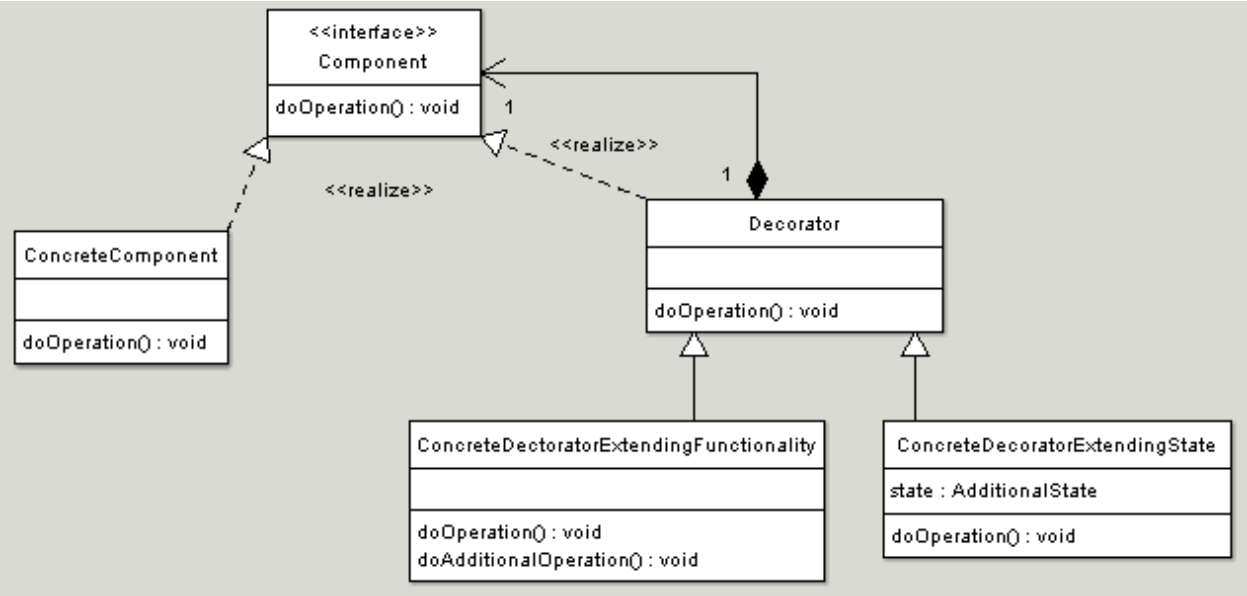
The Decorator Pattern
Design Patterns
Andreya Grzegorzewski – Ohio Northern University
Fall 2016-2017

Introduction

This assignment requires an application that makes use of the decorator pattern. This pattern is used as an alternative to creating multiple subclasses; it can extend an object's functionality dynamically. My app simulates a role-playing game where a group of players (called a "party" in RPGs) fights a monster. The user can add healers, damage dealers, and tanks to the party and then click the "Next Turn" button, when damage is dealt to the monster and to the party. The results of the turn are shown on several labels throughout the form.

The UML Diagram



Above is the UML diagram for the decorator pattern, courtesy of oodesign.com. At the top of the diagram is the Component interface, which describes every class in this pattern. Both the ConcreteComponent class and the Decorator class inherit from the Component interface. ConcreteDecorators are applied to a ConcreteComponent to extend the ConcreteComponent's functionality. On the next page is a table displaying the classes I used to implement this pattern.

Component	My component class is called Player. It refers to a player who is going to fight a monster.
ConcreteComponent	The ConcreteComponent class is called Party. This class is a representation of a collection of players who are working together to fight the monster.
Decorator	My Decorator class is called Role, which refers to the different roles that party members can play.
ConcreteDecorator ExtendingState1	My first ConcreteDecorator is called DamageDealer. This role deals a lot of damage.
ConcreteDecorator ExtendingState2	The second ConcreteDecorator is called Tank. It deals a little damage, but also can protect the party from some damage.
ConcreteDecorator ExtendingState3	My final ConcreteDecorator is called Healer. It does not deal damage, but it can heal the party by a large amount.

