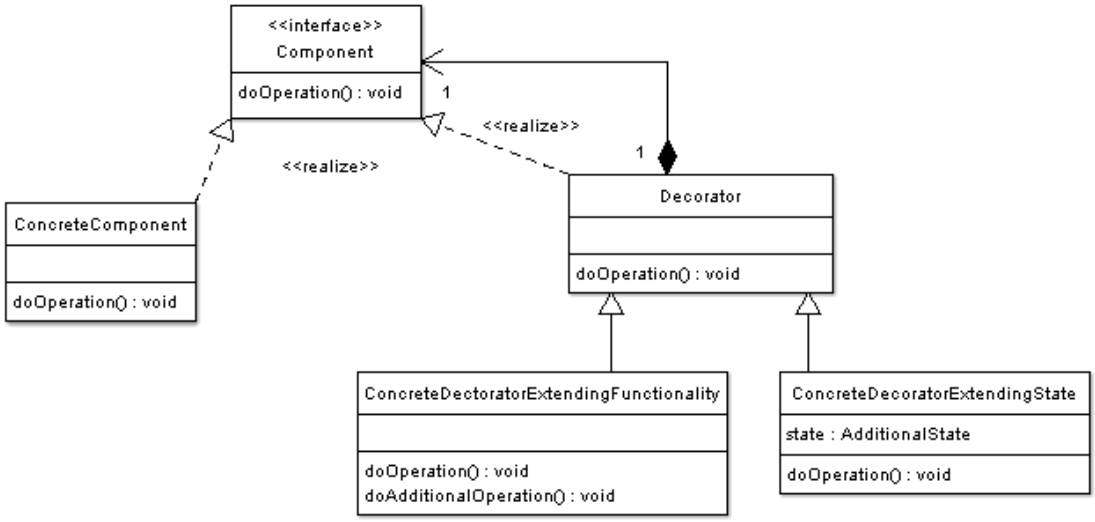
Jack Raney

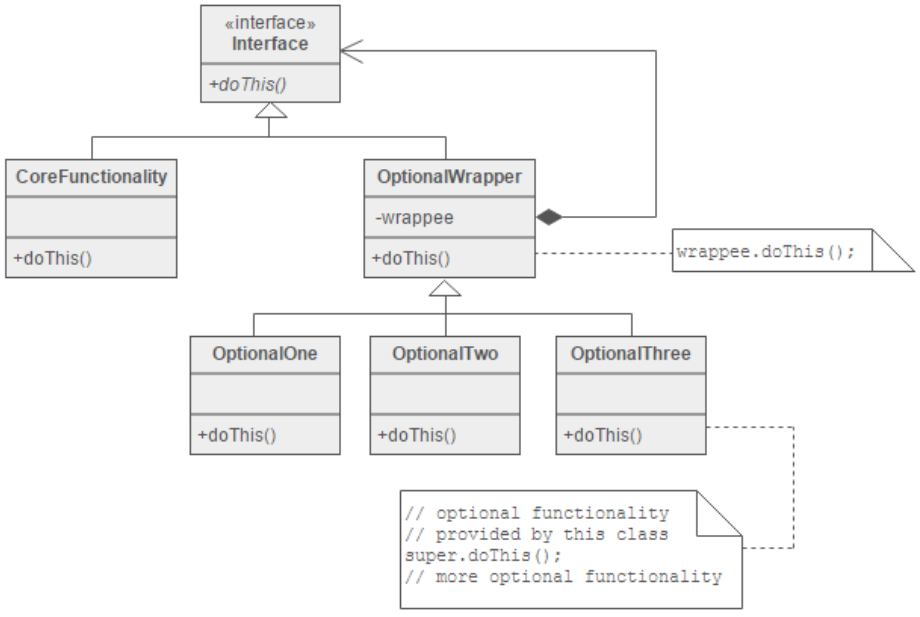
Design Patterns

11/17/2016

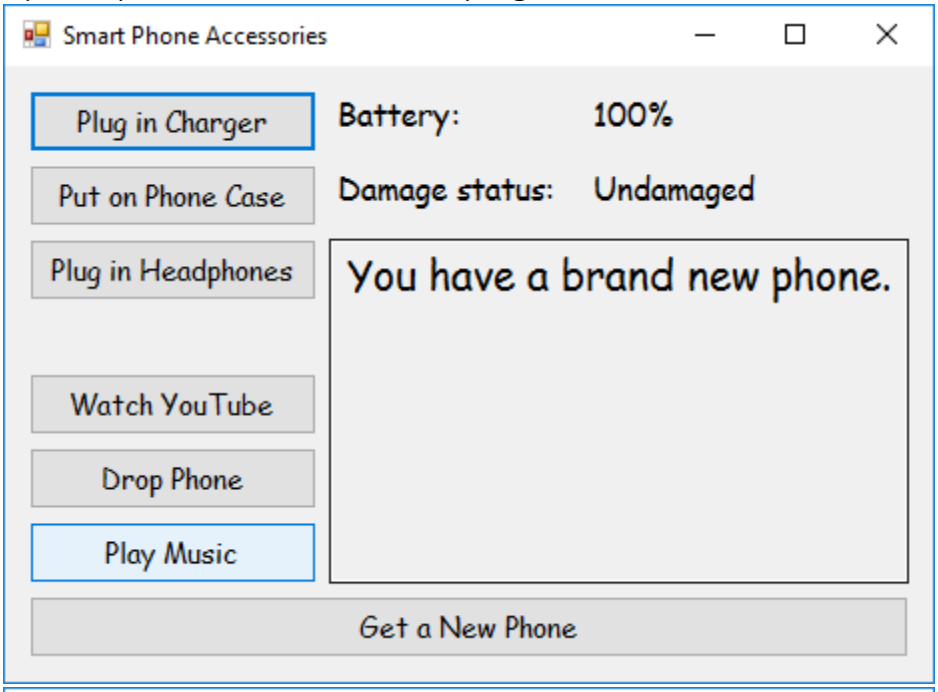
The Decorator Pattern

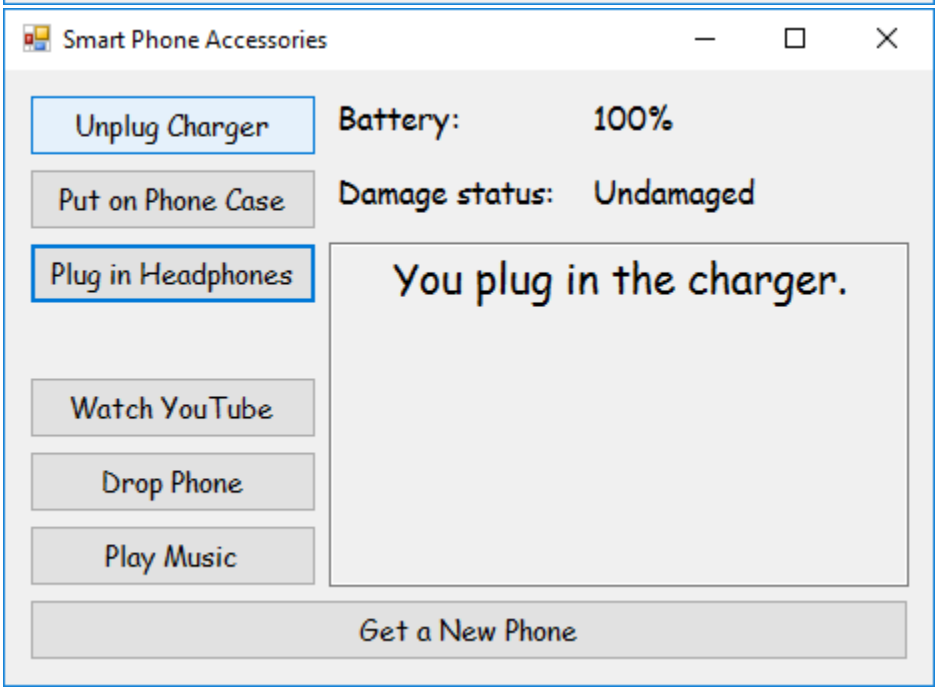
The purpose of this paper is to explain the Decorator Pattern, a design pattern that's purpose is, according to oodesign, "to add additional responsibilities dynamically to an object". This means that the pattern gives on object new properties and/or methods that it can use. Two UML diagrams for the Decorator pattern are shown below, courtesy of oodesign.com and sourcemaking.com.

