

Design Patterns Notes - An Overview Of Design Patterns

SEPTEMBER 03, 2015

🕒 Reading time ~50 minutes

Introduction

This blog post is a collection of notes on some more common design patterns. Each design pattern is explained in simple terms and includes an example.

This is written mainly for beginners, but it's also useful if you need to refreshen your understanding of a certain design pattern. I highly suggest you to further explore each pattern covered here, since this post's goal is to simply give you an overview of them. I've included some additional sources at the end of the post.

I've written this mainly because it's not always easy to understand what a pattern is about in its essence, since many of the explanations out there are either domain-specific or include examples that are somewhat complex. In my opinion, it's a lot easier to to understand them once you have a general idea of what the pattern does. This is why I've used somewhat "childish" examples here: those should help you concentrate on the pattern and not the domain details.

All of the code examples are in Java, but only the basic syntax is used, so if you're coming from a language like C# or have basic Java knowledge, you should have not problems in understanding this post. Even though a specific programming language is used here, the concepts are domain-independent and can be applied to other languages. The default access modifiers for member variables are used, to make the code interpretation more accessible to beginners.

Design Patterns

Singleton

The **Singleton** Pattern ensures that there is only one instance of the class and provides a global point to access that instance.

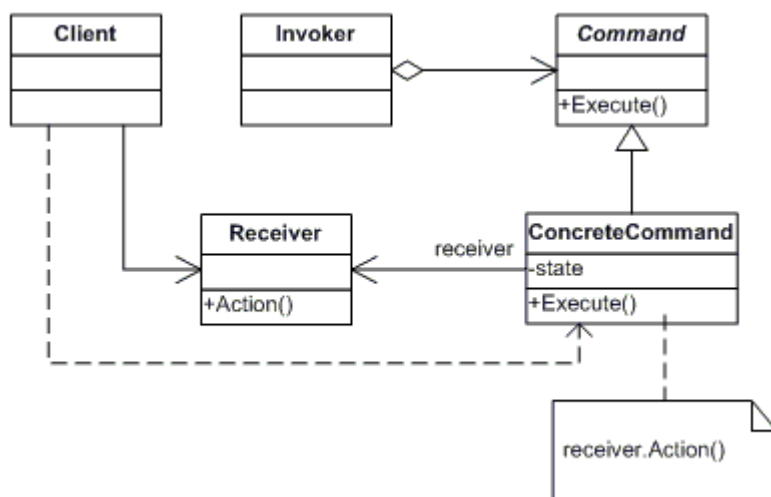
Java's implementation of **Singleton** makes use of a **private constructor** (to make sure that no one in the application can call the `new Singleton()`, thus possibly creating more than one instance of the class) and a **static method**, combined with a **static variable** (to make sure that there is only one instance of the class in the application, instead of calling "new", we ask the class itself to provide an instance and since we never directly instantiate it, the method has to be static). The class itself will be responsible for keeping track of its sole instance.

Below is an example implementation in Java. This version uses **lazy initialization** (the instance of the class isn't created until it's first requested via the `getInstance()` method).

```
public class Singleton {
    static Singleton instance = null;
    Singleton() { }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Command



The **Command** Pattern encapsulates a request as an object, thereby letting you parameterize other objects (the client) with different requests, queue or log requests, and support undoable operations.

- **encapsulates a request** by binding together a set of actions on a specific receiver. To achieve this, the **actions** and the **receiver** are packaged into one single object that only exposes one method: `execute()` (it can also have other like `undo()`, etc). An object from outside doesn't

know what actions will be performed, all it knows is that if they call **execute()** their request will be serviced.

- **parameterize other objects** with a command, in a sense that any command object can be passed to the client. The client doesn't know (or care) what the command is, all it does is call the **execute()** method.
- the Command Pattern makes it easy to implement **queue, log and undo operations**

The key in this pattern is an abstract **Command** class (or an interface), which declares an interface for executing operations. In its most simple form, this interface has an **execute()** method. A **concrete command** then has an **instance variable of the receiver**, whose methods are called in the **execute()**. The command itself doesn't perform any complex operations, those have to be done by the receiver, and that's what happens in the **execute()** method, receiver's methods get called.

Example

Let's say you have a remote control that controls the volume and the on/off state of your awesome sound system. For simplicity, let's assume that your remote control is actually a single button, which you can reprogram to do different actions (like volume up, volume down, on and off).



The first thing to do is to create the **Command** interface, which in this version, will contain the minimum (i.e. only the **execute()** method).

Next step will be to create four command classes (TurnOnCommand, TurnOffCommand, VolumeUpCommand and VolumeDownCommand). Each of those classes will have a reference to the **receiver** (in this case, the sound system), the **execute()** method and a constructor.

Now, just to demonstrate how the Command Pattern will be used, there will also be a button, which will have a reference to the command, a method to activate the command (**press()**) and a **setCommand()** method, so that different commands can be assigned to the same button.

```
public class SoundSystem {
    int soundLevel;
    int state; // 0:off 1:on

    public SoundSystem() {
        soundLevel = 0;
        state = 0;
    }

    public void volumeUp() {
        soundLevel++;
        System.out.println("Sound is at " + soundLevel);
    }
}
```

```
public void volumeDown() {
    soundLevel--;
    System.out.println("Sound is at " + soundLevel);
}

public void turnOn() {
    state = 1;
    System.out.println("ON!");
}

public void turnOff() {
    state = 0;
    System.out.println("OFF!");
}
}
```

```
public interface Command {
    public void execute();
}
```

```
public class TurnOnCommand implements Command {
    SoundSystem receiver;

    public TurnOnCommand(SoundSystem soundSystem) {
        receiver = soundSystem;
    }

    public void execute() {
        receiver.turnOn();
    }
}
```

```
public class TurnOffCommand implements Command {
    SoundSystem receiver;

    public TurnOffCommand(SoundSystem soundSystem) {
        receiver = soundSystem;
    }

    public void execute() {
        receiver.turnOff();
    }
}
```

```
public class VolumeUpCommand implements Command {
    SoundSystem receiver;
```

```
public VolumeUpCommand(SoundSystem soundSystem) {  
    receiver = soundSystem;  
}  
  
public void execute() {  
    receiver.volumeUp();  
}  
}
```

```
public class VolumeDownCommand implements Command {  
    SoundSystem receiver;  
  
    public VolumeDownCommand(SoundSystem soundSystem) {  
        receiver = soundSystem;  
    }  
  
    public void execute() {  
        receiver.volumeDown();  
    }  
}
```

```
public class Button {  
    Command activeCommand;  
  
    public Button() { }  
  
    public void setCommand(Command command) {  
        activeCommand = command;  
    }  
  
    public void press() {  
        activeCommand.execute();  
    }  
}
```

And just to show an example of usage, consider the main class below.

```
public class ButtonTest {  
    public static void main(String[] args) {  
        SoundSystem soundSystem = new SoundSystem();  
        Button button = new Button();  
  
        // Create some commands  
        Command turnOnCommand = new TurnOnCommand(soundSystem);  
        Command turnOffCommand = new TurnOffCommand(soundSystem);  
        Command volumeUpCommand = new VolumeUpCommand(soundSystem);  
        Command volumeDownCommand = new VolumeDownCommand(soundSystem);  
  
        // Assign an action to the button
```

```

button.setCommand(turnOnCommand);
button.press();

// Change the button's action
button.setCommand(volumeUpCommand);
button.press();
button.press();
button.press();

button.setCommand(volumeDownCommand);
button.press();

// Turn off after usage to save electric energy!
button.setCommand(turnOffCommand);
button.press();
}
}

```

Which produces the following output:

ON! Sound is at 1 Sound is at 2 Sound is at 3 Sound is at 2 OFF!

Note that the command invoker doesn't know what the command is doing or how it's, doing it, all it knows is that it has an **execute()** method. Also note that the **execute()** is doing very little himself, the main work is done by the receiver through some method.

Implementing the “Undo” Operation

In its simplest form, the **undo** operation is pretty straight forward. For example, what would the **undo** be for the **VolumeUpCommand**? Well, if that command increases the volume by 1, then the **undo** would decrease it by one. So the **VolumeUpCommand** would now look something like this:

```

public class VolumeUpCommand implements Command {
    SoundSystem receiver;

    public VolumeUpCommand(SoundSystem soundSystem) {
        receiver = soundSystem;
    }

    public void execute() {
        receiver.volumeUp();
    }

    public void undo() {
        receiver.volumeDown();
    }
}

```

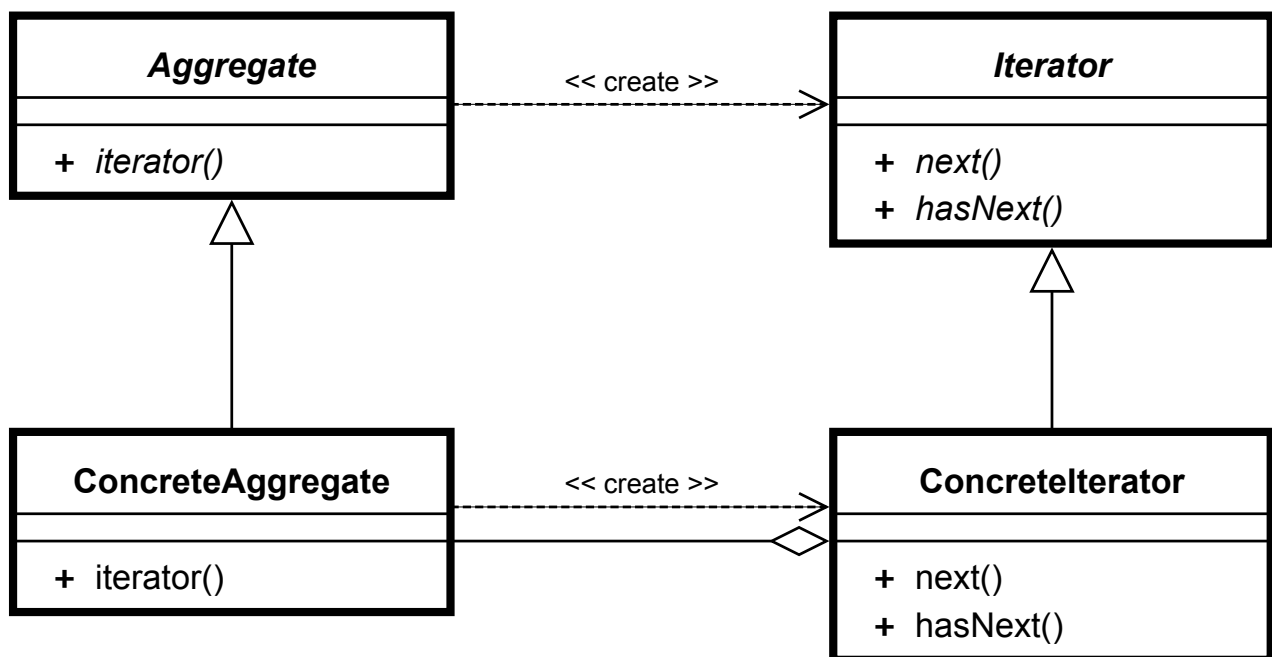
Besides that, the **undo()** method would also have to be added to the **Command** interface:

```
public interface Command {
    public void execute();
    public void undo();
}
```

You could also store all of the commands in a list, for example, and that would allow you to go through that list, calling **undo()** on every command object in there (that would be like a history of actions and you can undo multiple times, just like pressing **Ctrl + Z** on your keyboard multiple times will do more than one undo).

In a similar manner you could also extend the pattern to other functionality, such as queuing or logging.

Iterator



The **Iterator** Pattern provides a way to access the elements of an aggregate object (an object that contains an aggregate of something) sequentially without exposing its underlying representation.

So let's say you have an object that contains a collection of items and you want to iterate through every item. But there is a problem here: let's say you know what items you're accessing, but you don't know how they are stored, they might be in an **ArrayList**, **HashMap**, **LinkedList**, etc. Now how can you abstract the internal implementation (i.e. **how** the items are stored) away and have an interface that allows you to access those elements in a consistent manner, whichever the internal implementation is. The answer is (you've guessed it!) the **Iterator** Pattern.

So as mentioned before the **Iterator** Pattern allows you to step through the elements of an aggregate without knowing how the things are represented under the hood. It also allows to write **polymorphic code** that works with any of the aggregates (it doesn't matter if it's an `ArrayList`, a `HashTable` or even a `LinkedList`, it's an aggregate, that's all we care about).

The Iterator Pattern takes the responsibility of traversing the elements and gives that responsibility to the **iterator object**, not the **aggregate object**. This makes all the sense, since the aggregate object doesn't have to know how to iterate through items, all it has to know is how to store and manage them. It has to know which objects have been traversed already and which ones haven't.

In its simplest form, the **Iterator** Pattern consists of an Iterator interface, which contains two methods: * **next()**, which returns the next object in the aggregation that hasn't been iterated through yet. * **hasNext()** which returns a boolean indicating whether there are more items to be iterated through.

Example

Let's say you asked two of your friends to provide a list of their favorite songs. Your idea is simple: iterate through both of the lists (lists of songs, no one is talking about Java **Lists** here) and display the info of each one of the songs (title and author). You then realized that you didn't specify the format in which you want the songs to be sent (`ArrayList`, `HashTable`, `LinkedList`, etc) and you need to be able to iterate over the lists in a consistent way, without depending on the type of the aggregation.

You'll have to create a concrete iterator for every type of the aggregation, so let's say one of your friends send the songs in an **HashTable** and the other in a **LinkedList**.

Below is a possible solution using the **Iterartor** Pattern.

```
public class Song {
    String name;
    String artist;

    public Song(String name, String artist) {
        this.name = name;
        this.artist = artist;
    }

    public String getName() { return name; }
    public String getArtist() { return artist; }
}
```



```
import java.util.LinkedList;

public class EastCoastMusic {

    LinkedList<Song> eastCoastSongs;

    public EastCoastMusic() {
        eastCoastSongs = new LinkedList<Song>();

        Song s1 = new Song("Nas Is Coming", "NAS");
        Song s2 = new Song("What Up Gangsta", "50 Cent");
        Song s3 = new Song("Warrior", "Lloyd Banks");

        eastCoastSongs.add(s1);
        eastCoastSongs.add(s2);
        eastCoastSongs.add(s3);
    }

    public Iterator iterator() {
        return new LinkedListIterator(eastCoastSongs);
    }
}
```

```
import java.util.Hashtable;

public class WestCoastMusic {

    Hashtable<Integer, Song> westCoastSongs;
    int indexKey;

    public WestCoastMusic() {
        westCoastSongs = new Hashtable<Integer, Song>();
        indexKey = 0;

        Song s1 = new Song("Still Dre", "Dr.Dre");
        Song s2 = new Song("Let's Ride", "The Game");
        Song s3 = new Song("What's My Name?", "Snoop Dogg");

        westCoastSongs.put(indexKey++, s1);
        westCoastSongs.put(indexKey++, s2);
        westCoastSongs.put(indexKey++, s3);
    }

    public Iterator iterator() {
        return new HashTableIterator(westCoastSongs);
    }
}
```

```
public interface Iterator {
    public boolean hasNext();
}
```

```
public Object next();  
}
```

```
import java.util.LinkedList;  
  
public class LinkedListIterator implements Iterator {  
  
    LinkedList<Song> songs;  
  
    public LinkedListIterator(LinkedList<Song> songs) {  
  
        this.songs = songs;  
    }  
  
    public boolean hasNext() {  
        return songs.peek() != null;  
    }  
    public Object next() {  
        return songs.pop();  
    }  
}
```

```
import java.util.Hashtable;  
  
public class HashTableIterator implements Iterator {  
  
    Hashtable<Integer, Song> songs;  
    int indexKey; // songs are indexed by integers  
  
    public HashTableIterator(Hashtable<Integer, Song> songs) {  
        this.songs = songs;  
        indexKey = 0;  
    }  
  
    public Song next() {  
        return songs.get(indexKey++);  
    }  
  
    public boolean hasNext() {  
        return songs.get(indexKey) != null;  
    }  
  
}
```

```
public class Main {  
    public static void main(String args[]) {  
  
        WestCoastMusic westCoastMusic = new WestCoastMusic();  
        EastCoastMusic eastCoastMusic = new EastCoastMusic();  

```

```
// First, iterate through the West Coast Music list
displaySongs(westCoastMusic.iterator());
// Now, iterate through the East Coast Music list
displaySongs(eastCoastMusic.iterator());
}

public static void displaySongs(Iterator iterator) {
    Song song;

    while(iterator.hasNext()) {
        song = (Song)iterator.next();
        System.out.println("Name: " + song.getName() +
            " Artist: " + song.getArtist() + "\n");
    }
}
}
```

The application above produces the following output:

```
Name: Still Dre Artist: Dr.Dre

Name: Let's Ride Artist: The Game

Name: What's My Name? Artist: Snoop Dogg

Name: Nas Is Coming Artist: NAS

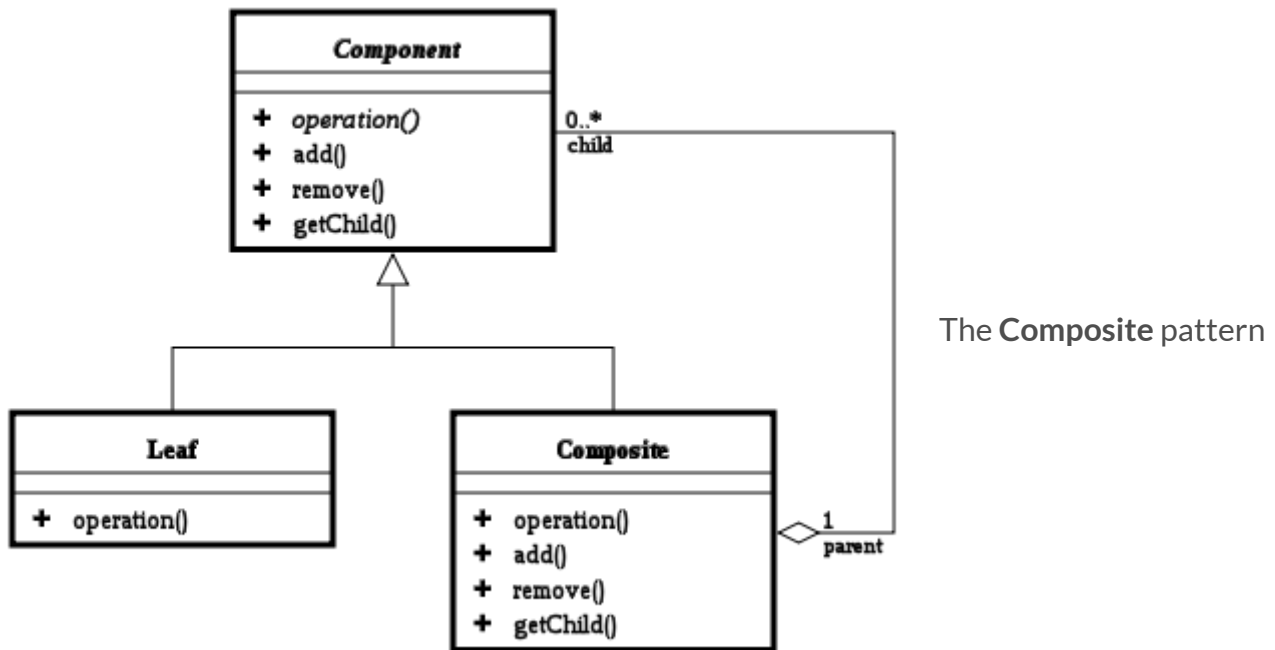
Name: What Up Gangsta Artist: 50 Cent

Name: Warrior Artist: Lloyd Banks
```

Note how by using the **Iterator** pattern we decoupled our application (in this case just the **Main** class) from the underlying implementation of each aggregation.

All we had to do to our friends code is implement the **Iterator** interface (which we defined) and add the **iterator()** method to their classes. If we didn't use the pattern, when iterating through the list of songs in the **Main** class, we would have to make a separate method for each aggregation we wanted to traverse.

Composite



allows you treat objects and compositions of objects uniformly. It allows you to **compose objects into tree structures** to represent **part-whole-hierarchies** (components can be divided into smaller and smaller components).

The idea here is really simple: you have an object A that supports some operations, then you have another object B that is an aggregation of objects of type A. The **composite** pattern allows you treat both of them using the same interface. Let's say object A is a **sheep** and supports the **sheer()** operation. So to shear a **single sheep** you simply call the **sheer** method on a **sheep object**. Now, object B is a **group of sheep**. How do you **shear a group of sheep**? Simple, **the same way you would shear an individual sheep**: by calling the **sheer** method on the object B. Note that the object B can contain sheep or other objects of the same type (of the type of object B).

So basically, the **composite** pattern allows us to **ignore** the differences between compositions of objects and individual objects.

Example

In this example, we will use the sheep example introduced above. So the idea is simple: each sheep has a name and the only operation it supports is **sheer()**.

Now, you shear a sheep by simply calling the **sheer()** method on it, but how can you shear a **group** of sheep? The same way, by calling the **sheer()** method on it. As mentioned before, the goal is to **ignore** the differences between an individual object and a group of objects, so we will define an abstract class, which is extended by both: an **individual sheep** and a **group of sheep** (here, it'll be called **SheepComponent**). This approach will also allow **SheepComponents** to contain other **SheepComponents**.

Below is a possible solution to the problem:

```
public abstract class SheepComponent {

    public void add(SheepComponent sheepComponent) {
        throw new UnsupportedOperationException();
    }

    public void remove(SheepComponent sheepComponent) {
        throw new UnsupportedOperationException();
    }

    public SheepComponent getComponent(int index) {
        throw new UnsupportedOperationException();
    }

    public String getSheepName() {
        throw new UnsupportedOperationException();
    }

    public void shear() {
        throw new UnsupportedOperationException();
    }
}
```

```
public class Sheep extends SheepComponent {
    String name;

    public Sheep(String name) {this.name = name; }

    @Override
    public String getSheepName() { return name; }

    @Override
    public void shear() {
        System.out.println("Sheering " + getSheepName() +
            "...\\n");
    }
}
```

```
import java.util.ArrayList;

public class SheepGroup extends SheepComponent {

    String groupName;
    ArrayList<SheepComponent> sheepComponents;

    public SheepGroup(String name) {
        sheepComponents = new ArrayList<SheepComponent>();
        groupName = name;
    }

    public String getGroupName() {return groupName; }
```

```

@Override
public void add(SheepComponent sheepComponent) {
    sheepComponents.add(sheepComponent);
}

@Override
public void remove(SheepComponent sheepComponent) {
    sheepComponents.remove(sheepComponent);
}

@Override
public SheepComponent getComponent(int index) {
    return sheepComponents.get(index);
}

@Override
public void sheer() {
    Sheep sheep;
    int numOfSheep = sheepComponents.size();
    System.out.println("Group Name: " + groupName +
        "\n" + "---" + "\n");

    // NOTE: The Iterator pattern is more suitable here
    for (int i = 0; i < numOfSheep; i++) {
        sheepComponents.get(i).sheer();
    }
}
}

```

```

public class Main {

    public static void main(String args[]) {
        SheepGroup sg1 = new SheepGroup("Sheep Group 1");

        Sheep s1 = new Sheep("Sheep 1");
        Sheep s2 = new Sheep("Sheep 2");
        Sheep s3 = new Sheep("Sheep 3");
        Sheep s4 = new Sheep("Sheep 4");
        Sheep s5 = new Sheep("Sheep 5");

        sg1.add(s2);
        sg1.add(s3);
        sg1.add(s4);
        sg1.add(s5);

        s1.sheer();
        sg1.sheer();

        SheepGroup sg2 = new SheepGroup("SheepGroup 2");
        sg2.add(s1);
        // Now, compose a group of sheep with another group of sheep
        sg2.add(sg1);
        Sheep s6 = new Sheep("Sheep 6");
    }
}

```

```
sg2.add(s6);  
sg2.sheer();  
}  
}
```

The output of the application above is:

```
Sheering Sheep 1...  
  
Group Name: Sheep Group 1  
---  
  
Sheering Sheep 2...  
  
Sheering Sheep 3...  
  
Sheering Sheep 4...  
  
Sheering Sheep 5...  
  
Group Name: SheepGroup 2  
---  
  
Sheering Sheep 1...  
  
Group Name: Sheep Group 1  
---  
  
Sheering Sheep 2...  
  
Sheering Sheep 3...  
  
Sheering Sheep 4...  
  
Sheering Sheep 5...  
  
Sheering Sheep 6...
```

Visitor



The **Visitor** pattern allows you to add new methods to the classes without changing them too much. You can add operations to a **Composite** structure without changing the structure itself.

Visitor is very useful when you have some unrelated operations that need to be performed on an object in an object structure and you don't want to "pollute" their classes by adding new methods to them to perform those operations. This pattern allows you to keep related operations together defined in a **separate class**. This is very useful when your object structure is shared by many

applications, but only some of those applications actually use those extra operations, since visitor allows you to put those operations only in the applications that need them.

So let's say you have the class code written and now you're wondering what changes you'll have to make to the class for it to support the **visitor** pattern. Okay, make sure you have a piece of paper and pen to write them down. Not really, actually all you have to do is **add an accept() method to your class** (it's a convention to call the method accept(), you can name it whatever you want). Yes, that's it! All you have to do for your classes to support the **visitor** pattern is **add a single method**. That's what we're basically going to do, except we're going to put this method in an **Interface**, we'll call it "**Visitable**" (this name makes sense, since the class **will be visited** by a visitor).

Okay, now how do we create those "visitors"? Simple, first we create another interface called **Visitor** (again, this name is just a convention). And what will that interface contain? It will contain **the new operations we want to add**. Now, since we're putting them in an interface, those operations have to be related (i.e. be somehow related, belong to the same "group").

Now you might be wondering what will you do if you had more than just one group of operations? Well, in that case you would have to add another **accept()** method to your **Visitable** interface (use method overloading).

Example

Let's say you have a store and your store sells three products: drinks, food and gadgets. Each of those three objects has a price. Here is how your code looks now:

```
public class Drink {  
    double price;  
  
    public Drink(double price) { this.price = price; }  
    public double getPrice() { return price; }  
}
```

```
public class Gadget {  
    double price;  
  
    public Gadget(double price) { this.price = price; }  
    public double getPrice() { return price; }  
}
```

```
public class Food {  
    double price;  
  
    public Food(double price) { this.price = price; }  
}
```



```
public double getPrice() { return price; }  
}
```

(Okay, a better choice would be to add an abstract class from which those methods derive, but I want to enforce the idea that **the classes don't have to be related in any way**).

Now you are asked to be able to calculate taxes for each product(the tax is 21%). That's where the visitor comes in. First you'll create a **Visitor** interface and add three **visit()** methods there. Then you'll create a new class: a **concrete visitor**, which implements the **Visitor** interface. Then, we'll create a **Visitable** interface and add the **accept()** method to it, which takes an object of type **Visitor** as an argument. The **Food**, **Drink** and **Gadgets** classes will implement the **Visitable** interface. Now your code will look something like this:

```
public interface Visitable {  
    public double accept(Visitor visitor);  
}
```

```
public interface Visitor {  
    public double visit(Drink drink);  
    public double visit(Food food);  
    public double visit(Gadget gadget);  
}
```

```
public class NormalTaxVisitor implements Visitor {  
    final double TAX_VALUE = 0.21;  
  
    public double visit(Drink drink) {  
        return drink.getPrice() * TAX_VALUE + drink.getPrice();  
    }  
  
    public double visit(Food food){  
        return food.getPrice() * TAX_VALUE + food.getPrice();  
    }  
    public double visit(Gadget gadget) {  
        return gadget.getPrice() * TAX_VALUE + gadget.getPrice();  
    }  
}
```

```
public class Drink implements Visitable {  
  
    double price;  
  
    public Drink(double price) { this.price = price; }  
    public double getPrice() { return price; }  
}
```

```
public double accept(Visitor visitor) { return visitor.visit(this); }  
}
```