Narrative and Code

I started writing my code by creating the *Component* interface, which provides prototypes for four functions. The functions are shown and described below.

```
public abstract class Player // This is the Component class
{
    /*
     * dealDamage returns the amount of damage dealt to the monster;
     * diff refers to the difference between the default damage dealt
     * and the actual damage dealt for the specific party member.
     */
    public abstract int dealDamage(int diff);

    /*
     * takeDamage returns the remaining health of the entire party;
     * damage is the amount of damage tha the monster deals
     * and diff is the amount of damage that is healed or protected from.
     */
    public abstract int takeDamage(int damage, int diff);

    // Returns the total health of the party.
    public abstract int getHealth();

    /*
     * Adds a new player to the party.
     * healthDiff is the difference between the default health value
     * and the actual health for the specific party member.
     */
    public abstract void addPlayer(int healthDiff);
}
```

Next, I defined the Party class, which serves as my ConcreteComponent. This class defines two variables and then overrides the ToString method, as well as all of the methods prototyped in the Player class.

```
class Party : Player // This is the ConcreteComponent class
{
    int health = 0;
    int damageToDeal = 0;

    public override int dealDamage(int diff)
    {
        return damageToDeal + diff;
    }

    public override int takeDamage(int damage, int diff)
    {
        health += diff - damage;
        return health;
    }

    public override string ToString()
    {
        return "";
    }

    public override int getHealth()
    {
        return health;
    }

    public override void addPlayer(int healthDiff)
    {
        health += 300 + healthDiff;
        damageToDeal += 100;
    }
}
```

I then created my Decorator class, which is called Role in my app. For this class, I created a Player variable and a constructor, then overrode two methods from the Player class. These are the methods whose implementations need to refer directly to a Party object.

```
public abstract class Role : Player // This is the Decorator class
{
    Player player;

    public Role(Player player)
    {
        this.player = player;
    }

    public override int getHealth()
    {
        return player.getHealth();
    }

    public override void addPlayer(int healthDiff)
    {
        player.addPlayer(healthDiff);
```

