Design Patterns are simpler than you thought

Henry Ng

Dedication

To Chi, Dau Xanh, and Tommy

Table of Content

Introduction

I have been loving to write a book to share my understanding on design pattern with others. Seven years ago, I tried to do that but I could only write about few patterns in my blogs. The writing at that time was not so good. It is very difficult to understand.

At some point of time, I am thinking about whether I can write about all 23 patterns without a break. Design patterns are so important in learning OOP. I think they are the best examples of OOP. They are the best means to learn OOP in depth. People believe that design patterns are advanced OOP and should not be introduced to OOP beginers. I think of the opposite. You should learn design patterns to understand OOP.

There are many books, blog posts, and discussion threads about design patterns. It is always required in job interview for software architect position. Recruiters believe that software architect should know design patterns very well. Surprisingly, design patterns are not widely taught at university. Books and pages about design patterns are too complex to understand. There are a couple of good posts about several patterns but not all of them.

Software architects do not need a book to learn design patterns. They already know them. They think that design patterns are trivial and easy to understand. Newbies think design patterns are too difficult to learn. So they don't learn them. This book tries to explain design patterns from a newbie's perspective. It is a minimal approach to design pattern which tries to strip off all unnecessary details and maintain the core principle of each pattern.

The Gang of Four are attributed to be the first who explicitly voiced the role of design patterns in OOP. They proposed 23 patterns. Following them, many other patterns have been introduced by different authors. This book sets an ambition to cover them all.

When rigorously examining all the GoF patterns, it is easy to figure out that many of them are similar. In fact, in a nutshell, we can confidently say that there are only 3 ways of using OOP.

- Creating objects without the keyword new
- Comprising an object by other objects as member properties
- Cross passing objects as param.

I will give examples at the end about all these ways of using OOP. Let's first start to look through all the patterns. Enjoy learning.

How you should read this book?

It is hard to have a book that targets all audiences. Advanced Java developers might find this too trivial and not worth their time. Beginners may think it is too difficult to finish one page. I believe some warming up exercises for those who have not worked with OOP for a long time

are useful. You need to get clear on how to use interface and abstract class as well as static methods and instances. Some patterns will require some sort of practices. Don't be shy if you have to copy line by line to finish a pattern. I will try to give some practice exercises so that you can try several times with patterns that you find challenging.

You can read through a book at one shot. However, it is also important to read it several times. It is very easy to forget what we have read.

Good luck.

Patterns to create an object

When I first heard about singleton and builder or factory, I was so perplexed. What the heck are they about? Patiently, I tried to google it a bit and jesus, I gave up after reading few lines.

How many ways are there to create an object in OOP?
Too difficult question it seems? Ok, make it simpler. Apart from using the keyword `new`, are there any other ways?

The fundamental principle of all creational design patterns is to create a new object without using the new() keyword.

Is that clear enough for you?
Not using new keyword. Are you kidding me?
Yes, I am serious.
But how?
Hahaha, it is so simple. Put the keyword new in a method and call that method to create the object you want.

This rule applies to all creational patterns including singleton, builder, factory method, abstract factory, and even prototype.

You see design patterns are so easy and that they (advanced programmers, software architects) always scare us. It seems they do that to maintain their own pious position: Hey it is for elite, you newbie should not touch.

Factory method

This is among the simplest design pattern. Let me show you how it embraces the *hiding keyword new principle* mentioned above.

Let's have a class Employee:

```
class Lecturer{
}
```

Now somewhere we want to use it

```
Lecturer e = new Lecturer()
```

No, it is not good because it using direct new keyword.

Let's have a method to hide it in a class namely Factory

```
class LecturerFactory{
  public static Lecturer createLecturer(){
        return new Lecturer();
  }
}
```

How simple it is.

That's about factory method. If you can understand it, that's enough. But if you want to go further. Let's complicate life a bit.

Now we want to create multiple object, It should not be only Lecturer but also other non-related stuff such as Order, Invoice, Bill, or Dog, Machine, Product.

We can extend our createLecturer() a little bit by generalizing it. We want to return different objects not only lecturer but also student, assistant etc. The pseudocode would be

```
public static Lecturer createLecturer(String objectYouWant){
   if(objectYouWant.equals("Lecturer"))
       return new Lecturer();
   else if(objectYouWant.equals("Student"))
       return new Student();
   else return null;
}
```

But this is impossible because we have to specify a generic type for object to be returned. We could do this by using multiple create methods such as createLecturer, createStudent etc. That's not very cool, right? We can use an abstract class Person to make a generic reference for all types of objects.

```
public class PersonFactory {

   public static Person create(String objectYouWant){
```

```
        if(objectYouWant.equals("Lecturer"))
            return new Lecturer();
        else if(objectYouWant.equals("Student"))
            return new Student();
        else return null;
    }
}

public class Person {
}

public class Lecturer extends Person {
}

public class Student extends Person {
}
```

Now let's use the factory to produce objects:

```
public class Main {

    public static void main(String[] args) {

        Person lecturer = PersonFactory.create("Lecturer");

        Person student = PersonFactory.create("Student");

    }
}
```

Wow, we can have objects without see or using the keyword new. How cool it is!

Up to this time, I hope you find design patterns are simple and natural like how we eat and breath. It comes to solve a common problem in a way that many people believe that it is better than what it has been solved.

But what if you still don't get the idea?

OOP Revisited:
Inheritance
I think you are struggling with abstraction, one of the four core principles of OOP. Abstraction in OOP rules that one class can extend other classes. By doing so, the subclass (child class) inherits all properties and methods provided by the base class (parent class).

```
public class Person {
    String name;
    int age;
```

```
    public void grow(){
        age++;
    }
}
```

Here is how the Person class is extended by Lecturer class:

```
public class Lecturer extends Person {

    int teachedHours;

    public void teach(){
        teachedHours += 2;
    }
}
```

Now if any lecturer object is created, it will have 3 properties (name, age, and teachedHours) and 2 methods (grow(), teach()).

Abstraction

Human sometimes is very ambitious creature. They always ask for more. Let's examine how they make life more complicated by introducing abstraction.

As you know we all have to work to earn a living. A lecturer (like I am) teaches to get money. A civil engineer constructs houses and building. Therefore, a person need to work but we don't know how specifically they work. We need to make them abstract and let the specific class define the work action.

Now we change the person class to abstract by putting the word abstract before the class word. It is English, right. An adjective comes before the noun that it modifies: abstract class.

```
public abstract class Person {

    String name;
    int age;

    public void grow(){
        age++;
    }

    public abstract void work();
}
```

Look the work() method has no body (What is a body? Body is code that written inside a pair of {}). It needs to be implemented by a subclass such as Lecturer class as follows:

```
public class Lecturer extends Person {
```

```
    int teachedHours;
    @Override
    public void work() {
        teachedHours += 2;
    }
}
```

Nicely done, huh?

Summary the pattern: Factory method design pattern seems to be most complicated one as it sounds but turns out to be the easiest one. The key of this pattern lies in the approach when people are fed up of using the keyword new to create an object. They hide it in a method and whenever they need an object they call that method.

When multiple types of objects need to be created, we define a create method with a String param to indicate what object we want to get.

Within its family, there is another variation that instead of using a method to create multiple objects of different types, we create different factory class for different object types. I want to talk about Abstract Factory. It is also super easy if you grasp the idea of hiding the keyword new when instantiating an object.

Abstract Factory
As mentioned above, in abstract factory pattern we will have multiple factory classes for different objects that need to be created. It is different from the factory method pattern where only one factory class is used to create objects of different types. I summarize the difference as follows:

Factory method: one class, one method
Abstract factory: many classes, each class contain one method

The advantage of abstract factory is that we can add more factory to create more object without touching existing codes.

Steps to implement this pattern:

Step 1: Define classes for objects created by the factory.

```
public abstract class Person {
}

public class Lecturer extends Person {
}
```

Step 2: Create an interface namely AbstracFactory

```
public interface AbstractFactory {
   public Person create();
}
```

Step 3: Implement the interface with a concrete class

```
public class LecturerFactory implements AbstractFactory {
   @Override
   public Person create() {
       return new Lecturer();
   }
}
```

Are you with me now? I hope the less than 10 lines of codes written above won't give you any trouble in understanding. At first, it seems tricky when there are many classes to be defined. However, it is too short and simple to even learn by heart. If you are very new to OOP or Java, this will take you some time to fully be fluent with this paradigm of coding: abstract class → concrete class, interface → implement interface.

For newbies, I recommend that you practice this example several times. Try to change the class name to create different object such as Dog, Cat, Product, Car, Phone, House, Window. If you redo this example 5 or 10 times, it will take only 20-30 minutes. Abstract class and interface will be used again and again in design patterns. Without fluency in abstraction, it is hard to comfortably read other design patterns.