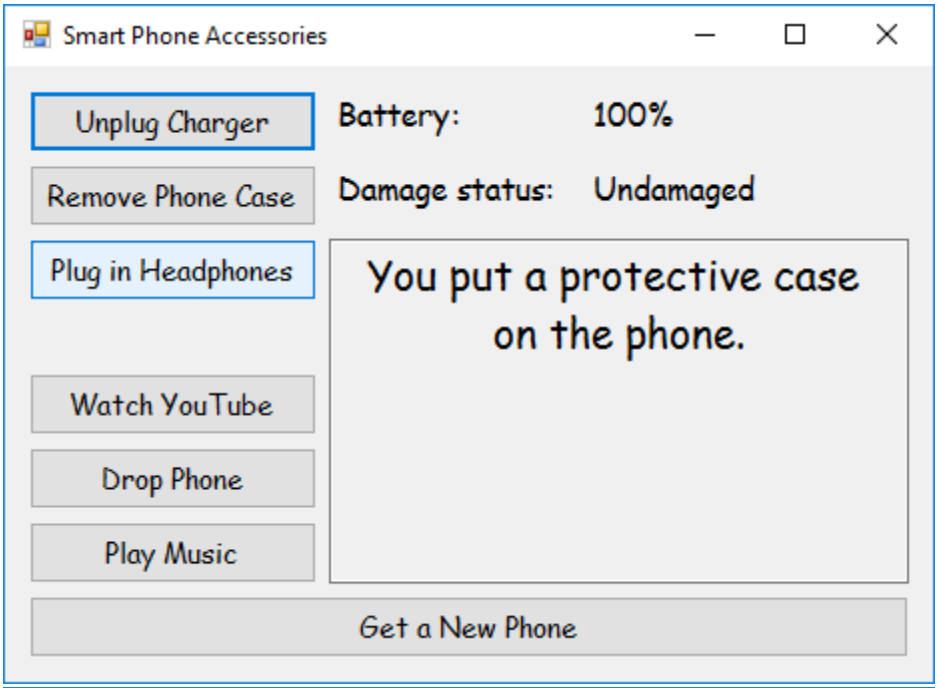




To exemplify this pattern, I created the program SmartPhoneAccessories.exe:






Smart Phone Accessories
Unplug Charger
Remove Phone Case
Plug in Headphones
Watch YouTube
Drop Phone
Play Music
Battery:
100%
Damage status:
Undamaged
You put a protective case on the phone.
Get a New Phone


Smart Phone Accessories
Unplug Charger
Remove Phone Case
Unplug Headphones
Watch YouTube
Drop Phone
Play Music
Battery:
100%
Damage status:
Undamaged
You plug in headphones.
Get a New Phone

Code for this Program

Phone.cs

```
public abstract class Phone //Component class
    {
        protected int charge;
        protected int durability;
        protected bool isLoud;

        public abstract int drain();
        public abstract int drop();

        public void drainCharge()
        {
            charge -= drain();
            if (charge < 0)
                charge = 0;
            if (charge > 100)
                charge = 100;
        }

        public void hurtPhone()
        {
            durability -= drop();
            if (durability < 0)
                durability = 0;
        }

        public void setLoud(bool l)
        {
            isLoud = l;
        }

        public int getCharge()
        {
            return charge;
        }

        public int getDurability()
        {
            return durability;
        }
        public bool getLoud()
        {
            return isLoud;
        }

    }
```

The Phone class represents the entire system of the phone and its accessories. It holds properties for a phone's battery charge, it's durability (which decreases as the phone is damaged) and whether or not the phone will be loud if noises are played.

