite speeds
```

This is after the flash is added.


Form1

Superman

☑ Superstrength — move planets

☑ Superspeed — the speed of light

☐ Flying

☑ Other Abilities — n, super hearing, supervision

Create Superhero

```
The Flash, can run at infinite speeds
Superman, is strong enough to move planets, can run at the speed of light, has
laser vision, super hearing, supervision
```

This is after superman is added.