```
public class Food implements Visitable {  
  
    double price;  
  
    public Food(double price) { this.price = price; }  
    public double getPrice() { return price; }  
  
    public double accept(Visitor visitor) { return visitor.visit(this); }  
  
}
```

```
public class Gadget implements Visitable {  
  
    double price;  
  
    public Gadget(double price) { this.price = price; }  
    public double getPrice() { return price; }  
  
    public double accept(Visitor visitor) { return visitor.visit(this); }  
  
}
```

Now you are asked to add yet another type of tax: a holiday tax. It's basically the same as the previous one, except the tax value is now 18% and you always subtract two cents (0.02) from the value after tax (here we'll ignore that you can get negative or zero values). Well, that's simple, just create a new class **HolidayTaxVisitor**, implement the **Visitor** interface and add override the methods with the requested functionality. Your new class will look something like this:

```
public class HolidayTaxVisitor implements Visitor {  
    final double TAX_VALUE = 0.18;  
    final double DISCOUNT = 0.02;  
  
    public double visit(Drink drink) {  
        return drink.getPrice() * TAX_VALUE + drink.getPrice() - DISCOUNT;  
    }  
  
    public double visit(Food food){  
        return food.getPrice() * TAX_VALUE + food.getPrice() - DISCOUNT;  
    }  
  
    public double visit(Gadget gadget) {  
        return gadget.getPrice() * TAX_VALUE + gadget.getPrice() - DISCOUNT;  
    }  
  
}
```

And now an example application:

```
public class Main {  
    public static void main(String[] args) {  
        Drink d = new Drink(1.50);  
        Food f = new Food(2.75);  
        Gadget g = new Gadget(7.25);  
  
        NormalTaxVisitor ntv = new NormalTaxVisitor();  
        HolidayTaxVisitor htv = new HolidayTaxVisitor();  
  
        System.out.println("Drink price after normal tax: " +  
            d.accept(ntv) + "\n");  
  
        System.out.println("Drink price after holiday tax: " +  
            d.accept(htv) + "\n");  
  
        System.out.println("Food price after normal tax: " +  
            f.accept(ntv) + "\n");  
  
        System.out.println("Food price after holiday tax: " +  
            f.accept(htv) + "\n");  
  
        System.out.println("Gadget price after normal tax: " +  
            g.accept(ntv) + "\n");  
  
        System.out.println("Gadget price after holiday tax: " +  
            g.accept(htv) + "\n");  
    }  
}
```

The output of the Main class above is the following:

```
Drink price after normal tax: 1.815  
  
Drink price after holiday tax: 1.75  
  
Food price after normal tax: 3.3275  
  
Food price after holiday tax: 3.225  
  
Gadget price after normal tax: 8.7725  
  
Gadget price after holiday tax: 8.535
```

Now what if you wanted a different group of operations, not related in any way to tax calculation? Let's say you wanted to be able to print out the name of the class of which an object is instance of. Well, in that case you'd have to create a new visitor interface, for example **NameVisitor**, create a concrete instance, which implements that interface, for example **NormalNameVisitor** and add a new **accept()** method to the **Visitable** interface.

```
public interface NameVisitor {  
    public String visit(Drink drink);  
    public String visit(Food food);  
    public String visit(Gadget gadget);  
}
```

```
public class NormalNameVisitor implements NameVisitor {  
    public String visit(Drink drink) { return "Drink"; }  
    public String visit(Food food) { return "Food"; }  
    public String visit(Gadget gadget) { return "Gadget"; }  
}
```

```
public interface Visitable {  
    public double accept(Visitor visitor);  
    public String accept(NameVisitor visitor);  
}
```

Factory Method



The idea behind the **Factory Method** pattern is to be able to decide which class you want to instantiate **dynamically** (i.e. at **runtime**). Basically you will have a method that will return one of several possible classes that **share a common superclass**. The **factory method** has this name, because it's responsible for "manufacturing" an object. This pattern allows you to encapsulate object creation in a method: the **factory method**.

The typical implementation uses a single class with a single method (the **factory method**) and this method returns an object based on the input passed as an argument. Which object is it? Well, that depends on the passed parameter.

Example

Let's say that you have a **Fruit** superclass that has two subclasses: **Apple** and **Orange**. You want your application to be able to instantiate each one of the subclasses dynamically, because you don't know which fruit you'll need (it might depend on the user input, for example).

All you'll have to do is create a **FruitFactory** class and add a **makeFruit()** method to it, which accepts, for example a string and returns a **Fruit** (note it returns the generic **Fruit** object, not a concrete **Apple** or **Orange**).

To simplify things, let's say that to create an apple you pass the "Apple" string as an argument and to create an orange you use the "Orange" string.

Below is a possible solution:

```
public abstract class Fruit {  
    final String type;  
  
    public Fruit(String type) { this.type = type; }  
  
    public String getType() { return type; }  
}
```

```
public class Apple extends Fruit {  
    public Apple() { super("Apple"); }  
}
```

```
public class Orange extends Fruit {  
    public Orange() { super("Orange"); }  
}
```

```
public class FruitFactory {  
    public Fruit makeFruit(String type) {  
        Fruit fruit = null;  
  
        if (type == "Apple")  
            fruit = new Apple();  
  
        else if(type == "Orange")  
            fruit = new Orange();  
  
        return fruit;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Fruit fruit;  
        FruitFactory fruitFactory = new FruitFactory();  
  
        fruit = fruitFactory.makeFruit("Apple");  
        System.out.println("The fruit is an " + fruit.getType() + ".");  
  
        fruit = fruitFactory.makeFruit("Orange");  
        System.out.println("The fruit is an " + fruit.getType() + ".");  
    }  
}
```

```

    }
}

```

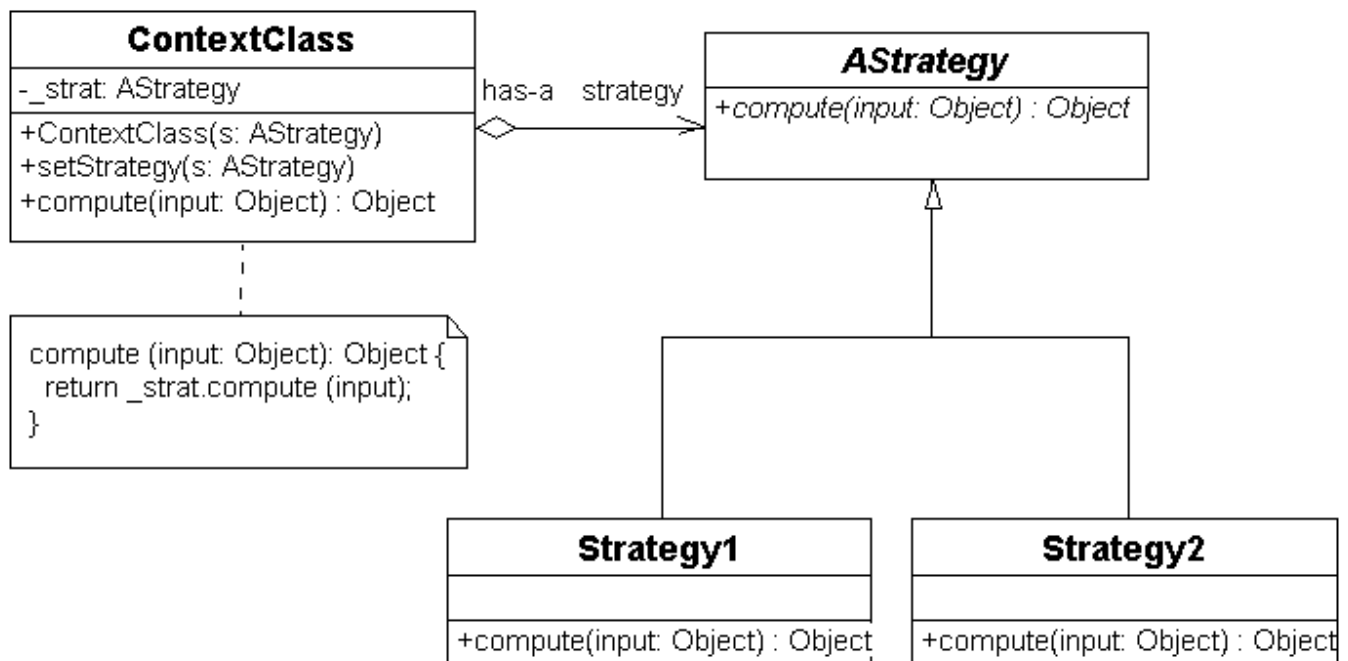
The Main class above produces the following output:

```

The fruit is an Apple.
The fruit is an Orange.

```

Strategy



The **Strategy** pattern lets you define a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithms vary independently from clients that use it.

This pattern is very useful in cases when you have many **hierarchically related** classes that differ only in their behaviour. Strategy allows you to configure every individual class in the hierarchy with **one of the many behaviours**. Another use case for it is when you have many variants of an algorithm.

Example

Let's say you have the class structure below:

```

public abstract class Animal {
    String name;
    String sound;

    public Animal(String name, String sound) {

```

```
this.name = name;
this.sound = sound;
}

public void makeSound() {System.out.println(name + " says: " + sound + ".\n");}
```

```
public class Dog extends Animal {
    public Dog(String name) { super(name, "Ruff Ruff"); }
}
```