The Builder design pattern

We talked about how we can create an object in OOP without using the keyword new. It sounds insane but I have shown you that it is possible in factory pattern. We achieve this by just hide using the keyword new in a method and we call this method to have a new object instead of directly issuing the new command.

The Builder design pattern works in almost the same way. The first subtle difference is we use method build() instead of create(). Haha, how clever the GoF is. Is it just a trick or play on word?

Well you can say so. Take a look at the following example:

```
public class Car {
}

public class CarBuilder {
   public Car build(){
       return new Car();
```

```
    }
}
```

There you go. Superbly clever. In Builder pattern, we use the method build() to create an object rather than create() which is used in Factory pattern. Hold on. There are a little bit more. But if you can understand this example, I guarantee that Builder pattern is not something that can give you a hard time.

When making a car or a house is not a one-day work, right? It is often a long process that involves many steps. For example, building a car will involves painting its body and assemble its engine. A car is only ready for sell when it is painted and has an engine. Let's follow that process to build a car.

```
public class Car {

    String color;
    String engine;

    }



public class CarBuilder {

    Car car = new Car();

    public CarBuilder paint(String color){
        car.color = color;
        return this;
    }

    public CarBuilder assembleEngine(String engine){
        car.engine = engine;
        return this;
    }

    public Car build(){
        return car;
    }

}
```

Two things you should pay attention to in this builder class.

- CarBuilder has a property which is a car object. Yes, we hide the creation of car in here.
- Both the paint and assembleEngine methods return the CarBuilder object itself. That's why we use the keyword this. The usage of this and method that return the object itself is

a very common technique in OOP. This will enable the chain of command which allows us to call multiple method in the same line. We can see it in the main method as follows:

```
public class Main {
   public static void main(String[] args) {
       CarBuilder carBuilder = new CarBuilder();
       Car car = carBuilder.paint("red").assembleEngine("BMW").build();
   }
}
```

You now can see that we can call the three methods paint, assembleEngine, and build in the same line.

This pattern sounds very simple but I am sure when you try to repeat it yourself you will face some challenges. To newbies, a method that return this is something completely weird. You might never touch it if you don't read this book on design pattern. Now I hope you appreciate what I said in the beginning of the book: design patterns is not a goal but it is a excellent mean to learn OOP. In other words, we could learn OOP through design patterns. From my experience, it is an extremely useful way to learn "the trick of the trade" of using OOP.

In summary, to implement Builder pattern, we need:

- Define a builder class that must contain an object that represents what you want to build.
- In the builder class, define all relevant methods to set up properties and states of an object. To make it handy, use return this in all these methods.
- The last method of the builder class should return the target object that you want to build.

OOP revisited

Member objects:

In OOP you can use keyword new to create a member object when you define a class. For example:

```
public class CarBuilder {
   Car car = new Car();
}
```

When a CarBuilder object is created, a car is also instantiated. The car object will not null. You can also use the keyword new to create a car inside a constructor which is equivalent to this.

Using keyword "this":

Some may be confused by the keyword this. It is in fact very easy to understand. When you use an object outside the class, we always have a object variable to refer to that object. But if we want to refer to an instance of a class within the class declaration, we must use the keyword this.

What is “this”? It is a representative of an object. Therefore, we can use this.property or this.method() to access its members.

There you have it. They are two new ways to have an object without using the keyword new. We have two more left for you: singleton and prototype. They are two opposite pattern. Singleton aims to create a unique instance while prototype is to create multiple same object. Stay tune.

Singleton

This is very famous pattern. Most of us have heard about it at least once. You are not an exception either. Unlike builder and factory, we can understand the meaning and the goal of singleton at the first glance. Yes, it is about a single and unique object. One object per class only. Many things in our life are singleton. One sun, one moon, one Antarctica. We don’t need to create many suns, do we?

Singleton pattern sounds easy but implementing it is a bit tricky. We need to mix a static property with a non-static properties and methods. It sometimes makes us very puzzled. Let’s start.

To make a singleton, we must think of a way to block all possibility of creating a new object. In other words, we must disable the use of new.

We can do this easily by making a constructor private. This idea is very innovative, isn’t it? Let’s just do it:

```
public class SunSingleton {
   private SunSingleton() {
   }
}
```

We make the constructor private because we don’t want user to create many objects. But wait, we now could not even make a new instance of SunSingleton. Poor us!

Don’t give up. Remember the modifier private only blocks the access from outside of the class, we can still use constructor inside this class. Let’s use it to create an instance of SunSingleton.

```
public class SunSingleton {

   static SunSingleton instance = new SunSingleton();

   private SunSingleton() {
```

```
    }
}
```

Look we have to use the keyword static because that is the only way we can access the variable instance from outside of this class.

```
public class Main {
    public static void main(String[] args) {
        SunSingleton sun = SunSingleton.instance;
        //now we can do anything with sun object such as access its methods or properties
    }
}
```

With such few lines of code, we have successfully implemented a singleton. If you are a perfectionist, you must ask for more. How about this:

```
public class SunSingleton {

    private static SunSingleton instance = new SunSingleton();

    private SunSingleton() {
    }

    public static SunSingleton getInstance(){
        return instance;
    }
}
```

We make the instance variable private and define a method getInstance() to return this instance. In OOP, they call it encapsulation and recommend us to follow this rule of design. All properties must be private.

Singleton vs static methods

When I just graduated uni, I could not see the reason why we need to use singleton. OOP offers static method and static variable which is very much equivalent to OOP. Later on in my career life, I slowly understand that singleton has what static approach offers but singleton is still a normal class. It can have inheritance and abstraction that static methods and properties could not have. OK. sound very convincing. Let's stick to that.

Prototype

Well, congratulation. You have been encountering 3 most important design patterns. OOP objects need to be instantiated before we can use them. No doubt that they are more significant than others. I am refering to 20 other patterns discussed by the Gang of Four. The last pattern in the creational group is prototype.

What is prototype?

In English, prototype refers to a sample or model based on which we can replicate many other instances. In OOP, prototype pattern is about how to clone (duplicate) an object to many other objects.

It sounds very stupid when we need a pattern just for copying objects. However, if you remember that objects in OOP is mainly references. Assigning one object to another could not copy the object but only point the object variable to the existing one.

Let's look at the following example:

```
public class Product {
   String title;
}

public class Main {
   public static void main(String[] args) {
      Book book = new Book();
      book.title = "Go with wind";

      Book book2 = book;

      book.title = "50 shades";
      System.out.println(book2.title);
      //Do you know what is printed?
      //It is 50 shades
   }
}
```

Because objects in OOP are just references, backing up an object by assigning it to another object is not possible. We have to clone object instead of assigning it. Java offers Cloneable interface. However, it only permits shallow copy. What is shallow copy by the way? Shallow copy can only copy properties of primitive data type such as int, float, String. It can't copy properties of object type.

But first let's try to clone object using Cloneable interface.

```
public class Book implements Cloneable {
   String title;

   public Book clone() throws CloneNotSupportedException {
       return (Book)super.clone();
   }
}
```

Now let's use it in main:

```
public class Main {
   public static void main(String[] args) throws CloneNotSupportedException {
      Book book = new Book();
      book.title = "Go with wind";

      Book book2 = book.clone();
      book.title = "50 shades";

      System.out.println(book2.title);
       //Do you know what is printed?
      //It is Go with wind
   }
}
```

The clone method is offered by Object class. You know that in Java Object class is a base class of all other classes. When we call super.clone(), Java will go through each properties of the object and duplicate them and assign their values to those of the new object. We of course can build our own clone method without depending on Java.

How about this?

```
public class Book {
   String title;

   public Book clone()  {
       Book book = new Book();
       book.title = this.title;
       return book;
   }
}
```

This class also follows the core principle that I explained previously: hiding the use of keyword new in creating object. In this case, the keyword new is placed inside the clone method. When we need a new object we just need to call clone instead of using new.

Defining our own clone method yields the same result as using Java Cloneable interface. However, the problem with our manual cloning is that we have to build the clone method for all the classes that you want to clone. We can generalize this a bit by defining an abstract class that implements Cloneable interface. After that, if we want any class to have to be clonable, we just need to extend this abstract abstract.

Let's have a prototype abstract class as follows:

```
public abstract class Prototype implements Cloneable {

   public Prototype clone() throws CloneNotSupportedException {
```

```
        return (Prototype) super.clone();
    }
}
```

Cloneable interface has no method for us to implement. It is just to notify the compiler that this class has ability of cloning offered by the Object class. It sounds strange, right? super.clone() is actually a method of Object class, not the Cloneable interface.