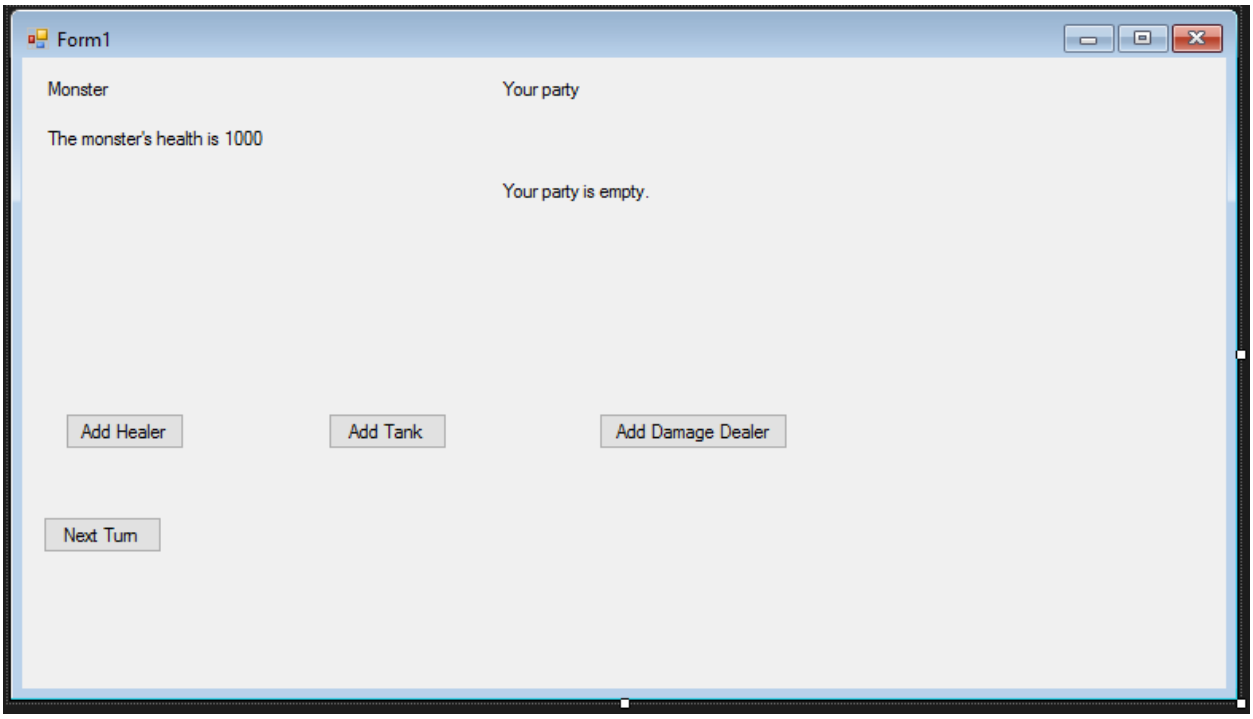
```
    }
}
```

I continued by creating my three Concrete Decorators: Tank, Healer, and DamageDealer. These classes are all implemented very similarly, so only one will be included here. In this class, I define several constants, create a Player variable, and override the methods defined in the Player class. I also define a getter for one of the constants.

```
class DamageDealer : Role // This is a Concrete Decorator
{
    const int DAMAGE_DIFF = 250;
    const int HEALTH_DIFF = 0;
    Player player;

    public DamageDealer(Player player) : base(player)
    {
        this.player = player;
    }

    public override int dealDamage(int diff)
    {
        return player.dealDamage(diff + DAMAGE_DIFF);
    }

    public override int takeDamage(int damage, int diff)
    {
        return player.takeDamage(damage, diff);
    }

    public override string ToString()
    {
        if (player.ToString().Contains("tank") ||
          player.ToString().Contains("healer") || player.ToString().Contains("damage
          dealer"))
            return player.ToString() + " and a damage dealer";
        else
            return player.ToString() + "a damage dealer";
    }

    public override int getHealth()
    {
        return base.getHealth();
    }

    public int getHealthDiff()
    {
        return HEALTH_DIFF;
    }
}
```

The other two ConcreteDecorator classes are available on my GitHub page in the Design-Patterns repository at https://github.com/andreya-grzegorzewski/Design-

Patterns. This concludes the code necessary for my decorator pattern. When I finished this, I moved on to the form. First, I designed the form; a screenshot of my form from the design view of Visual Studio is shown below.



The user can add healers, tanks, and damage dealers to the party with the three buttons near the middle of the screen. Beneath the “Your party” label, another label is updated with the cumulative health of the party members. The label under that is also updated to contain a list of the party members. When the “Next Turn” button is pressed, the monster’s health and the party’s health are updated appropriately.

Once I designed my form, I started writing code for it. I began by creating some variables and filling in the Form1 constructor:

```
Player player;

int monsterDamage = 150; //Default value of damage dealt by the monster each turn
int monsterHealth = 1000;

public Form1()
{
    player = new Party();
    InitializeComponent();

    partyDescLabel.MaximumSize = new Size(400, 0);
    partyDescLabel.AutoSize = true;
}
```

Next, I handled the events fired upon button clicks for the Add Healer, Add Tank, and Add Damage Dealer buttons. These methods are all very similar, so only one is included here. Again, the other two can be found in my Design-Patterns respository on GitHub.

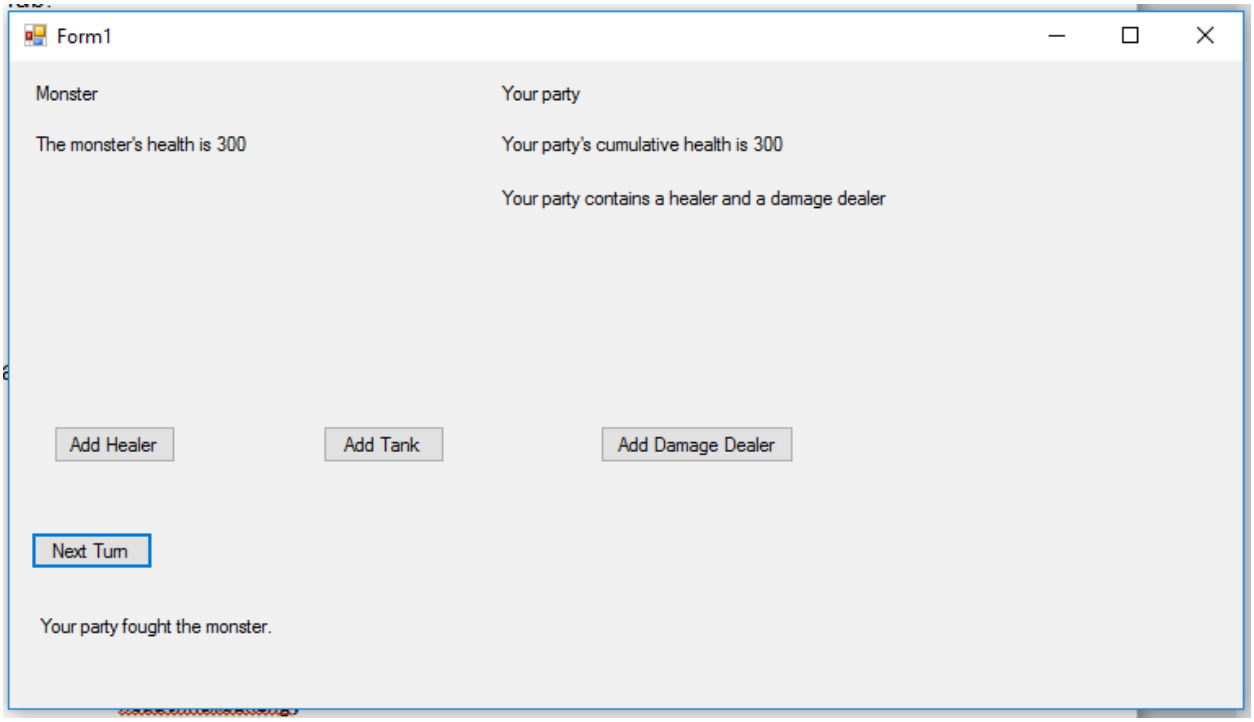
```
private void healerButton_Click(object sender, EventArgs e)
{
    Healer h = new Healer(player);
    player = h;
    player.addPlayer(h.getHealthDiff());

    resultLabel.Text = "You have just added a healer to your party.";
    partyDescLabel.Text = "Your party contains " + player.ToString();
    partyHealthLabel.Text = "Your party's cumulative health is " +
      player.getHealth();
}
```