```
public class Cat extends Animal {
    public Cat(String name) { super(name, "Meow"); }
}
```

Now you are asked to create a **Bird** class, which is also a subclass of the **Animal** class. The thing about the **Bird** class, is that it needs to be able to fly. You must also keep a common interface for all of the three subclasses, that means you can't just add a **fly()** method to the **Bird** class, from now on every animal must print "Can't fly!" or "Flying!" when the **fly()** method is invoked on an **Animal** object.

So how can we solve this? Well, we surely can just add a "fly()" method to the abstract class (or implement an interface) and override its behaviour in every subclass. But this creates several problems:

- code duplication.
- the change in super class will **break** the code in subclasses.
- implementing an interface which only has one method in it is usually a bad approach.

So what can we do to avoid those problems? Well, since all of our classes only differ in one behaviour (some fly and some don't) it's the right time to use the **Strategy** pattern!

What we'll do instead is instead of **inheriting** the ability to fly, we will compose the class with objects which have the correct ability built-in (in our case we will only add one object).

This approach will give us many benefits, for example:

- it's possible to create many types of "flying" without affecting the **Animal** class or any of its subclasses (since the "types of flying" are separate classes).
- we are **decoupling**, that means we are **encapsulating the behavior that varies** (in our case the behaviour is the ability to fly).

- it's possible to change the ability at **runtime**. That means we can, for example, create an object that couldn't fly before and make it flyable!

What we'll do is create an interface called **Fly**, it will only have one method: **fly()** (which is the behaviour we want to encapsulate) and then for every "type of flying" we will create a subclass, each one implementing the **Fly** interface (in our case it will be two classes: **CanFly** and **CantFly**). We'll also have to add a new variable to the **Animal** class of type **Fly**.

Anyways, here is a possible solution using the **Strategy** pattern, with an example of a client application that uses it:

```
public interface Fly {  
    public String fly();  
}
```

```
public class CanFly implements Fly {  
    public String fly() { return "Flying!"; }  
}
```

```
public class CantFly implements Fly {  
    public String fly() { return "Can't fly!"; }  
}
```

```
public abstract class Animal implements Fly {  
    String name;  
    String sound;  
  
    Fly flyBehaviour; // new variable!  
  
    public Animal(String name, String sound) {  
        this.name = name;  
        this.sound = sound;  
    }  
  
    public void makeSound() {System.out.println(name + " says: " + sound + ".\n");}  
  
    /* New methods below */  
    public String fly() { return name + ": " + flyBehaviour.fly() + "\n"; }  
  
    // allow to change the flying ability dynamically  
    public void setFlyingBehaviour(Fly flyBehaviour) {  
        this.flyBehaviour = flyBehaviour;  
    }  
}
```



```
public class Dog extends Animal {  
    public Dog(String name) {  
        super(name, "Ruff Ruff");  
        flyBehaviour = new CantFly();  
    }  
}
```

```
public class Cat extends Animal {  
    public Cat(String name) {  
        super(name, "Meow");  
        flyBehaviour = new CantFly();  
    }  
}
```

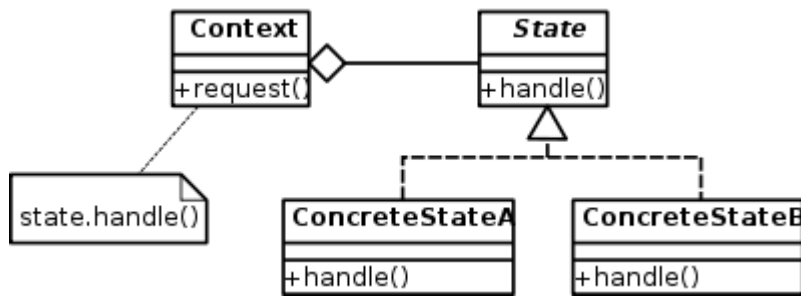
```
public class Bird extends Animal {  
    public Bird(String name) {  
        super(name, "Tweet");  
        flyBehaviour = new CanFly();  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Animal cat = new Cat("Mittens");  
        Animal dog = new Dog("Rufus");  
        Animal bird = new Bird("Tweetie");  
  
        System.out.println(cat.fly());  
        System.out.println(dog.fly());  
        System.out.println(bird.fly());  
  
        // Rufus can fly now!  
        dog.setFlyingBehaviour(new CanFly());  
        System.out.println(dog.fly());  
    }  
}
```

The main class produces the following output:

```
Mittens: Can't fly!  
  
Rufus: Can't fly!  
  
Tweetie: Flying!  
  
Rufus: Flying!
```

State



The **State** pattern allows to modify its behavior when its internal state changes. The object will also appear to change its class. It actually mimics the finite state machine: you have different states and certain actions make you transition from one state to another.

This pattern encapsulates every state into a separate class (all those states derive from a common class). If you have an object and you completely change its behavior, it appears that the object changed its class (in reality you will just be using composition to give the appearance of class change by referencing different state objects).

The **State** pattern is a great way to get rid of if statements, but it usually requires a large amount of extra classes to be created.

Each state has a reference to the object whose state it represents and is able to change it dynamically (usually through a setter method).

Example

Too keep it simple, consider the following problem: You have an automatic door with two buttons: **open** and **close**. You have four states:

- open
- closed

The transitions between the states are intuitive: for example, if a door is closed, by pressing the **open** button, the door will go to the “open” state. To simplify, you can’t interrupt an opening/closing action (there are no intermediate **DoorOpeningState** and **DoorClosingState** classes).

What you’ll do is create an abstract **DoorState** class, with two operations:

- openDoor()
- closeDoor()

Then, you will create two subclasses: **DoorOpenState**, **DoorClosedState**. Each one of the states will have a reference to the **Door** object (passed as an argument in the constructor) whose state it

represents and it will be able to change its state through the **setStateMethod()** which will be a part of the Doors interface.

In this example, the **Door** class will contain references to every possible state which will be instantiated when a new **Door** object is created.

Below is a possible solution to the problem.

```
public abstract class DoorState {
    Door door;
    public abstract void open();
    public abstract void close();

    public DoorState(Door door) { this.door = door; }
}
```

```
public class DoorOpenState extends DoorState {
    public DoorOpenState(Door door) { super(door); }

    public void open() { System.out.println("Door is already open!\n"); }
    public void close() {
        System.out.println("Closing door... New state: DoorClosedState.\n");
        door.setState(door.DOOR_CLOSED_STATE);
    }
}
```

```
public class DoorClosedState extends DoorState {
    public DoorClosedState(Door door) { super(door); }

    public void open() {
        System.out.println("Opening door... New state: DoorOpen.\n");
        door.setState(door.DOOR_OPEN_STATE);
    }
    public void close() { System.out.println("Door is already closed!\n"); }
}
```

```
public class Door {
    public final DoorState DOOR_OPEN_STATE = new DoorOpenState(this);
    public final DoorState DOOR_CLOSED_STATE = new DoorClosedState(this);

    DoorState state;

    public Door() {
        // initially, the door is closed
        setState(DOOR_CLOSED_STATE);
    }
}
```

```
}

public void open() { state.open(); }
public void close() { state.close(); }

public void setState(DoorState newState) { state = newState; }
}
```

```
public class Main {

    public static void main(String[] args) {
        Door door = new Door();

        door.open();
        door.close();
        door.close();
    }
}
```

The output of the application above is:

```
Opening door... New state: DoorOpen.

Closing door... New state: DoorClosedState.

Door is already closed!
```

Abstract Factory

The **Abstract Factory** pattern provides an interface for creating families of related or depended objects without specifying their concrete classes.

The idea if an abstract factory is to define a common interface for a group of factories, then use those factories to produce objects. It provides an abstract type for creating a family of products and **subclasses** of that abstract type define how those products are produced. To use a specific factory, you'll have to instantiate it and pass into some code that is written against the **abstract type**.

Factory Method vs Abstract Factory

It's important to understand the difference between the **Abstract Factory** and the **Factory Method** patterns. While both are really good at **decoupling applications from specific**

implementations, they do it in different ways. Both of the patterns **create objects**, but they do it in a different way:

- **Factory Method** uses **classes** - the objects are created through **inheritance**.
- **Abstract Factory** uses **objects** - the objects are created through **object composition**.

Often the **Abstract Factory** uses **Factory Methods** to implement its concrete factories. The concrete factories use a factory method to create their products (they are used purely to create products).

Example

Consider the example from the **Factory Method** section. Well, here we also have fruit, except that one of the farmers has discovered a new sort of fruit: the **Advanced Fruit**. What distinguishes a “normal” **Fruit** from an **AdvancedFruit**? Well, unlike the “normal” **Fruit**, all **AdvancedFruit** have guns and engines!

The farmer in question is only producing **AdvancedApple** and **AdvancedOrange** types of **AdvancedFruit**, so those are the ones that will be covered in this example.

What distinguished an **AdvancedApple** from an **AdvancedOrange**? Well, while **AdvancedApple** has a **BlueGun** and a **V16Engine**, the **AdvancedOrange** has a **RedGun** and a **V8Engine**.

Hmmm... Isn't that a good place to use the **Abstact Factory** pattern to abstract out the specific gun and engine creation? Note, that both, advanced apples and oranges have guns and engines, except that they are of different types. We sure can come up with a common abstract interface to produce those pieces of equipment and then have two specific implementations of those: one for the advanced apples and one for the advanced oranges.

When defining an abstract interface for out factories we have to question **what makes an AdvancedFruit an AdvancedFruit**? Well, it's the fact that every **AdvancedFruit** has an **Engine** and a **Gun**.

Our abstract interface only will have two methods: **makeEngine()** and **makeGun()** (we'll call it the **AdvancedFruitFactory**). Then in **AdvancedAppleFactory** and **AdvancedOrangeFactory** we'll make those methods return the correct **Engine** and **Gun** objects.

Now every advanced fruit will receive a concrete factory in its constructor, because it's through a method call on the **AdvancedFruit** object, that the specific engine and gun will be “given” to it.

Below is a possible solution to the problem:



```
public abstract class Fruit {  
    final String type;  
  
    public Fruit(String type) { this.type = type; }  
  
    public String getType() { return type; }  
}
```

```
public abstract class AdvancedFruit extends Fruit {  
    Engine engine;  
    Gun gun;  
  
    public AdvancedFruit(String name) { super(name); }  
  
    public void printInfo() { System.out.println("Name: " + type +  
        " Gun: " + gun.toString() + " Engine: " + engine.toString() + "\n"); }  
  
    public abstract void makeFruit();  
}
```

```
public class AdvancedApple extends AdvancedFruit {  
    AdvancedFruitFactory aff;  
  
    public AdvancedApple(String type, AdvancedFruitFactory aff) {  
        super(type);  
        this.aff = aff;  
    }  
  
    public AdvancedApple(AdvancedFruitFactory aff) { this("Generic AdvancedApple", aff); }  
  
    public void makeFruit() {  
        engine = aff.makeEngine();  
        gun = aff.maneGun();  
    }  
}
```

```
public class AdvancedOrange extends AdvancedFruit {  
    AdvancedFruitFactory aff;  
  
    public AdvancedOrange(String type, AdvancedFruitFactory aff) {  
        super(type);  
        this.aff = aff;  
    }  
  
    public AdvancedOrange(AdvancedFruitFactory aff) { this("Generic AdvancedOrange", aff); }  
  
    public void makeFruit() {  
        engine = aff.makeEngine();  
        gun = aff.maneGun();  
    }  
}
```

```

    }
}

```

```

public abstract class AdvancedFruitBuilding {

    // Hey, that's the Factory Method! Except now it "protected" and not "public"
    protected abstract AdvancedFruit makeAdvancedFruit(String typeOfFruit);

    public AdvancedFruit orderAdvancedFruit(String typeOfFruit) {
        // First, create the requested AdvcancedFruit
        AdvancedFruit af = makeAdvancedFruit(typeOfFruit);

        // Now, complete each AdvancedFruit with its specific properties
        af.makeFruit();

        return af;
    }
}

```

```

public class NaturalAdvancedFruitBuilding extends AdvancedFruitBuilding {
    /* A Specific Builder */

    // Hey! It's a factory method! It's used to create concrete products
    protected AdvancedFruit makeAdvancedFruit(String typeOfFruit) {
        AdvancedFruit af = null;

        if (typeOfFruit == "Apple") {
            // For Advanced Apples, we will use the AdvancedAppleFactory
            AdvancedFruitFactory aff = new AdvancedAppleFactory();
            af = new AdvancedApple(aff);
        }
        else if (typeOfFruit == "Orange") {
            // For Advanced Oranges, we will use the AdvancedOrangeFactory
            AdvancedFruitFactory aff = new AdvancedOrangeFactory();
            af = new AdvancedOrange(aff);
        }

        return af;
    }
}

```

```

public interface AdvancedFruitFactory {
    /* The abstract factory interface */
    public Engine makeEngine();
    public Gun maneGun();
}

```

```
public class AdvancedAppleFactory implements AdvancedFruitFactory {  
    /* A specific factory that produces engines and guns for adv. apples */  
    public Engine makeEngine() { return new V16Engine(); }  
    public Gun maneGun() { return new BlueGun(); }  
}
```

```
public class AdvancedOrangeFactory implements AdvancedFruitFactory {  
    /* A specific factory that produces engines and guns for adv. oranges */  
    public Engine makeEngine() { return new V8Engine(); }  
    public Gun maneGun() { return new RedGun(); }  
}
```

```
public interface Engine {  
  
    public String toString();  
}
```

```
public interface Gun {  
  
    public String toString();  
}
```

```
public class V16Engine implements Engine {  
  
    public String toString() { return "V16 Engine"; }  
}
```

```
public class V8Engine implements Engine {  
  
    public String toString() { return "V8 Engine"; }  
}
```

```
public class BlueGun implements Gun {  
  
    public String toString() { return "Blue Gun"; }  
}
```

```
public class RedGun implements Gun {  
    public String toString() { return "Red Gun"; }  
}
```



```
}
```

```
public class Main {  
    public static void main(String[] args) {  
        AdvancedFruitBuilding afb = new NaturalAdvancedFruitBuilding();  
        AdvancedFruit apple = afb.orderAdvancedFruit("Apple");  
        AdvancedFruit orange = afb.orderAdvancedFruit("Orange");  
  
        apple.printInfo();  
  
        orange.printInfo();  
    }  
}
```

The output of the **Main** class is as follows:

```
Name: Generic AdvancedApple Gun: Blue Gun Engine: V16 Engine
```

```
Name: Generic AdvancedOrange Gun: Red Gun Engine: V8 Engine
```

Template Method

The **Template Method** defines the skeleton of an algorithm in a method, deferring some steps to the subclasses. This pattern allows the subclasses to change certain parts of the algorithm without changing the algorithm's structure.

So let's say you have a generic algorithm, for example an algorithm for making tea. The general(abstract) idea is simple:

1. Boil water.
2. Put tea bag in cup.
3. Pour in the boiled water(still hot).
4. Add condiments.

The general algorithm is simple, however it can vary. Some prefer tea bags, while others prefer to use loose leafs. Some like to pour in water at 100°C, while others prefer it a bit cooler. Some like to add condiments and some don't.

So there are variations to the algorithm, but in essence, the algorithm doesn't change.

That's where the **Template Method** comes in. Basically you define an abstract template method, and then let the subclasses customize parts of it (or even the whole algorithm). This customization is obtained using **method overriding** and the so called **hooks** (methods that return True/False).

Example

Consider two types of cars: **SportsCar** and **CityCar**. The generic algorithm for car creation is the following:

1. Add chassis.
2. Add body.
3. Add wheels.
4. Add windows.
5. Add air conditioning.
6. Add radio.

That's the general algorithm that all cars follow, the **SportsCar** however, doesn't have air conditioning or radio and it uses a different type of wheels (sports wheels), while the **CityCar** uses regular wheels has A/C and radio. All of the cars **must have** a chassis, a body and windows.

A possible solution using the **Template Method** can be found right below:

```
public abstract class Car {  
  
    // The template method is final, so that the  
    // subclasses can't change the generic algorithm,  
    // they can only customize it.  
    public final void makeCar() {  
  
        addChassis();  
        addBody();  
        addWheels();  
        addWindows();  
  
        if(carNeedsAC())  
            // only add A/C if needed  
            addAC();  
  
        if (carNeedsRadio())  
            // only add radio if needed  
            addRadio();  
    }  
  
    public void addChassis() {  
        // All cars have the same chassis,  
        // this can't be changed.  
        System.out.println("Adding chassis...\n");  
    }  
}
```

```

    }

    public void addBody() {
        System.out.println("Adding body...\n");
    }

    public void addWindows() {
        System.out.println("Adding windows...\n");
    }

    public void addAC() {
        System.out.println("Adding A/C...\n");
    }

    public void addRadio() {
        System.out.println("Adding radio...\n");
    }

    // Let specific cars, explicitlyly specfy the
    // wheels they want to use.
    public abstract void addWheels();

    // "Hooks" : let the cars costumize if they need
    // some parts or not.
    public abstract boolean carNeedsAC();
    public abstract boolean carNeedsRadio();
}

```

```

public class SportsCar extends Car {
    public void addWheels() {
        System.out.println("Adding sports wheels...\n");
    }

    public boolean carNeedsAC() {
        return false;
    }

    public boolean carNeedsRadio() {
        return false;
    }
}

```

```

public class CityCar extends Car {
    public void addWheels() {
        System.out.println("Adding regular wheels...\n");
    }

    public boolean carNeedsAC() {
        return true;
    }

    public boolean carNeedsRadio() {

```

```

        return true;
    }
}

```

```

public class Main {
    public static void main(String[] args) {
        Car sc = new SportsCar();
        Car cc = new CityCar();

        System.out.println("Sports Car Production Begin!\n---\n");
        sc.makeCar();
        System.out.println("Sports Car Production End!\n---\n");

        System.out.println("Sports Car Production Begin!\n---\n");
        cc.makeCar();
        System.out.println("City Car Production End!\n---\n");

    }
}

```

The output of the client application is:

```

Sports Car Production Begin!
---

Adding chassis...

Adding body...

Adding sports wheels...

Adding windows...

Sports Car Production End!
---

Sports Car Production Begin!
---

Adding chassis...

Adding body...

Adding regular wheels...

Adding windows...

Adding A/C...

Adding radio...