Methods drain() and drop() return values change battery charge and durability respectively; these methods will differ between the different child classes of Phone. They are called in drainCharge() and hurtPhone() to change those values.

The rest of the methods are for setting the isLoud Boolean and getting each of the three

BasePhone.cs

```
public class BasePhone : Phone //ConcreteComponent class
    {
        protected Random rand = new Random();

        public BasePhone()
        {
            charge = 100;
            durability = 100;
            isLoud = true;
        }

        public BasePhone(int c, int d)
        {
            charge = c;
            durability = d;
            isLoud = true;
        }
```

BasePhone is the phone itself. It holds a Random property that it uses for its drain() and drop() overrides.

It has two different constructors: one for a brand new phone, and another it uses to reconstruct the whole system when an accessory is removed.

```
        public override int drain()
        {
            return rand.Next() % 20 + 1;
        }

        public override int drop()
        {
            return rand.Next() % 50 + 1;
        }
    }
```

drain() in BasePhone reduces the battery life of the phone.

drop() in BasePhone reduces the durability of the phone.

PhoneAccessory.cs

```
public abstract class PhoneAccessory : Phone //Decorator class
    {
        protected Phone phone;
    }
```

PhoneAccessory represents the different accessories for a phone. The only common property that the accessories share is that they are attached to a phone.

Charger.cs

```
public class Charger : PhoneAccessory   //ConcreteDecorator
    {
        public Charger(Phone p)
        {
            phone = p;
            charge = p.getCharge();
            durability = p.getDurability();
        }

        public override int drain()
        {
            return -(phone.drain());
        }

        public override int drop()
        {
            return phone.drop();
        }
    }
```

All three accessories inherit the phone and its charge and durability.

Each accessory changes one of the phone's properties or methods.

The Charger object makes drain return a negative value, allowing the phone to gain power…

Case.cs

```
public class Case : PhoneAccessory  //ConcreteDecorator
    {
        public Case(Phone p)
        {
            phone = p;
            charge = p.getCharge();
            durability = p.getDurability();
        }

        public override int drain()
        {
            return phone.drain();
        }

        public override int drop()
        {
            int damage = phone.drop();
            if (damage <= 40)
                return 0;
            return damage - 40;
        }
    }
```

…the Case object greatly reduces damage from dropping the phone…

Headphones.cs

```
public class Headphones : PhoneAccessory    //ConcreteDecorator
    {
        public Headphones(Phone p)
        {
            phone = p;
            charge = p.getCharge();
            durability = p.getDurability();
            phone.setLoud(false);
        }

        public override int drain()
        {
            return phone.drain();
        }

        public override int drop()
        {
            return phone.drop();
        }
    }
```

…and the Headphones object sets the isLoud variable of the phone to false.

Form1.cs

```
public partial class Form1 : Form
    {
        Phone phone;

        bool hasCharger;
        bool hasCase;
        bool hasHeadphones;

        public Form1()
        {
            InitializeComponent();
            removeChargerButton.Hide();
            removeCaseButton.Hide();
            removeHeadphonesButton.Hide();

            phone = new BasePhone();

            hasCharger = false;
            hasCase = false;
            hasHeadphones = false;

            messageBox.Text = "You have a brand new phone.";
        }

        private void addChargerButton_Click(object sender, EventArgs e)
        {
            hasCharger = true;

            phone = new Charger(phone);

            messageBox.Text = "You plug in the charger.";

            addChargerButton.Hide();
            removeChargerButton.Show();
        }
```

Within the form, there is a Phone object, and Booleans for whether or not each accessory is attached to the phone.

When the program starts, the buttons for removing accessories are hidden (because no accessory has been added), phone is set as a BasePhone, and the Booleans for having each accessory are set to false.
(Messages are given to the user through a text box every time an action is taken.)