Any class that you want to clone just need to extend this Prototype

```
public class Book extends Prototype {
    String title;
}
```

Now use it:

```
public class Main {
    public static void main(String[] args) throws CloneNotSupportedException {
        Book book = new Book();
        book.title = "Design patterns are easy";

        Book book1 = (Book) book.clone();
        System.out.println(book1.title);
        //It printed Design patterns are easy to the console
    }
}
```

So easy, isn't it?

So far, we have discussed all design patterns used to create objects (creational patterns). Again, the key to these patterns is how to hide the use of keyword new from the place we want to instantiate an object. If you could grasp this principle, I don't think they pose any problem to you. I would like to reiterate that creational patterns are among the most important patterns compared to others. You may not need to know all the design patterns but you should at least know these 5 patterns.

In the next chapter, we will go through design patterns from easy to difficult. I will start with patterns that I feel easy first and gradually move to more complex patterns.

Objects that leverage objects to perform its own work

The overarching classification of design patterns is a bit problematic to me. At least, it is too broad and it does not say anything about the internal structure of objects. After examining all the patterns I figure out that many of them just use their object members to perform some tasks. I

know this is a fundamental principle in programming but I want to emphasize that in some patterns, an aggregate class has some methods in which it invokes methods of its member objects.

What is an aggregate class by the way? Aggregation class is a class that has a member variable whose types is another class. Aggregation is one specific case of association which represents the whole/part and has-a relationship. We can have many examples of that in real life. For example, a company has many employees, a computer has different components such as ram, cpu, hard drive etc. We can shortly see how this basic principle of OOP is applied to Facade design pattern.

Facade design pattern

Let's examine the pattern Facade by a simple example. A supermarket sells many things but they don't actually make anything. They get them from suppliers and resell them. Some supermarket try to sell their own brands but they outsource to some suppliers. I see this example perfectly fit to the idea of facade.

In English, facade refers to the outer face of a building. It may or may not truly reflect what are actually inside. In our example, supermarket can supply to customers foods and drinks but they rely on suppliers to do that. What they provide are just a facade. Suppliers are really the ones who supply the goods.

```
public class Supermaket {

   public void supplyFood(){

   }

   public void supplyDrink(){

   }
}
```

Is it how a supermarket works? No it is not true. They ask suppliers to do that for them. In fact, the class should be refactored to become:

```
public class Supermaket {

   FoodSupplier foodSupplier;
   DrinkSupplier drinkSupplier;

   public void supplyFood(){
       foodSupplier.supplyFood();
   }
```

```
    public void supplyDrink(){
        drinkSupplier.supplyDrink();
    }
}
```

Of course, we can use different method names in FoodSupplier class and Supermarket class. However, I try to use the same name to emphasize that the Supermarket does nothing, they copy exactly the method name to show that they have such ability but in fact they just borrow it.

We don't have to care about what actually happen inside supplyFood and supplyDrink methods. This is all about facade pattern. I think it is most straight-forward and simple pattern compared to others.

I hope you get it now, don't you?

Template method pattern

I would like to discuss the next easy design pattern which is very related to the Facade pattern. As you already know OOP offer abstract class with which a class can have a number of unimplemented methods. One advantage of abstract class is that it allows subclasses implement their own behaviors. A classical example of this is animal. All animals can make make a sound. However, it is impossible to specify the sound for the animal class. We have to defer it to concrete class where cat goes meow, dog goes woof woof. To some extents, the abstract class is a form of template pattern because abstract class defines placeholders so that subclasses can follow and fill in.

Let's look at the following abstract class:

```
public abstract class Animal {
    public abstract void makeSound();
}
```

Now we implement the makeSound method in both Dog and Cat class:

```
public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("meow meow");
    }
}

public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("woof woof");
```

```
    }
}
```

This example can be considered as a trivial form of template method. A template makeSound() has been defined and all other subclasses must implement it. Template forces you to follow a set of rules and give you a limit set of choices.

But let's take it to a higher level. Let's say an animal must make a sound to send early warning signs to enemies before he attacks. We must ensure that the makeSound() method is invoked before the attack() method. The only way to achieve it is to use Template method pattern.

First, we must identify and define action that embraces all other actions and make it final. We do not allow any override this method. Surely, we have to do that because this method is considered as a template. Did you try to change the template form that your boss requests you to fill?

There we go, a template method in Animal class:

```
public abstract class Animal {
    public abstract void makeSound();
    public abstract void attack();

// The fight() is a template method.
    public final void fight(){
        makeSound();
        attack();
    }
}
```

The fight() is a template method. It is not supposed to be override so we use the keyword final. It is not supposed to be implemented by subclasses so it is NOT abstract.

The two methods makeSound and attack must be abstract because every animal has their own way to make sound and attack.

If you can get it up to now, I think the rest is very easy to you. You can just update the two subclasses Cat and Dog by implementing the attack() method.

If you are still uncertain, let me help you:

```
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("woof woof");
```

```
    }

    @Override
    public void attack() {
        System.out.println("bite");
    }
}

public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("meow meow");
    }

    @Override
    public void attack() {
        System.out.println("scratch");
    }
}
```

I hope you are not perplexed by these too many lines of codes. I always bear in mind that all the examples should be in their simplest forms so that readers can grasp the core idea quickly. I know that explaining a technical topic through a book (even though with the support of lots of examples) is always difficult to the readers to fully grab it.

In summary, the following steps are required to make a Template method pattern:

a) Define an abstract class with a number of abstract methods
b) Define a non-abstract method and make it final (non-overridable). It will act as a template method.
c) Invoke all the abstract methods inside this non-abstract method.

Congratulation, you have a template method pattern.

If you have some years programming with OOP, I guess you have tried this pattern several times even though you are not aware of.

Adapter

In design patterns, there are some patterns that sound very easy to understand. Adapter is among them. Other methods such as Memento, Visitor etc. are extremely hard to guess what they are about.

At the first glance, adapter gives us an idea about changing from one thing to another. It is about compatability. If you are a frequent traveller, electric adapter is what accompany you, right?

While the example of electric adapter seems to be powerful in explaining the adapter pattern, I had a hard time figuring out what is the adapter and what need to be transformed. I think the example of an interpreter is better. In an international conference, when everyone can speak English, a non-English participant must need a translator to help her to communicate with others. We can easily ask the participant to learn English but it is impossible in a matter of few days. That's legacy issue or installed base as my colleagues in University of Oslo often refers when we discuss the theory of information infrastructure.

The example is very interesting and very fit to the context of the adapter pattern. Still, there is a mismatch when we try to convert it to an OPP example. Anyway, I will just go ahead. You might find it match better than me, maybe.

Let's say we have a standardized library (large part of system depends on it to function) that contains an interface as follows:

```
public interface Person {
   public String getName();
}
```

There are a couple of implementation of this interface. For example, the VietnamesePerson class:

```
public class Vietnamese implements Person {
   @Override
   public String getName() {
       return "Last name first";
   }
}
```

In our legacy code, we also have a French class that has a method getNom() which is very similar to getName method that we have in Person interface.

```
public class French {
   public String getNom(){
       return "Prénom d'abord";
   }
}
```

It is perfect if we can treat French class as the same way as Vietnamese, i.e. call the getName() method. If it is possible, we can put both Vietnamese and French into an array and iterate through it. The French class is legacy, you do not have the source code or we don't want to change this class as it can affect other parts of the system.

Actually, the problem can be solved easily by using the adapter pattern. What is adapter pattern? We haven't discussed it, right.

OK. Here it is. We can solve this problem in a very simple way by wrapping the French class inside a class that implement the Person interface. For that, French becomes a member object of that wrapper[1] class. We invoke the getNom method of French class inside the standardized getName() method of our Person interface.

```
public class FrenchAdapter implements Person {
   French french = new French();

   @Override
   public String getName() {
       return french.getNom();
   }
}
```

How simple and brilliant it is! And watch, is it very similar to the facade pattern. We call method of a member variable (getNom of French) inside the method of the aggregate class (getName of FrenchAdapter).

Let's use the adapter:

```
public class Main {
   public static void main(String[] args){
       Person[] people = {new Vietnamese(), new FrenchAdapter()};
       for (Person person: people) {
           System.out.println(person.getName());
       }
   }
}
```

In summary, it is quite easy to have an adapter.

a) You define a class that extend that interface that you want to make the old class to be compatible with.

[1] For that reason, adapter pattern is also called wrapper, translator pattern.

b) In this class, you declare a variable with the type as the old class.
c) When implementing methods from the interface, call methods of this member variable.

I hope you find this pattern easy to understand. The most important thing that you need to remember is that the adapter class must wrap the old class, i.e. has a member variable (field) that refers to the old class. You should also realize that using an object as member variable of another object is very common in OOP and this rule applies to most of the design patterns.