Finally, I implemented the event handler for the Next Turn button's click event.

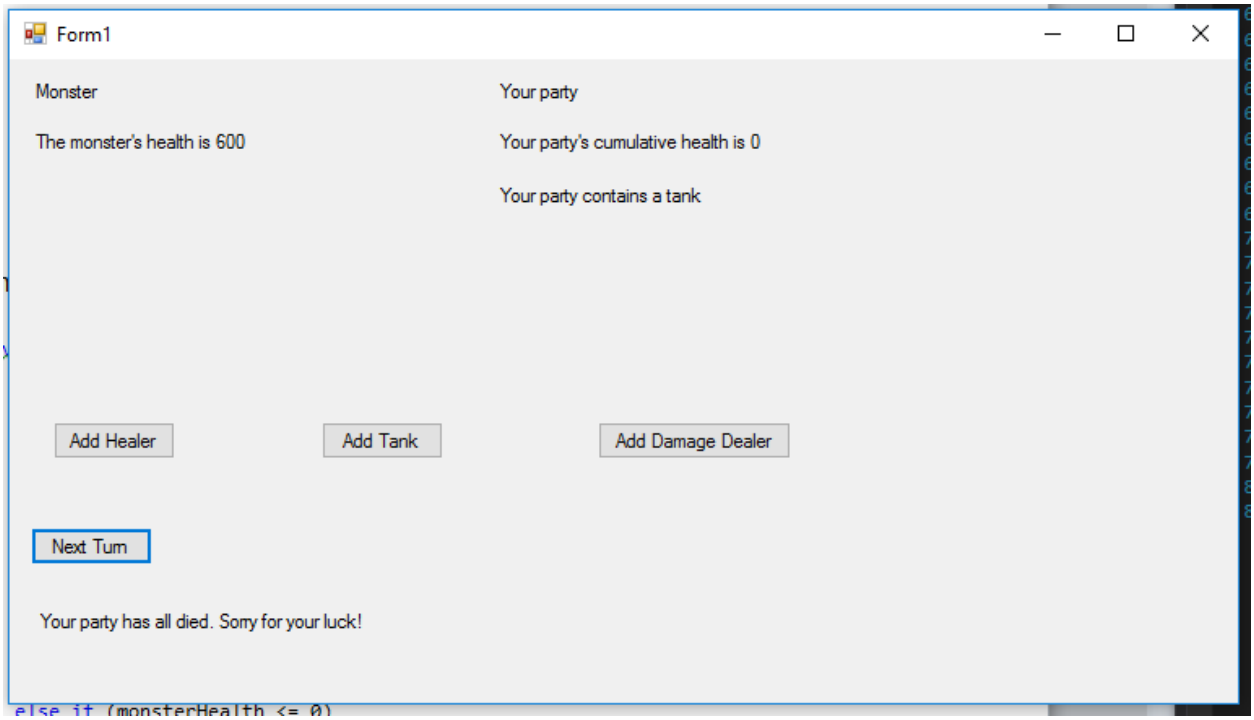
```
private void nextTurnButton_Click(object sender, EventArgs e)
{
    // Get damage dealt/health remaining values
    int healthRemaining = player.takeDamage(monsterDamage, 0);
    int damageDealt = player.dealDamage(0);
    monsterHealth -= damageDealt;

    // Update display
    resultLabel.Text = "Your party fought the monster.";
    monsterHealthLabel.Text = "The monster's health is " + monsterHealth;
    partyHealthLabel.Text = "Your party's cumulative health is " +
      healthRemaining;

    if (healthRemaining <= 0)
        resultLabel.Text = "Your party has all died. Sorry for your luck!";
    else if (monsterHealth <= 0)
        resultLabel.Text = "You have killed the monster! Congratulations!";
}
```