Each of the three adding buttons set their respective Booleans to true, updates the phone system to add the respective accessory, and places the previous version of the system inside the new one. The button is then "swapped" with its respective removing button.

```
        private void removeAcc()     //Remove accessory
        {
            phone = new BasePhone(phone.getCharge(), phone.getDurability());
            if (hasCharger)
                phone = new Charger(phone);
            if (hasCase)
                phone = new Case(phone);
            if (hasHeadphones)
                phone = new Headphones(phone);
        }

        private void removeChargerButton_Click(object sender, EventArgs e)
        {
            hasCharger = false;

            removeAcc();

            messageBox.Text = "You unplug the charger.";

            removeChargerButton.Hide();
            addChargerButton.Show();
        }
```

The removeAcc() function resets the system and adds all the accessories that weren't removed to give the illusion of simply removing the accessory.

The remove buttons follow the same structure as the adding buttons, only they set their respective Booleans to false and call the removeAcc() function.

```
    private void watchButton_Click(object sender, EventArgs e)
        {
            if (phone.getCharge() == 0 && !hasCharger)
            {
                messageBox.Text = "Your phone is dead.";
                return;
            }

            phone.drainCharge();

            powerLabel.Text = phone.getCharge() + "%";

            if (hasCharger)
            {
                messageBox.Text = "You watch a funny video.";
            }
            else
            {
                if (phone.getCharge() == 0)
                    messageBox.Text = "Your phone dies in the middle of the video.";
                else messageBox.Text = "You watch a funny video, which drains your phone's
power.";
            }
        }
```

The button for watching a video drains or charges the phone depending on whether or not the charger is plugged in. If battery life reaches 0%, no more videos can be watched and no more music can be listened to.

```
        private void showDamage()
        {
            int damage = phone.getDurability();
            if (damage == 100)
                damageLabel.Text = "Undamaged";
            if (damage < 100)
                damageLabel.Text = "Slightly Damaged";
            if (damage < 67)
                damageLabel.Text = "Moderately Damaged";
            if (damage < 34)
                damageLabel.Text = "Severely Damaged";
            if (damage == 0)
                damageLabel.Text = "Broken";
        }

        private void dropButton_Click(object sender, EventArgs e)
        {
            if (phone.getDurability() == 0)
            {
                messageBox.Text = "You drop your broken phone because why not?";
                return;
            }

            phone.hurtPhone();

            showDamage();

            if (phone.getDurability() == 0)
            {
                 messageBox.Text = "You dropped your phone and broke it. Well, time for a new
one!";
                 watchButton.Enabled = false;
                 musicButton.Enabled = false;
            }
            else messageBox.Text = "You dropped your phone. Thankfully, it's not broken.";
        }

        private void musicButton_Click(object sender, EventArgs e)
        {
            if (phone.getCharge() == 0)
            {
                messageBox.Text = "Your phone is dead.";
                return;
            }

            if (phone.getLoud())
                 messageBox.Text = "You play some music through the phone's
speaker. Anyone nearby is annoyed at you.";
            else messageBox.Text = "You play some music through your
headphones.";
        }
```

The showDamage() function is only for displaying the current status of the phone's durability.

The drop button hurts the phone. The damage done is dependent on whether or not a case is on the phone. If the phone breaks, a new one must be retrieved.

The button for playing music causes the phone to make noise through either the phone's speaker or headphones; this is, of course, dependent on whether or not headphones are present.

```
    private void newPhoneButton_Click(object sender, EventArgs e)
    {
        phone = new BasePhone();

        messageBox.Text = "You buy a new phone. You feel $200 poorer (because you are).";

        watchButton.Enabled = true;
        musicButton.Enabled = true;

        powerLabel.Text = phone.getCharge() + "%";
        showDamage();

        removeChargerButton.Hide();
        addChargerButton.Show();
        removeCaseButton.Hide();
        addCaseButton.Show();
        removeHeadphonesButton.Hide();
        addHeadphonesButton.Show();
    }
}
```

The new phone button resets the phone object completely, as an entirely new phone object was created (this functionally resets the program).

Reflection

To make up for the fact that several of the previous programs made for this class were relatively simple to make, I had several issues come up while making this one; I believe this program took longer to make than the previous two combined. I still, of course, managed to successfully create it, which is what truly matters here.

I do not particularly like this pattern, as something in the back of my mind keeps telling me that there's a better way to structure code than this. Either way, I at least know the design pattern.