Now we move to the two most complicated and difficult to grasp of design patterns: Observer and Visitor. While the name Observer seems to be intuitively meaningful, the Visitor would be hard for someone who tries to understand what it is through its name.

Observer

As I said, observer pattern seems to be very easy for us to guess what it is and what it actually does. Observe in English means look, watch, and early detect if there is something new, unusual, and dangerous. We have observatory towers which are used to detect if there is any threats of enemy attacks. In beaches, there are observatory towers to find and save drown victims. Observe thus goes with watching for a change and subsequent actions based on that change. This notion applies exactly to observer design pattern. There are thus 3 instances here: observer, observee, and action (when observer a change is detected).

In another scenario, if we have many observers who observe on one subject such as many customers watch for discount of airline ticket, it will be very costly if all of them have to check for the change in the ticket price every 5 minutes. Long time ago, when I read a book on Spring there is one very famous quote that has been used again and again to describe the principle of dependency injection: ***Don't call us. We will call you***. Rod Johnson, the author of the book, said this is a very popular style which is commonly used in Hollywood.
I have googled it just now and found out very interesting explanation from Matthew Mead.

"I have always admired the cleverness of the name Hollywood Principle. I love its name! The essence of this principle is "don't call us, we'll call you". As you know, this is a response you might hear after auditioning for a role in a Hollywood movie. And this is the same concept for

the software Hollywood Principle too. The intent is to take care to structure and implement your dependencies wisely."

(http://matthewtmead.com/blog/hollywood-principle-dont-call-us-well-call-you-4/)

OK, let's go back to our design pattern. Following the Hollywood principle, the subject (the observee) should inform all the observers that subscribe to it about the change, not the other way around.

We have a new keyword here: subscribe. You subscribe to a Youtube channel, it will notify you whenever there is a new video published on the channel. You subscribe to a newspaper, they send to your house every time there is a new issue. In OOP, the subject thus must maintain a list that contains all the observers who want to observe them. Whenever there is a change in the state of subject, i.e. value of property change, the subject must call a method that go through a the observer and notify them. It sounds very complicated but it is much simpler if we examine the following example:

Let's have a Subject class that has a String property which is state. The setState is the method that change the state of a subject.

```
public class Subject {

   String state;

   public void setState(String state) {
       this.state = state;
   }
}
```

We want that whenever the method setState is invoked, all the objects that subscribe to this subject will be notified. Let's define our Observer class:

```
public class Observer {

   public void beNotified(Subject subject){
       System.out.println("The state has been changed to "+ subject.getState());
   }
}
```

The Observer class has a method beNotified that will do something based on the state value change. In some examples found on Internet, most people use this method without a param. Instead, the observer class has a property whose the type is Subject. I don't think both ways are fine.

As I have said, the Subject class needs to maintain a list of observers. This is similar to Youtube channel subscription where the channel must have a list of subscribers.

```
public class Subject {

   List<Observer> observers = new ArrayList<>();

   String state;

   public String getState() {
       return state;
   }

   public void setState(String state) {
       this.state = state;
   }
}
```

Now let the observer subscribes the subject, we need a subscribe method for the Observer class:

```
public class Observer {

   public void beNotified(Subject subject){
       System.out.println("The state has been changed to "+ subject.getState());
   }

   public void subscribe(Subject subject){
       subject.observers.add(this);
   }
}
```

The subscribing process happens exactly the same way in real life. We write the name of the subscribers to the list. Here is how we did in Observer subscription:

```
  public void subscribe(Subject subject){
       subject.observers.add(this);
   }
```

Tricky, simple and tempting enough, right? Now let the Subject notify their subscribers when there is a change in its state. We write a method namely notifyObservers that iterates through the observer list and call the Observer's beNotified() method. One last thing, we call notifyObservers inside the setState method to indicate that every time the state changes, all subscribers must be notified.

```
public class Subject {
```

```
    List<Observer> observers = new ArrayList<>();

    String state;

    public String getState() {
        return state;
    }

    public void setState(String state) {
        this.state = state;
        notifyObserver();
    }

    public void notifyObservers(){
        for (Observer observer: observers){
            observer.beNotified(this);
        }
    }
}
```

Now let's use the pattern in Main class:

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

        Observer observer1 = new Observer();
        observer1.subscribe(subject);

        Observer observer2 = new Observer();
        observer2.subscribe(subject);

        subject.setState("new state");
    }
}
```

Congratulation. You have completed the Observer pattern, one of the hardest pattern among 23 patterns. In the past, when I encountered this pattern, I thought there must be some magic happening behind the scene. There is no magic here. The subject must go through a list of observer objects and inform each of them about the change. Observers won't go to Subject to change for change but Subject will notify when there is a change. That's Hollywood principle: We'll call you.

We can revise our pattern a little bit by making Observer as an abstract class. By doing that, we can have multiple types of objects with different behaviors when Subject changes its state.

If you find this pattern challenging as well as interesting and you want to continue with this genre, I am happy to offer the next complex pattern: the Visitor.

Visitor pattern

I was struggling to find a good example to demonstrate the use of this pattern. The best way to explain a pattern is to argue that why we need that pattern. Sure, I will do it. But first, what kind of examples you want. A real life example or a OOP related example. To make this book more human, I will take an example of the earlier. There is a father who loves her daughter very much. His daughter graduated university for some time but could not find a job. Her fiancee family said they only allow a wedding if she is employed. The father decided to insist a local school to hire his daughter. He would use his own money to pay her salary. And for that, she got employed and got married. The father who actually paid salary but for some reasons he had to ask others to pay on his behalf secretely.

Read this story so far, you might ask how it is relevant to the Visitor pattern. In fact, it is. The visitor pattern is about some actions you can do but you can't then you have to ask someone (visitor) to do it for you.

Go back to OOP, we know that so often in subclasses, an object has some additional methods but they could not be invoked because the reference is to the parent class. We can cast it but it is considered as a not good approach.

```
public abstract class Person {
}

public class Lecturer extends Person {
   public void teach(){
       System.out.println("Lecturer teach");
   }
}

public static void main(String[] args){
  Person lecturer = new Lecturer();
  //this is NOT possible unless we cast
  lecturer.teach();
}
```

If we want to call teach() we have to cast:

```
public static void main(String[] args){
  Person lecturer = new Lecturer();
  //this is possible
   ((Lecturer)lecturer).teach();
}
```

Let clearly define our problem. Our problem now is whether we can call teach() without casting. Yes, it is possible. We use the Visitor pattern to do that :-).

Now we define a Visitor class with a method visit(). In this visit method, we call whatever lecturer object cannot call.

```
public class Visitor {
   public void visit(Lecturer lecturer){
       lecturer.teach();
   }
}
```

Ok, if a visitor visits our lecturer, she can help the lecturer call the method teach() because the param of visit is the concrete type, not an abstract reference. But a visitor can not come without invitation, the lecturer must invite him. So we need a method invite and a lecturer must be able to call this method. To do that, the invite method must be declared in the abstract class Person.

```
public abstract class Person {
   public abstract void invite(Visitor visitor);
}
```

Now we update our Lecturer class by implementing this abstract invite() method:

```
public class Lecturer extends Person {
   public void teach(){
       System.out.println("Lecturer teach");
   }

   @Override
   public void invite(Visitor visitor) {
       visitor.visit(this);
   }
}
```

Well done, huh? Now I show you how to use this pattern in the main method and how it perfectly addresses the problem without using casting.

```
public class Main {
   public static void main(String[] args){
      Person person = new Lecturer();
       //the following statement is equivalent to lecturer.teach();
      person.invite(new Visitor());
   }
}
```

You might see variations of this pattern through a number of examples on Internet. They are a bit more complex. In this example I remove many unnecessary details to make it clearer and easier to understand. Many people use the accept method instead of invite. I see invite is much more relevant than accept.

This pattern is very flexible. In case we want to invite different visitor for other activities such as marking, building lecture slides, and meeting we just need to define more visitor classes and call invite corresponding visitors. For example, we can have TeachingVisitor, MarkingVisitor, ContentBuildingVisitor etc. But if we want to reuse the accept method for different visitors we need to create an abstract class or interface namely Visitor. I use abstract class but you should use interface as interface allows higher flexibility, i.e. one class can implement multiple interfaces.