City Car Production End!
---

```

- The template method is declared **final** because we don't want the subclasses to be changing the algorithm.
- Have some "hooks" that return booleans, to decide whether we should run a certain method or not.
- Make a method abstract when you want to force the user to override the method.
- Create a hook when you want to make a part of your algorithm totally optional.

Observer

The **Observer** pattern defines a one-to-many dependency between objects so that when one object changes state all of its dependents get notified and updated automatically.

A good analogy is a newspaper. A newspaper (the **Subject**) can have zero or more subscribers (**Observers**) and every time a new edition comes out (the newspaper, i.e. the **Subject** changes), the subscribers receive the new edition in the mail (i.e. the **Observers** get notified and updated about that change).

Example

Consider you have a **Subject**: a Facebook user. Anyone can subscribe to it and they receive updates every time the **Subject** updates its status or "likes" something.

Of course you could put each "observer" in an infinite loop testing whether there has been a new status update or a new like, but that would be extremely uneficient! The **Template** pattern provides a much more elegant and resourse-friendly solution.

Below is a possible solution to the problem:

```
public interface Subject {  
    public void registerObserver(Observer o);  
    public void unregisterObserver(Observer o);  
    public void notifyObservers();  
}
```

```
public interface Observer {  
    public void update(String status, String contentLiked);  
}
```

```
/* This is the user to whom users "subscribe" */
import java.util.ArrayList;

public class FacebookUserSubject implements Subject {
    ArrayList<Observer> observers;
    String status;
    String lastContentLiked;

    public FacebookUserSubject() {
        observers = new ArrayList<Observer>();
        status = "";
        lastContentLiked = "";
    }

    public void registerObserver(Observer o) {
        observers.add(o);
    }

    public void unregisterObserver(Observer o) {
        // get the index of the observer
        int index = observers.indexOf(o);
        // unregister the observer
        observers.remove(index);
    }

    public void notifyObservers() {
        // All of the oservers have the update() method,
        // that's the one we use to notify them of a change.

        for(Observer o : observers)
            o.update(status, lastContentLiked);
    }

    public void setStatus(String newStatus) {
        status = newStatus;
        // status has been updated! notify the subscribers
        notifyObservers();
    }

    public void setLastLiked(String newLike) {
        lastContentLiked = newLike;
        // last like has been updated! notify the subscribers
        notifyObservers();
    }
}
```

```
/* This is an Observer. It "subscribes" to FacebookUserSubjects */

public class Subscriber implements Observer {

    // Keep track of Subjects state
    String status;
    String lastContentLiked;
```

```
public void update(String status, String lastContentLiked) {
    this.status = status;
    this.lastContentLiked = lastContentLiked;

    printSubjectState();
}

public void printSubjectState() {
    System.out.println("Status: " + status + " |**| Last Liked: " + lastContentLiked + "\n");
}
}
```

```
public class Main {
    public static void main(String[] args) {
        FacebookUserSubject facebookUser = new FacebookUserSubject();

        Observer o1 = new Subscriber();
        Observer o2 = new Subscriber();
        Observer o3 = new Subscriber();

        facebookUser.setStatus("First status update!");
        facebookUser.setLastLiked("FistLike");

        facebookUser.registerObserver(o1);
        facebookUser.setStatus("Second status update!");
        facebookUser.setLastLiked("SecondLike");

        facebookUser.registerObserver(o2);
        facebookUser.registerObserver(o3);
        facebookUser.setStatus("Third status update!");
        facebookUser.setLastLiked("ThirdLike");

        facebookUser.unregisterObserver(o1);
        facebookUser.setStatus("Forth status update!");
        facebookUser.setLastLiked("ForthLike");

        facebookUser.unregisterObserver(o2);
        facebookUser.unregisterObserver(o3);
        facebookUser.setStatus("Fifth status update!");
        facebookUser.setLastLiked("FifthLike");
    }
}
```

The output of the application is as follows:

```
Status: Second status update! |**| Last Liked: FistLike

Status: Second status update! |**| Last Liked: SecondLike

Status: Third status update! |**| Last Liked: SecondLike

Status: Third status update! |**| Last Liked: SecondLike
```

```
Status: Third status update!|**| Last Liked: SecondLike
```

```
Status: Third status update!|**| Last Liked: ThirdLike
```

```
Status: Third status update!|**| Last Liked: ThirdLike
```

```
Status: Third status update!|**| Last Liked: ThirdLike
```

```
Status: Forth status update!|**| Last Liked: ThirdLike
```

```
Status: Forth status update!|**| Last Liked: ThirdLike
```

```
Status: Forth status update!|**| Last Liked: ForthLike
```

```
Status: Forth status update!|**| Last Liked: ForthLike
```

Decorator

The **Decorator** design pattern allows you to attach additional responsibilities to an object dynamically. It provides a flexible **alternative to subclassing** for extending functionality.

This is usually a great choice of a pattern whenever you want to be able to add responsibilities to individual objects dynamically (at **runtime**) without affecting other objects. The responsibilities you add can be withdrawn later on, also dynamically. Sometimes extension by subclassing is simply impractical: you might have a large amount of independent extensions and that would produce an explosion of subclasses and every time you added a new extension, you would have to create another subclass, which only aggravates the problem. The pattern gives you the capabilities of inheritance, but with functionality added at runtime.

Example

Let's say a car dealership is selling a certain model of a car. It's base price (for model with no extras) is \$100.000, but you can also add extras like **AirConditioning**(extra \$5.000), **Spoiler**(extra \$3.000) and a custom **BodyKit** (extra \$15.000).

One way to solve this would be to create a subclass for every possible combination, but that would give you 7 subclasses just for the cars ($3! + 1 = 7$)! And what if you wanted to add another extra? You can understand the way this is heading...

This seems like the right place to use the **Decorator** pattern. We want to add responsibilities dynamically.

First, we'll create the **Car** interface, which is the **common** interface for every car and every extra. Then, we'll create a **BasicCar** model, which represents the most **basic** car, which then will be

decorated by extras. All of the extras derive from the **CarDecorator**, which, just as the **BasicCar**, implements the **Car** interface. The decorator will store a reference to the object it decorates. In the decorator's superclass we want to implement the interface of the object that that group of decorators will be decorating.

Here is a possible way to solve this exercise:

```
public interface Car {  
  
    public String getDescription();  
    public int getPrice();  
}
```

```
public class BasicCar implements Car {  
  
    public String getDescription() { return "Basic Car Model"; }  
  
    public int getPrice() { return 100000; }  
}
```

```
public abstract class CarDecorator implements Car {  
  
    protected Car car; // reference to the car that is being decorated  
  
    public CarDecorator(Car car) { this.car = car; }  
}
```

```
public class BodyKit extends CarDecorator {  
  
    public BodyKit(Car car) { super(car); }  
  
    public String getDescription() {  
        return car.getDescription() + " + Body Kit"; }  
  
    public int getPrice() { return car.getPrice() + 15000; }  
}
```

```
public class AC extends CarDecorator {  
  
    public AC(Car car) { super(car); }  
  
    public String getDescription() {  
        return car.getDescription() + " + A/C"; }  
}
```

```
public int getPrice() { return car.getPrice() + 5000; }
}
```

```
public class Spoiler extends CarDecorator {

    public Spoiler(Car car) { super(car); }

    public String getDescription() {
        return car.getDescription() + " + Spoiler"; }

    public int getPrice() { return car.getPrice() + 3000; }
}
```

```
public class Main {

    public static void main(String[] args) {

        Car car = new BasicCar();
        System.out.println(car.getDescription() + " |**| Price : " + car.getPrice());

        // add a Body Kit to the car
        CarDecorator bk = new BodyKit(car);
        System.out.println(bk.getDescription() + " |**| Price : " + bk.getPrice());

        // add A/C to the car
        CarDecorator ac = new AC(bk);
        System.out.println(ac.getDescription() + " |**| Price : " + ac.getPrice());

        // add a Spoiler to the car
        CarDecorator sp = new Spoiler(ac);
        System.out.println(sp.getDescription() + " |**| Price : " + sp.getPrice());

    }
}
```

The application outputs:

```
Basic Car Model |**| Price : 100000
Basic Car Model + Body Kit |**| Price : 115000
Basic Car Model + Body Kit + A/C |**| Price : 120000
Basic Car Model + Body Kit + A/C + Spoiler |**| Price : 123000
```

Note, that on second and third “decorations” you are actually passing a **CarDecorator** and not a **BasicCar** object, so when you call **getDescription()** on one, it calls the **getDescription()** on another, until a **getDescription()** reaches a plain return of a string (here it happens in the **BasicCar** class).

This is why it's both, the decorator and the decorated (the car) must implement a **common** interface.

Adapter

The **Adapter** design pattern converts the interface of a class into another interface that the client expects. Adapter lets classes work together that couldn't otherwise because they have incompatible interfaces.

This is usually the right pattern to go with when you want to use an existing class without modifying it, but its interface doesn't match the one that you need.

Adapters in OO are just like adapters in real life. Have you ever needed to use an electronic device with an US AC plug in Europe? Well, since the European wall outlet (the client) expects a different "shape" of adapter what do you do? Change the wall outlet? Of course not! You use an **adapter** to adapt the US AC Plug into a Europe AC plug, as shown in the image below (image from www.safaribooksonline.com).



(This is kind of *off-topic*, but not all European wall outlets looks like that, in fact I believe I've never encountered one that's of the same shape as in the image, but they do exist.)

Example

Let's say that you have a duck simulator application. What does it do? Well, it simulates ducks by invoking **quack()** and **fly()** methods on various ducks.

Everything has been going good, until your users demanded that you include turkeys from another popular simulator "Turkey Simulator" straight into your game. The developers of "Turkey Simulator" are willing to share they turkey classes from they gigantic turkey database with you. That's great, except for one thing: all of the turkey classes have a different interface: instead of the **quack()** method, they have **gobble()** and despite the **fly()** method having the same name, they don't really fly the same way as the ducks do. You are certainly not going to change every class provided by the "Turkey Simulator" team, since that would take too much time and you would have to do that to every new turkey added, besides that, to ensure the quality of the turkeys used, they gave you the permission to **use** their code, but **not modify** it.

To solve our problem, we will create a new **TurkeyAdapter** class, which will store a reference to the turkey that it is adapting into a duck as well as implement the same interface as the **Duck** class is using.

Below is a possible solution using the pattern:

```
public interface Duck {  
    public void quack();  
    public void fly();  
}
```

```
public interface Turkey {  
    public void gobble();  
    public void fly();  
}
```

```
public class MallardDuck implements Duck {  
    public void quack() {  
        System.out.println("Quack!");  
    }  
  
    public void fly() {  
        System.out.println("Flying!");  
    }  
}
```

```
public class WildTurkey implements Turkey {  
    public void gobble() {  
        System.out.println("Gobble!");  
    }  
  
    public void fly() {  
        System.out.println("Flying a short distance.");  
    }  
}
```

```
public class TurkeyAdapter implements Duck {  
    Turkey turkey;  
  
    public TurkeyAdapter(Turkey turkey) { this.turkey = turkey; }  
  
    public void quack() {  
        turkey.gobble();  
    }  
}
```

```

public void fly() {
    for (int i = 0; i < 5; i++) {
        turkey.fly();
    }
}
}

```

```

public class Main {
    public static void main(String[] args) {
        Duck md = new MallardDuck();
        Turkey wt = new WildTurkey();

        System.out.println("Duck\n---");
        md.fly();
        md.fly();
        System.out.println("---\n");

        // adapt turkey to the duck's interface
        TurkeyAdapter turkeyAdapter = new TurkeyAdapter(wt);
        System.out.println("Turkey\n---");
        turkeyAdapter.fly();
        turkeyAdapter.quack();
        System.out.println("---\n");
    }
}

```

The application outputs the following:

```

Duck
---
Flying!
Flying!
---

Turkey
---
Flying a short distance.
Flying a short distance.
Flying a short distance.
Flying a short distance.
Flying a short distance.
Gobble!
---

```

There are two kind of adapters: **class adapter** and **object adapter**. This text only covered the **object adapters**. Class adapters aren't possible in Java, since they require **multiple inheritance**.

Object Adapter vs Class Adapter

The only difference between a class and an object adapter is that in the **class adapters**, the Target and the Adaptee are subclassed to create the Adapter. In **object adapters** object **composition** is used to pass the requests to an adaptee. Below is the **class adapter** UML.

Facade

The **Facade** pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

This pattern also allows you to avoid tight coupling between clients and subsystems, as well as provides a simple default view of the subsystem that is good enough for most clients. Only the clients that need more customizability will look beyond the pattern.

So in other words, the pattern allows you to take a complex subsystem and make it easier to use by implementing a Facade class, that provides one, more simple interface. It not only **simplifies an interface**, as well as **decouples a client from a subsystem of components**.

Example

As an example consider that you have a home theater system installed at your place. Suppose it consists of a DVD player, a projector, a screen, a sound system and lighs. Each one of the components is defined in it's own class:

```
public class DVDPlayer {  
  
    public void turnOn() {  
        System.out.println("Turning DVD On...");  
    }  
  
    public void turnOff() {  
        System.out.println("Turning DVD Off...");  
    }  
  
    public void play() {  
        System.out.println("Starting movie playback...");  
    }  
  
    public void stop() {  
        System.out.println("Stopping movie playback...");  
    }  
}
```

```
public class Projector {  
  
    public void turnOn() {  
        System.out.println("Turning Projector On...");  
    }  
  
    public void turnOff() {  
        System.out.println("Turning Projector On...");  
    }  
  
    public void start() {  
        System.out.println("Projecting started...");  
    }  
  
    public void stop() {  
        System.out.println("Projecting stopped...");  
    }  
}
```

```
public class Screen {  
  
    public void turnOn() {  
        System.out.println("Turning Screen On...");  
    }  
  
    public void turnOff() {  
        System.out.println("Turning Screen Off...");  
    }  
}
```

```
public class SoundSystem {  
    int volume;  
  
    public SoundSystem(int volume) {  
        this.volume = volume;  
    }  
  
    public SoundSystem() {  
        this(0);  
    }  
  
    public void turnOn() {  
        System.out.println("Turning Sound System On...");  
    }  
  
    public void turnOff() {  
        System.out.println("Turning Sound System Off...");  
    }  
  
    public void setVolume(int value) {  
        if (value > -1)  
            volume = value;  
    }  
}
```

```
        else
            volume = 0;

        System.out.println("Volume set to: " + volume);
    }
}
```

```
public class Lights {

    int intensity;

    public Lights(int intensity) {
        this.intensity = intensity;
    }

    public Lights() {
        this(0);
    }

    public void setIntensity(int value) {
        if (value > -1)
            intensity = value;
        else
            intensity = 0;

        System.out.println("Intensity set to: " + intensity);
    }

    public void dim(int value) {
        System.out.println("Dimming the lights...");
        setIntensity(intensity - value);
    }

}
```

Okay, now let's say you want to watch a movie. In order to do that, you have to perform a few tasks:

1. Turn the screen on
2. Turn the projector on
3. Turn the DVD player on
4. Set the sound to
5. Set the light's intensity to
6. Start the DVD playback

The code to do that without using the pattern would look something like this:


```
// Variables are properly initialized before
    screen.turnOn();
    projector.turnOn();
    dvdPlayer.turnOn();
    soundSystem.setVolume(5);
    lights.setIntensity(1);
    dvdPlayer.play();
```

Here are some issues with that approach:

- It's complex
- Let's say you want to reset everything after the movie is over, wouldn't you have to do everything again, but in reverse order?
- If you decide to change your system, and let's say put your phone on silence when the movie starts, as well as set the volume to 7 instead of 5, you're going to have to change your code in every place you've used it
- Wouldn't it be as complex to play a music CD or turn on the radio?

So what can we do here? The **complexity** of the current approach is evident. **Facade** pattern to the rescue! What we're going to do is create a Facade for our home theater system. To do this, we'll create a `HomeTheaterFacade` class which exposes a few methods such as `watchMovie()`, `stopMovie()`, `playCD()`, etc.

What are going to be the contents of those methods? Well, it's going to be all that complex code that was mentioned above. Here is a possible implementation:

```
public class HomeTheaterFacade {
    DVDPlayer dvdPlayer;
    Lights lights;
    Projector projector;
    Screen screen;
    SoundSystem soundSystem;

    public HomeTheaterFacade(DVDPlayer dvdPlayer,
                            Lights lights,
                            Projector projector,
                            Screen screen,
                            SoundSystem soundSystem) {

        this.dvdPlayer = dvdPlayer;
        this.lights = lights;
        this.projector = projector;
        this.screen = screen;
        this.soundSystem = soundSystem;
    }

    public void watchMovie() {
```

```

        System.out.println("Get ready for the movie...");
        screen.turnOn();
        projector.turnOn();
        dvdPlayer.turnOn();
        soundSystem.setVolume(5);
        lights.setIntensity(1);
        dvdPlayer.play();
    }

    public void endMovie() {
        System.out.println("Ending movie playback...");
        dvdPlayer.stop();
        lights.setIntensity(5);
        soundSystem.setVolume(0);
        dvdPlayer.turnOff();
        projector.turnOff();
        screen.turnOff();
    }

    // Other methods such as playCD(), ...
}

```

Now to watch a movie all we have to do is call the `watchMovie()` method on the `HomeTheaterFacade` object. The code is now a lot simpler and if we at some point decide to change the steps in any of the methods, the only place where we need to modify the code is in the `HomeTheaterFacade` class, since you **decoupled your client implementation from any one subsystem**.

Here is how the application that uses `HomeTheaterFacade` would look like:

```

public class Main {

    public static void main(String[] args) {
        DVDPlayer dvdPlayer = new DVDPlayer();
        Lights lights = new Lights(5);
        Screen screen = new Screen();
        Projector projector = new Projector();
        SoundSystem soundSystem = new SoundSystem();

        HomeTheaterFacade homeTheaterFacade = new HomeTheaterFacade(dvdPlayer,
                                                                    lights, projector, screen, soundSystem);

        homeTheaterFacade.watchMovie();
        homeTheaterFacade.endMovie();
    }
}

```

It produces the following output:

```
Get ready for the movie...
Turning Screen On...
Turning Projector On...
Turning DVD On...
Volume set to: 5
Intensity set to: 1
Starting movie playback...
Ending movie playback...
Stopping movie playback...
Intensity set to: 5
Volume set to: 0
Turning DVD Off...
Turning Projector On...
Turning Screen Off...
```

Facade vs Adapter

Facades and adapters may wrap multiple classes, but a **facade's intent is to simplify**, while an **adapter's intent is to convert one interface to another**.


Additional Resources

This post was heavily based on the Head First Design Patterns book. It provides an easy to read and understand introduction to the topic. I would suggest you reading it if you're new to design patterns. It's also a good book if you need to quickly refreshen some of the concepts.

Another great book, probably the most well-known one is Design patterns : elements of reusable object-oriented software. It's more complex, but it explores real world examples.

JAVA EXPLANATION

 LIKE  TWEET  +1

 FOLLOW @ILUXONCHIK

FOLLOW @ILUXONCHIK

 INSTAGRAM

Read More

Chave Móvel Digital's Log Out Button Does Not Log The User Out

The Log Out button in the Portuguese government's authentication system does not log the user out. This opens a window to unauthorized account access. [Continue reading](#)

Chave Móvel Digital Phone Number Leakage

Published on May 13, 2018

Chave Móvel Digital Multiple XSS Vulnerabilities

Published on May 13, 2018

© 2019 Illya Gerasymchuk. Powered by Jekyll using the HPSTR Theme.

comments powered by Disqus