This concludes the code that I wrote for this app. Following on the next two pages is a series of screenshots of my app in action.

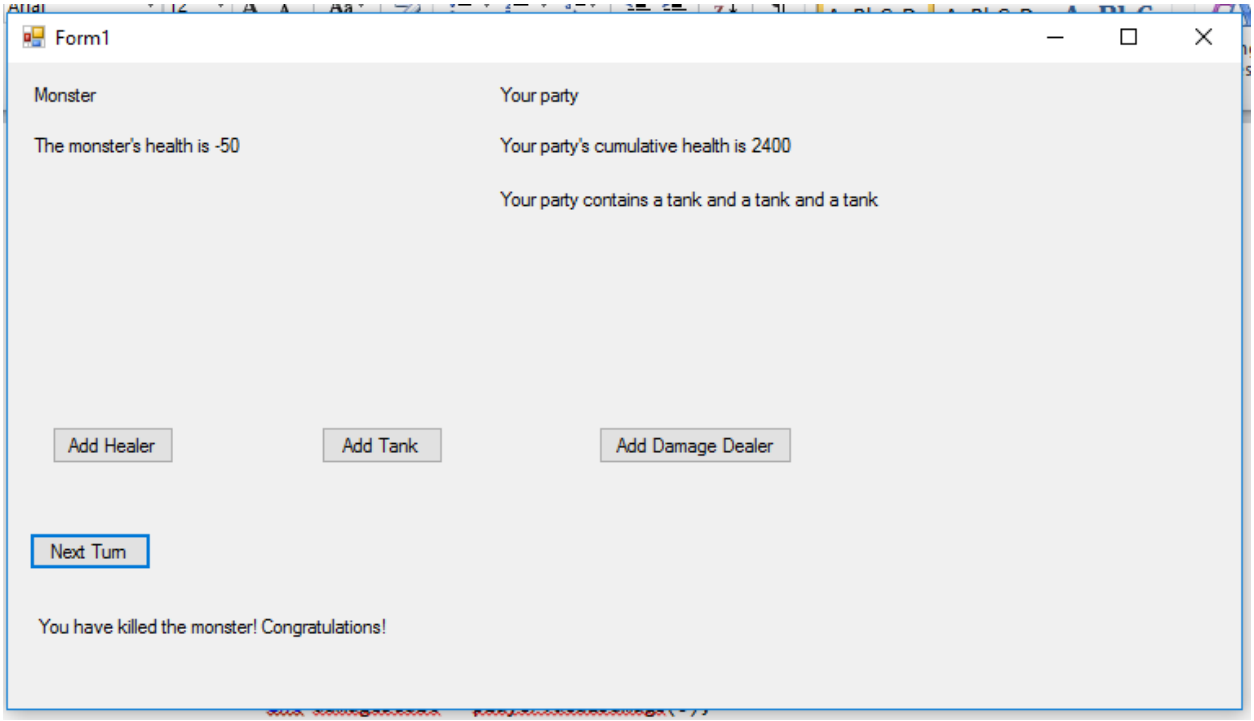
This is the app after adding a healer and a damage dealer to the party, before any turns have been taken. The labels display the appropriate health values, the members of the party, and the most recent action taken.



This is the app after one turn has been taken; the health labels and action label have been updated.



This is the app after everyone in the party has died!



Finally, this is the app after the party has defeated the monster.

Observations

This pattern was particularly difficult for me because I had a hard time grasping what it could be used for and how to implement it. It took me a long time to come up with a concept that I could apply this pattern to. I had a lot of problems trying to get the numbers right; I tried several times before getting dealDamage, takeDamage, and getHealth to work properly. I think that my app appropriately demonstrates the decorator pattern, but I would like to hear more about it in class so I can understand it better. I would also like to better understand why someone would choose to use this pattern in actual software development.