```
public abstract class Visitor {
   public abstract void visit(Lecturer lecturer);
}


public class TeachingVisitor extends Visitor {
   @Override
   public void visit(Lecturer lecturer) {
       lecturer.teach();
   }
}

public class MarkingVisitor extends Visitor {
   @Override
   public void visit(Lecturer lecturer) {
       lecturer.mark();
   }
}
```

Now let's use this pattern:

```
public static void main(String[] args){
  Person person = new Lecturer();
   //the following statement is equivalent to lecturer.teach();
  person.invite(new TeachingVisitor());
   //the following statement is equivalent to lecturer.mark();
  person.invite(new MarkingVisitor());
}
```

The flexibility of this pattern is also in its ability to define multiple concrete type of Person. So far, we only use Lecturer. What about Student class with methods such as

like study, attemptExam etc. or SupportStaff class methods such as receiveRequest, resolveTicket etc.

Congratulation. You have learned one of the hardest design patterns. The rest will be only easier and easier ones. I hope you don't lose your interest in reading this book.

We now start with a design pattern with medium level of difficulty: Decorator

Decorator

At the first glance, this pattern is very similar to a wrapper (You might hear this term: a wrapper in Adapter pattern). You wrap and decorate a gift before giving it to someone. We don't change the gift inside the gift box. We make it look more attractive. We add more value to it. How can we improve a value of something without changing it content? That's the job of decorator pattern.

I will demonstrate this pattern by one example that is not related to gift and present. Well, I still couldn't find a way to make a gift as a good example when writing code for this pattern. Therefore, I will use an example in civil engineering: building, painting, and eventually DECORATING a house.

First, we have a House class with a method decorate():

```
public class House {

   public void decorate(){
       System.out.println("Basic decoration");
   }
}
```

Here is how we use it:

```
public static void main(String[] args) {
   House house = new House();
   house.decorate();
}
```

Now we want to do more advanced decoration: lights, arrangement etc. For some reasons, we can't change our decorate method. And we either can't wrap house with

other object to add more functionality. The decoration pattern can help to solve that problem perfectly.

But first, we must have a Decorator interface and you have to define a House class that implement this interface:

```
public interface Decorator {
   public void decorate();
}

public class House implements Decorator {

   public void decorate(){
       System.out.println("Basic decoration");
   }
}
```

Now everything becomes much easier. If you want to do light decoration, you just need to add a class LightDecorator:

```
public class LightDecorator implements Decorator {

   Decorator decorator;

   public LightDecorator(Decorator decorator) {
       this.decorator = decorator;
   }

   @Override
   public void decorate() {
       decorator.decorate();
       System.out.println("Light decorating");
   }
}
```

The most tricky part of this pattern is ***we add a member variable of the interface*** that we implement. Why we need that? Simply because we want to reuse the basic decoration before we can add more advanced ones.

Now we can use it easily:

```
public static void main(String[] args) {
   Decorator house = new LightDecorator(new House());
   house.decorate();
}
```

Wait what is the advantage of doing this? We can have other decorator without having to change our existing code, we just need to add one more:

```
public class FurnitureDecorator implements Decorator {

   Decorator decorator;

   public FurnitureDecorator(Decorator decorator) {
       this.decorator = decorator;
   }

   @Override
   public void decorate() {
       decorator.decorate();
       System.out.println("Furniture decorating");
   }
}
```

And use it with a chain of object creation:

```
public static void main(String[] args) {
   Decorator house = new LightDecorator(new FurnitureDecorator(new House()));
   house.decorate();
}
```

This chain of object creation make it flexible to decide what kind of decorations we want to apply for our house.

I think you are tired of learning too many new things and I will continue the book with some very easy design patterns: Memento and Flyweight.

Flyweight

This design pattern has very strange name. In combat sports, flyweight refers a lightest weight class. In boxing, flyweight is the fight between boxers whose weight are very light between 49 and 52, light as a fly. I don't see any connection between this flyweight with the flyweight in design pattern.

To me flyweight is very much about caching, a common technique in computer science that leverage the reuse of scarce resources in this case is storage and memory (RAM).

The purpose of using flyweight pattern is to prevent creation of unnecessary objects. Similar objects should be reused rather than being created new.

Let's start by examining the following example. In a national exam, every student must register to one university. There are millions students but only a few hundred universities. If when creating every student we create a new university object, it is very wasteful. Java inventor was among the first realized this problem and in the implementation of String, Java use intern approach to reuse all the strings that have the same values. If you don't believe you can try the following example:

```
String s = "Design pattern";
String s1 = "Design pattern";

if(s==s1) System.out.println("s and s1 points to the same object");
```

This approach can save lots of memory used for storing strings. Go back to our example of university exam. Let examine all the involved class:

```
public class Student {
   String name;
   University university;

   public Student(String name, University university) {
       this.name = name;
       this.university = university;
   }
}


public class University {
   String id;
   String name;

   public University(String id, String name) {
       this.id = id;
       this.name = name;
   }
}
```

Now this is how we often do:

```
public class Main {
   public static void main(String[] args) {
       University rmit = new University("rmit", "RMIT Vietnam");
       Student moor = new Student("David Moor", rmit);
```

```
        //In other places, we keep creating more of the same object
        University rmituni = new University("rmit", "RMIT Vietnam");
        Student ronaldo = new Student("Christian Ronaldo", rmit);
    }
}
```

Everyone knows that this is a waste of resource. Java should have been able to detect objects with the same properties and automatically cache them for us, like what it does with String.

Many people criticize design patterns. They argue that design patterns just proves the flaws in the design of a programming language. They might be true. But that's another story. Now we try to find how to resolve this problem.

The approach is we do not allow users to create a new university object through the keyword new. We have discussed this issue many time and the whole bunch of creational design pattern group is used to achieve that goal. We can use Factory, a factory is a special class that is used to create many objects of another class. Let's try to be tricky a bit by converting the University class itself to a factory class. We will make the constructor method private and define a static method to create university object. By doing so, we can prevent users creating excessive university object. To store already created objects, we use a Java collection namely Map inside the University class.

So clever are we?

```
public class University {
    String id;
    String name;

    static Map<String, University> pool = new HashMap<>();

    private University() {
    }

    public static University create(String universityId, String name){

        University university = pool.get(universityId);
        if(university==null) {
            university = new University();
            university.id = universityId;
            university.name = name;
            pool.put(universityId, university);
        }
```

```
        return university;
    }
}
```

This approach is very similar to the Singleton. The only difference is instead of creating one object we create a pool of shared and reusable objects. Here how we use it:

```
public static void main(String[] args) {
    University rmit = University.create("rmit", "RMIT Vietnam");
    Student moor = new Student("David Moor", rmit);

    //In other places, we keep creating more of the same object
    University rmituni = University.create("rmit", "RMIT Vietnam");
    Student ronaldo = new Student("Christian Ronaldo", rmituni);

    //Both rmit and rmitunit points to the same object
    System.out.println(rmit==rmituni);
}
```

In summary, the core of flyweight pattern is the use of a pool of shared objects. To achieve that goal we need to enforce that the creation of all objects must be done by a create method instead of using keyword new. In our example, we use factory to do that.

I hope you find this pattern easy and natural as it goes. Let's move to another easy pattern which to someone has a very strange name: Memento.

Memento

Memento in English means a souvenir, a gift or a thing that remind us about something. In OOP, many situations, we want our object roll back to its original state, i.e. restoring properties that have been changed. For example, in a bank transaction, if there is a problem occurs such as not-enough money, the balance of the bank account should not be deducted. Having a way to restore the state of an object is thus important.

We can do this by using object cloning. With this approach, all the properties of the class can be reserved in one place so that we can freely manipulate our object. Whenever we want to roll back our object we just need to copy original states. Memento suggests a slightly different approach by dedicating a new class for that job. In the target class, we have two methods to backup and restore state from memento.

Let's use the bank account as an example:

```
public class Account {
   float balance;

   public AccountMemento saveToMemento(){
       return new AccountMemento(balance);
   }

   public void restoreFromMemento(AccountMemento memento){
       this.balance = memento.balance;
   }
}


public class AccountMemento {
   float balance;

   public AccountMemento(float balance) {
       this.balance = balance;
   }
}
```

Let use it:

```
public static void main(String[] args) {
   Account account = new Account();
   account.balance = 100;
   AccountMemento memento = account.saveToMemento();

   //update balance through a transaction
   account.balance = 80;
   //opp, transaction fails. let restore our account
   account.restoreFromMemento(memento);
}
```

I hope you now believe me that Memento pattern is one of the most easy and simple ones. I am sure you have no problem to grasp and use it.

After two very easy and simple patterns, I think it is time to move to a more challenging pattern. I want to discuss the Dependency Injection pattern. One of the major design pattern that Spring introduced to their framework. The name of this patterns remind me lots of memories when reading and learning Spring in early 2000s. Wow, it has been nearly 15 years.

Dependency Injection

This pattern in fact is not in the list of 23 patterns discussed by GoF. However, inspired by the work of GoF, many OOP experts have come up with many new patterns. There is a group of patterns that deal with problems in designing enterprise application (they are often called J2EE patterns). Dependency Injection later on was included in subsequent version Java.

But first, what is the dependency? Long time ago, I wrote one post about dependency injection with Spring on a Java forum and received lots of compliments. I think I should start the discussion on dependency injection by explaining what is dependency. At its simplest form, one class (class A) depends on another class (class B) when it has a property whose type is class B:

```
public class B {
   public void method2(){
   }
}

public class A {
   B b;

   public A(B b) {
       this.b = b;
   }

   public void method1(){
       b.method2();
   }
}
```

B is a dependency that A needs in order to work. Therefore, every time we create A object we need to have B object as well.

