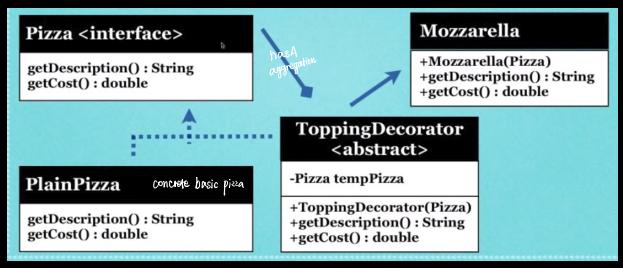



```java
public interface Pizza {

    public String getDescription();

    public double getCost();

}
```

```java
abstract class ToppingDecorator implements Pizza {

    protected Pizza tempPizza;

    public ToppingDecorator(Pizza newPizza){

        tempPizza = newPizza;

    }

    public String getDescription(){

        return tempPizza.getDescription();

    }

    public double getCost(){

        return tempPizza.getCost();

    }
```

```java
public class PlainPizza implements Pizza{

    @Override
    public String getDescription() {
        // TODO Auto-generated method stub
        return "Thin Dough";
    }

    @Override
    public double getCost() {
        // TODO Auto-generated method stub
        return 4.00;
    }


}
```

```java
public class Mozzarella extends ToppingDecorator{

    public Mozzarella(Pizza newPizza) {
        super(newPizza);

        System.out.println("Adding Dough");

        System.out.println("Adding Moz");
    }

    public String getDescription(){

        return tempPizza.getDescription() + ", Mozzarella";

    }

    public double getCost(){

        return tempPizza.getCost() + .50;

    }
```

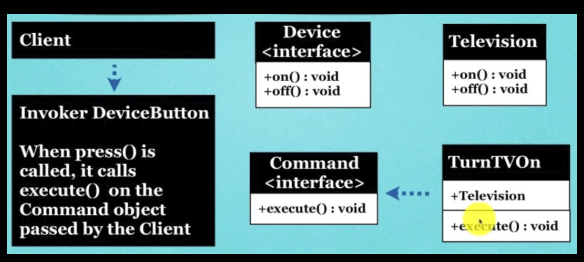
```java
public class TomatoSauce extends ToppingDecorator{

    public TomatoSauce(Pizza newPizza) {
        super(newPizza);

        System.out.println("Adding Sauce");

    }

    public String getDescription(){

        return tempPizza.getDescription() + ", Tomato Sauce";

    }

    public double getCost(){

        return tempPizza.getCost() + .35;

    }
```

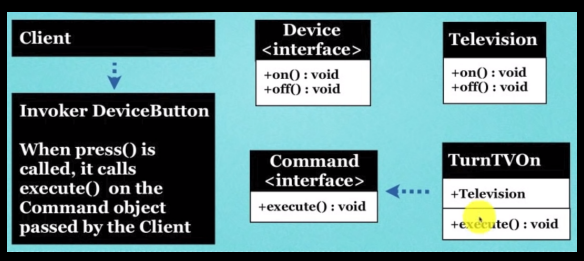
```java
lic class PizzaMaker{

public static void main(String[] args){

    Pizza basicPizza = new TomatoSauce(new Mozzarella(new PlainPizza()));

    System.out.println("Ingredients: " + basicPizza.getDescription());

    System.out.println("Price: " + basicPizza.getCost());

}
```

What is the Decorator Design Pattern?

- The Decorator allows you to modify an object dynamically
- You would use it when you want the capabilities of inheritance with subclasses, but you need to add functionality at run time
- It is more flexible than inheritance
- Simplifies code because you add functionality using many simple classes
- Rather than rewrite old code you can extend with new code

extend the functionality during runtime





What is the Command Design Pattern?

- The command pattern is a behavioural design pattern in which an object is used to represent and encapsulate all the information needed to call a method at a later time.
- This information includes the method name, the object that owns the method and values for the method parameters.

What is the Command Design Pattern?

- Allows you to store lists of code that is executed at a later time or many times.
- Client says I want a specific Command to run when execute() is called on 1 of these encapsulated (hidden) Objects
- An Object called the Invoker transfers this Command to another Object called a Receiver to execute the right code
- TurnTVOn - DeviceButton - TurnTVOn - Television.TurnTVOn()

↑ receiver

Benefits of the Command Design Pattern

- Allows you to set aside a list of commands for later use
- A class is a great place to store procedures you want to be executed
- You can store multiple commands in a class to use over and over
- You can implement undo procedures for past commands
- Negative: You create many small classes that store lists of commands

```java
public interface ElectronicDevice {

    public void on();

    public void off();

    public void volumeUp();

    public void volumeDown();

}
```

```java
public interface Command {

    public void execute();
                undo();
}
```

```java
public class Television implements ElectronicDevice {

    private int volume = 0;

    @Override
    public void on() {

        System.out.println("TV is ON");

    }

    @Override
    public void off() {
        System.out.println("TV is OFF");

    }

    @Override
    public void volumeUp() {

        volume++;
        System.out.println("TV Volume is at " + volume);
```

```java
public class TurnTVOn implements Command{

    ElectronicDevice theDevice;

    public TurnTVOn(ElectronicDevice newDevice){

        theDevice = newDevice;

    }

    @Override
    public void execute() {

        theDevice.on();

    }
    public void undo() {
        theDevice.off();
    }
}
```

```java
public class DeviceButton{

    Command theCommand;

    public DeviceButton(Command newCommand){

        theCommand = newCommand;

    }

    public void press(){

        theCommand.execute();

    }


}
```

```java
public class TVRemote{

    public static ElectronicDevice getDevice(){

        return new Television();

    }

}
```

```java
import java.util.ArrayList;
import java.util.List;

public class PlayWithRemote{

    public static void main(String[] args){

        ElectronicDevice newDevice = TVRemote.getDevice();

        TurnTVOn onCommand = new TurnTVOn(newDevice);

        DeviceButton onPressed = new DeviceButton(onCommand);

        onPressed.press();

    }

}
```