```
public static void main(String[] args) {
   B b = new B();
   A a = new A(b);
   //this method depends on A's method2 to work
   a.method1();
}
```

In a large system, there are too many such dependencies and creating of object A is also repeated in many places. It will be hard if we want to change B to another type as we have to find it in all other places and update it.

If we can move all the dependency creation to a single place, it becomes easier when we want to change. This is the core principle of dependency injection which means that dependency should not be defined by object itself but should be injected from outside.

Take the example of message sending below. There are many other examples but I use this because of its simplicity.

```
public class EmailService {
   public void sendEmail(String receiver, String message){
       System.out.println("Send "+ message+" to "+receiver);
   }
}
```

We need to use this service in an application namely MarketingTool. This tool need Email Service in order to deliver message to customers.

```
public class MarketingTool {
   EmailService emailService = new EmailService();

   public void sendMessage(String receiver, String message){
       emailService.sendEmail(receiver, message);
   }
}
```

Now use it in main method:

```
public static void main(String[] args) {
   MarketingTool marketingTool = new MarketingTool();
   marketingTool.sendMessage("Test", "Hello Test");
}
```

We can easily see that we have to hardcode Email Service in the MarketingTool class which is considered as not so good.

```
EmailService emailService = new EmailService();
```

What if we want to switch to SMS message or Facebook messager message? We have to go to the MarketingTool class to change. If we have another class that use Email Service, we also need to update it.

Can we have it in one single place and every we want to update the service, we just have to update in that place? Yes, we can use dependency injection pattern. As I mentioned above, Spring is the most famous framework that advocates dependency injection. Spring in its earlier version uses an XML file (beans.xml) to let users define all the dependencies between objects. Based on this configuration file, Spring has a factory class (ApplicationContext) that returns wired-objects (objects with dependencies injected). Using Spring, we don't have use the keyword new to create any object. Every object can be created using ApplicationContext.getBean() method.

I am trying to reinvent the wheel here but just try to refactor the example a bit by introducing a Factory class that takes care of the dependency injection. Remember injection in English means that we put something into a body.

First, as a best practice with OOP, we use interface to allow flexibility:

```
public interface MessageService {
   public void send(String receiver, String message);
}
```

The EmailService should implments this interface:

```
public class EmailService implements MessageService {
   @Override
   public void send(String receiver, String message) {
       System.out.println("Send "+ message+" to "+receiver);
   }
}
```

We remove the hardcoded EmailService from the MarketingTool class:

```
public class MarketingTool {
   EmailService emailService;

   public void setEmailService(EmailService emailService) {
       this.emailService = emailService;
   }

   public void sendMessage(String receiver, String message){
       emailService.send(receiver, message);
   }
}
```

Now define an injector class with a getBean method:

```
public class Injector {
   public MarketingTool getBean(String bean){
```

```
        if(bean.equals("MarketingTool")){
            MarketingTool marketingTool = new MarketingTool();
            //use setter for injecting. we can use constructor as well
            marketingTool.setEmailService(new EmailService());
            return marketingTool;
        }
        return null;
    }
}
```

Everytime we want to have a Marketing Tool object we just need to call getBean:

```
public static void main(String[] args) {
    MarketingTool marketingTool = Injector.getBean("MarketingTool");
    marketingTool.sendMessage("Test", "Hello Test");
}
```

There you have it. Dependency injection is very important pattern. I hope you don't have any difficulty so far in reading this example.

Next, I will discuss a relatively simple design pattern namely Mediator.

Mediator

Mediator as its name suggests that objects should not directly communicate but the communication should be done via a mediator. Clearly, this approach can reduce number of communication links from many to many to one to many.

Take a look at the following example:

We have a class Mediator that mediating message send/receive between users:

```
public class Mediator {
    public static void sendMessage(User sender, String message, User receiver){
        System.out.println(sender + " send " + message + " to "+ receiver);
        receiver.receive(sender, message);
    }
}
```

In User class we send and message via this medicator:

```
public class User {
    String name;
    public User(String name) {
```

```
        this.name = name;
    }

    public void send(User recipient, String message){
        Mediator.sendMessage(this, message, recipient);
    }

    public void receive(User sender, String message){
        System.out.println("Received "+ message + " from "+sender);
    }
    @Override
    public String toString() {
        return name;
    }
}
```

Now we use it:

```
public static void main(String[] args){
    User ronaldo = new User("Ronaldo");
    User messi = new User("Messi");
    ronaldo.send(messi, "See you in Champion League 2017");
}
```

To be honest, to me this example is not so good to demonstrate the uniqueness of Mediator pattern. I hope we can find a better example later on.

To continue I would like to discuss another simple but powerful pattern namely Composite.

Composite

If you really like OOP, you may realize that composition is one of the best thing OOP offers. One object can be comprised by many objects (single or collection). This makes OOP capable to simulate almost all relationships in the real world: is-a, part-of, has-a etc.

When I work with DHIS2, I appreciate the use of this pattern in building a hierarchy tree of organisation units. With only one single class, i.e. OrganisationUnit, the pattern can model all types of hierarchy structures in the health care section in many countries. The strength of this pattern lies in the flexibility which allows unlimited levels of hierarchy that it can define.

Let's look at the OrganisationUnit class:

```
public class OrganisationUnit {
   String id;
   String name;
   List<OrganisationUnit> children = new ArrayList<>();

   public void add(OrganisationUnit organisationUnit){
       children.add(organisationUnit);
   }
   public void remove(OrganisationUnit organisationUnit){
       children.remove(organisationUnit);
   }
}
```

Let use this pattern to define a hierarchy of administrative level in the US:

US
--California
  --Los Angeles
  --Orange County
--Texas

```
public static void main(String[] args){
   OrganisationUnit usa = new OrganisationUnit("USA");
   OrganisationUnit ca = new OrganisationUnit("California");
   OrganisationUnit tx = new OrganisationUnit("Texas");
   usa.add(ca);
   usa.add(tx);

   OrganisationUnit los = new OrganisationUnit("Los Angeles");
   OrganisationUnit orange = new OrganisationUnit("Orange County");
   ca.add(los);
   ca.add(orange);
}
```

After knowing this pattern, I have used it many of my applications and it appears to be extremely powerful.

If the number of organisation is huge, recursively having parent-children relationship like this can be very expensive. It will fail to load so many objects to the memory to process. We might have to ignore the children attributes or make it lazy loading.

The following change can make the pattern work in case of huge number of nodes:

```
public class OrganisationUnit {
   String id;
   String name;
   OrganisationUnit parent;

   public OrganisationUnit(String id, String name, OrganisationUnit parent) {
       this.id = id;
       this.name = name;
       this.parent = parent;
   }
}

public class OrganisationUnitManager {
   List<OrganisationUnit> organisationUnits = new ArrayList<>();

   public List<OrganisationUnit> getOrganisationUnitsbyParent(OrganisationUnit parent){
       List<OrganisationUnit> units = new ArrayList<>();
       for(OrganisationUnit organisationUnit: organisationUnits){
           if(organisationUnit.id == organisationUnit.id){
               units.add(organisationUnit);
           }
       }
       return units;
   }
}
```

As we only get direct children (immediate) not grand-children for each organisation unit, the load will be reduced significantly.

Do you like it? This is the way I used to deal with problem of hierarchy tree in my projects and it worked very well.

We have gone through many patterns. Might be you very tired? Oop. Did you read it from the beginning without a break? Superb. When I started writing this book, I thought I could finish it in one shot, i.e. write continuously 8 to 10 hours without a break as I believed everything was already in my mind. However, I took many breaks and it lasted many days. When did I start it? I think it was about 10 days ago. Not too bad, isn't it?

I think majority of the time I spent was for finding relevant examples and tried to simplify it to the level that every newbies can understand without removing all essence of design patterns. I hope I have achieved what I have planned.

There is only a couple of patterns left. Let's finish them all.

I will start with one of the easy one: the Proxy.

Proxy

In English, proxy means substitution or surrogate or replacement. I guess you know at least one of these words. This book as I suggested at the beginning is written by a non-native English speaker. I deliberately chose simple words and sentences so that newbies with limited English skills (probably IELTS around 4.5 - 5.5) can understand it easily. Regarding the examples and contexts I tried to use the most generic and universal ones so that everyone can grasp without using a need of a dictionary. When writing these lines, I just quickly read few pages of the very famous book about design patterns: Head First Design Patterns. It contains more than 600 pages but just discuss a handful of design patterns, those they believe most significant. There is no doubt that it is an excellent book written by top OOP experts. I am just skeptical that how many developers can have time and patience to digest all of 600 pages. It might be ok for native speakers but for many developers it is really a challenge.

I hope my book is useful as it requires minimal level of English and OOP skills. Now let me go back to the discussion of the proxy pattern.

We've many times heard about the word proxy in proxy war. The United States and alliances from one side and Russia and Iran from another side support two different sides in Syria war. On the ground, the war seemed to be between Syrian Assad and rebels but at the background, it is between the US and Russia. Similarly, Korean war and Vietnam war are two examples of proxy war. Instead of fighting directly, superpowers (US in one side, and Russia-China in another side) borrowed blood and lives of Korean and Vietnamese for their own war.

That's history but let's go back by taking an example in computer network. Many organisations setups a proxy (a special server) to control flows of data in and out. I still remember in the past, all the browsers behind the proxy be configured so that they can access Internet. Proxy can speed up a request because it caches big files such as images or videos.This concept is very similar to the proxy pattern.

Let discuss it now.

In proxy pattern, an object that is not available immediately, for examples images, audio, or other network or I/O resources, can have a placeholder (a substitution). We name this placeholder as proxy. Both the actual object and proxy implements the same interface so that they can be used interchangeably. The proxy, however, needs a reference to the actual object so that when the actual object is ready, it helps to return the actual object.

In the following example, I use Image as an interface with a method show() and two implementations are ActualImage and ProxyImage. Remember the most important thing: ProxyImage must have a reference to ActualImage. That' all.

```
public interface Image {
   public void show();
}


public class ActualImage implements Image {
   @Override
   public void show(){
       System.out.println("Showing actual image");
   }
}

public class ProxyImage implements Image {
   ActualImage actualImage;

   @Override
   public void show(){
       if(actualImage == null) {
           System.out.println("Loading actual image. Please wait");
           actualImage = new ActualImage();
       }
           actualImage.show();
       }
   }
}

public static void main(String[] args){
   Image image = new ProxyImage();
   image.show();
}
```

Wow amazing. I am satisfied with myself. Too tired and sleepy but there are still six patterns left. Let's do them all. I start with the most simple one: Strategy.

Strategy

Long time ago, when searching information about design patterns, I read a post from a guy who said like this:

```
i++
```

Look, could you name what pattern I am using? It is a incremental pattern. He made a joke because there is an overuse of the term design pattern. Many small things could be described a pattern which might not be necessary. Strategy might be one of that kind.

But watch out. Strategy is trivial but many people don't use it in their project. I guess that is why GoF included it into their list.

What is the strategy pattern by the way?

I hope you still remember the dependency injection I discuss previously. If we don't include the Injector, it will turn to a strategy pattern. So we may say dependency injection is a more advanced case of strategy.

There are plenty examples for this pattern. However, the file compression example I read on Internet sounds very interesting. We can reuse the MessageService in the dependency injection but this time let's use a new one:

There is one interface with 2 implementations:

```
public interface FileCompress {
   public void compress();
}


public class Rar implements FileCompress {
   @Override
   public void compress() {
       System.out.println("Compress file with rar format");
   }
}

public class Zip implements FileCompress {
   @Override
   public void compress() {
       System.out.println("Compress file with zip format");
   }
}
```

We need a Context to use this service. Remember the ApplicationContext class in Spring framework? We use a setter to set dependency for the context object.

```
public class Context {
   FileCompress fileCompress;

   public void setFileCompress(FileCompress fileCompress) {
       this.fileCompress = fileCompress;
   }

   public void compress(){
       fileCompress.compress();
   }
}
```

Now we use it:

```
public static void main(String[] args){
   FileCompress rar = new Rar();
   Context context = new Context();
   context.setFileCompress(rar);

   context.compress();
}
```

For a logic, I think this pattern should be discussed before the dependency injection pattern as they are highly related.

State

State is not a complex pattern. However, examples available on Internet about it could be very confusing. In all electronic devices, there are a button that is in charge of both on and off. When we press the button it will change the status of device from on to off and vice versa. To implement that, we often do:

```
public class TV {
   String state = "off";
   public void press(){
       if(state=="on"){
           System.out.println("Turning off");
           state = "off";
       }
       else{
           System.out.println("Turning on");
           state = "on";
```

```
        }
    }
}

public static void main(String[] args){
    TV tv = new TV();
    tv.press();
    tv.press();
}
```

It works nicely right? But what is the problem with this implementation? The problem is if we want to add a new state such as standby, we have to update the TV class. Is there a way to allow defining new state without changing the TV class? We can use a class for State rather than a String. In this approach, each state will need one class. And all state classes must implement a State interface. The TV class now has a reference to State interface, not a string anymore.

We will move the state change logic out of the TV class and put it into the State implementation. Every time the press method of TV class is called, it will ask its state member variable to update the state.

```
public interface State {
    public void press(TV tv);
}

public class OffState implements State {
    @Override
    public void press(TV tv) {
        System.out.println("Turning on");
       tv.state = new OnState();
    }
}

public class OnState implements State  {
    @Override
    public void press(TV tv) {
        System.out.println("Standing by");
        tv.state = new Standby();
    }
}

public class Standby implements State {
    @Override
    public void press(TV tv) {
        System.out.println("Turning off");
        tv.state = new OffState();
    }
}
```

```
Design Patterns are simpler than you thought
Live support 24/7: https://www.facebook.com/Easydesignpattern-747369588799016/

public class TV {
   State state  = new OffState();
   public void press(){
       state.press(this);
   }
}

public static void main(String[] args){
   TV tv = new TV();
   tv.press();
   tv.press();
   tv.press();
   tv.press();
}
```

Bridge

The problem of design patterns is sometime the way it is named. For example, after examining many examples of the Bridge pattern, I could not see the link between the pattern structure or behavior with its name. Let's see the example below before we discuss what is bridge pattern.

We have an interface namely Drawer which will set the contract for all other implementations. We have two concrete implementations which are ColorDrawer and BorderDrawer. In an abstract class namely Shape, we define a member variable namely drawer that refers to this interface. In concrete classes that extend Shape, we can use this member variable to draw either border or color, depending on our need.

Here is the Drawer interface and its implementations:

```
public interface Drawer {
   public void draw();
}

public class ColorDrawer implements Drawer {
   @Override
   public void draw() {
       System.out.println("Coloring");
   }
}

public class BorderDrawer implements Drawer {
   @Override
   public void draw() {
       System.out.println("Border drawing");
   }
}
```

Here is the abstract class Shape that has drawer as a member variable. Note that we use a setter to assign value to this variable. Using constructor can be problematic when we have to define constructors in subclasses.

```
public abstract class Shape {
   Drawer drawer;
   public void setDrawer(Drawer drawer) {
       this.drawer = drawer;
   }

   public abstract void draw();
}
```

Now we use it in the main method:

```
public static void main(String[] args){
   Shape circle = new Circle();
   circle.setDrawer(new ColorDrawer());
   circle.draw();

   circle.setDrawer(new BorderDrawer());
   circle.draw();
}
```

So what are the benefits of doing it this way? If we want another subclass of Shape such as Rectangle, we need to add only one class. After that we can have up to 6 combinations (2x3) of shape and drawer. If we used abstraction, we would need 6 classes: ColorCircle, ColorSquare, ColorRectangle, BorderCircle, BorderSquare, BorderRectangle. The number of classes required will grow exponentially if we want to have another shape or another drawer.

Command

To me, this is a very controversial pattern. It is controversial because there are different interpretations about how we could use it. At its most basic form, command pattern has a Command interface with a method namely execute.

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

Other advantages of Command such as encapsulation of all information needed to perform an action or trigger an event are very vague. Some say that Command is a form of callback when an object with execute methods is passed to a call. Well, is that a characteristic of OOP?

I shall now take an example that I feel most relevant to the Command pattern in which it allows a command undo its action when needed. You probably know about the Ctrl+Z (or Command + Z) in Windows, right?

In this example, there is a shape that needs to be drawn on a game screen. I recently made a mini game on Android so I would like to use this example. The shape will be encapsulated inside a Command namely MoveShapeCommand. MoveShapeCommand implements the Command interfaces with two method execute() and undo(). In order to undo, the MoveShapeCommand class must have variables to store the previous position of the shape. In the undo method, original position of a shape will be restored. If we have multiple MoveShapeCommand objects, the history of command can be reserved and it is very easy to restore to any of these states. I think that is the real power of the command pattern and it is very specific not vague like other explanation we often hear.

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


public class Shape {
   int x, y;
}

public class ShapeMovingCommand implements Command {
   Shape object;
   int x, y;
   int backupX, backupY;

   public ShapeMovingCommand(Shape object, int x, int y) {
       this.object = object;
       this.x = x;
       this.y = y;
   }
   @Override
   public void execute() {
       this.backupX = object.x;
       this.backupY = object.y;

       object.x = this.x;
       object.y = this.y;
   }
   @Override
   public void undo() {
       object.x = backupX;
       object.y = backupY;
   }
}
```

Note that in the ShapeMovingCommand class we do not put any param to the execute and undo method. Instead, we have a reference to Shape and 2 integer variables to backup its location.

Now use it in main method:

```
public static void main(String[] args){
   Shape object = new Shape();
   ShapeMovingCommand mover = new ShapeMovingCommand(object, 10, 10);
   mover.execute();
   System.out.println(object);

   mover.undo();
   System.out.println(object);
}
```

Iterator

This is probably the simplest pattern. We all know about it, I think. Let's say we have a collection of objects. Can we go through every item in the collection and perform some actions on each item? Yes, it is easy, right.

We often do this with array and arraylist:

```
public static void main(String[] args){
   String[] countries = {"USA", "Australia", "English", "Vietnam"};
   for (int i = 0; i < countries.length ; i++) {
       System.out.println(countries[i]);
   }

   List<String> countries2 = new ArrayList<>();
   countries2.add("USA");
   countries2.add("Australia");
   countries2.add("English");
   countries2.add("Vietnam");

   for (String country: countries2) {
       System.out.println(country);
   }
}
```

By using iterator pattern, we will define a uniform way to access all kinds of collections. A iterator pattern will include an Interface with 2 methods: next() and hasNext. Java offers this interface already but just in case you want to define your own:

```
public interface MyIterator {
```

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


public class StudentList implements MyIterator {
    List<String> students = new ArrayList<>();
    int currentItem = 0;

    @Override
    public boolean hasNext() {
        if(currentItem >= students.size()){
            currentItem = 0;
            return false;
        }
        return true;
    }

    @Override
    public Object next() {
        return students.get(currentItem++);
    }
}
```

Now let's use this pattern:

```
public static void main(String[] args){
    List<String> students = new ArrayList<>();
    students.add("David");
    students.add("Wastor");
    students.add("Mily");
    students.add("Cheery");

    StudentList studentList = new StudentList();
    studentList.students = students;

    while(studentList.hasNext()){
        System.out.println(studentList.next());
    }
}
```

We will move to one of the last two pattern before we complete the book: Chain of Responsibility.

The Chain of Responsibility

Many times, we have to do many things in sequence to complete a job. From management perspective, breaking down a big task with multiple parallel and sequential

small steps is considered a good approach. People also have applied this principle in software coding, aren't they? A program will be broken down to many sub-programs with different granularity classes and methods.

When we have many tasks to be done subsequently, we need to maintain a list of workers who perform the action:

object1.method1();
object2.method2();
object3.method3();

We can use recursive object link to trigger the series of method invocations with only one call.

In the following example, we define an AbstractPost which has a member namely next that refer to AbstractPost itself. We have one abstract method write and one concrete method writeAndSubmit. You could think of an example of posting to social network. The same post is also repeated in different places, i.e cross posting.

```
public abstract class AbstractPost {
   AbstractPost next;

   public void setNext(AbstractPost next) {
       this.next = next;
   }

   public void writeAndSubmit(){
       write();
       if(next!=null)
           next.writeAndSubmit();
   }

   public abstract void write();
}
```

We implement 3 subclasses with different types of writing a post:

```
public class FacebookPost extends AbstractPost {
   @Override
   public void write() {
       System.out.println("Write a FB post");
   }
}

public class InstagramPost extends AbstractPost {
```

```
    @Override
    public void write() {
        System.out.println("Write an instagram post");
    }
}


public class TweeterPost extends AbstractPost {
    @Override
    public void write() {
        System.out.println("Write a tweeter post");
    }
}
```

Now we use it in main. Please note that the invocation of method is done only once but all the nodes in the chain are executed:

```
public static void main(String[] args){
    //Create objects and setup the reference
    AbstractPost facebookPost = new FacebookPost();
    AbstractPost instagramPost = new InstagramPost();
    AbstractPost tweeterPost = new TweeterPost();

    facebookPost.setNext(instagramPost);
    instagramPost.setNext(tweeterPost);

    //everything is done with one call (it is a recursive)
    facebookPost.writeAndSubmit();
}
```

Well done, right? If you have to remember this pattern. Remember the recursive methods. Let's continue with the final one: interpreter.

Interpreter

The last one is very controversial. It is not very common pattern. Only for those who write software about syntax, grammar to process an expression (in computer or natural languages).

To illustrate for this pattern, let's take the following example. If we have a string "2 + 3" how can we return 5.

This can be done by using interpreter pattern. Let's try.

It is a bit complex to process a long expression with complex operand and operators. In this example, I will only try to write a program to solve a very simple express, i.e. 2 operands and 1 operator. The algorithm will be as follower. Split the string by a white space " ". After that, try to evaluate each token (interpret) to see if it is a number or an operator. We will need two different Expression classes to deal with number and operator.

```
public interface Expression {
   public int interpret();
}
```

There are two implementations of this interface. The interpret method of NumberExpression just needs to convert a string to integer. In OperatorExpression, the interpreter needs to check the type of operator to return corresponding values.

```
public class NumberExpression implements Expression {
   String number;

   public NumberExpression(String number) {
       this.number = number;
   }

   @Override
   public int interpret() {
       return Integer.parseInt(number);
   }
}

public class OperatorExpression implements Expression {
   String operator;
   NumberExpression left;
   NumberExpression right;

   public OperatorExpression(String operator, NumberExpression left, NumberExpression right) {
       this.operator = operator;
       this.left = left;
       this.right = right;
   }

   @Override
   public int interpret() {
       switch (operator){
           case "+":
               return left.interpret() + right.interpret();
           case "-":
               return left.interpret() - right.interpret();
           case "*":
```

```
                return left.interpret() * right.interpret();
            case "/":
                return left.interpret() / right.interpret();
        }
        return 0;
    }
}
```

Now let's use it:

```
public static void main(String[] args){
    String s = "2 + 3";
    String[] tokens = s.split(" ");

    NumberExpression ne1 = new NumberExpression(tokens[0]);
    NumberExpression ne2 = new NumberExpression(tokens[2]);
    OperatorExpression oe = new OperatorExpression(tokens[1], ne1, ne2);

    System.out.println(oe.interpret());
}
```

You can see now that we have successfully implemented the interpreter pattern. For simplicity, in this example I hardcoded the size of the expression to only 1 operator and 2 operands. We can deal with more complex expression if we use a Stack to push and pop tokens. But in order to do that, we have to convert our expression from in-fix to post-fix. For example:

2 + 3 should become 2 3 +
2 + 3 - 1 should become 2 3 + 1 -
(2 + 3 - 1) / 2 should become 2 3 + 1 - 2 /

But that's another story. Our aim is to understand Interpret design pattern. And I hope you have got the idea from reading and practice.

Pattern in one sentence

If I allow to use one sentence to describe about a pattern. I think my sentence should be as follows:

1. Factory method: hide the use of keyword new behind a method.
2. Builder: return an object of the same class where the method is define.
3. Singleton: a static field (properties) that refers to the class itself

4. Prototype: define a method to copy an object to another object
5. Facade design: an aggregator class uses methods from its member objects to pretend that it can do the same jobs.
6. Template method: a class defines a methods that invoke two abstract methods, i.e. not defined yet.
7. Adapter: implements an interface that uses methods provided a member object.
8. Observer: an object subscribes a subject by becoming a member of a list inside subject
9. Visitor: a class invokes a method of another class in order run a method of itself.
10. Decorator: an interface implementation that has a field referring to the interface itself.
11. Flyweight: use a Map to cache and reuse similar objects instead of creating new.
12. Memento: an object with two methods backup and restore to backup properties ofobjects of another class.
13. Dependency Injection: use a factory to set properties for an object (external setter)
14. Mediator
15. Composite: a class that has a field that is a collection of objects from the same class.
16. Proxy: a class that has a member from the actual class that it substitutes
17. Strategy: just about the use of interface
18. State: each state should be defined as a concrete class and has a method to decide the next state
19. Bridge: an abstract class that has a field that refers to an interface
20. Command: class that implements execute method from a command interface and contain all member objects required for that execute method to work.
21. Iterator: implement an interface with two method: next() and hasNext()
22. Chain of Responsibility: an abstract class that has a field namely next which refer to the abstract class itself.
23. Interpreter: use an interface with interprete method to evaluate an expression