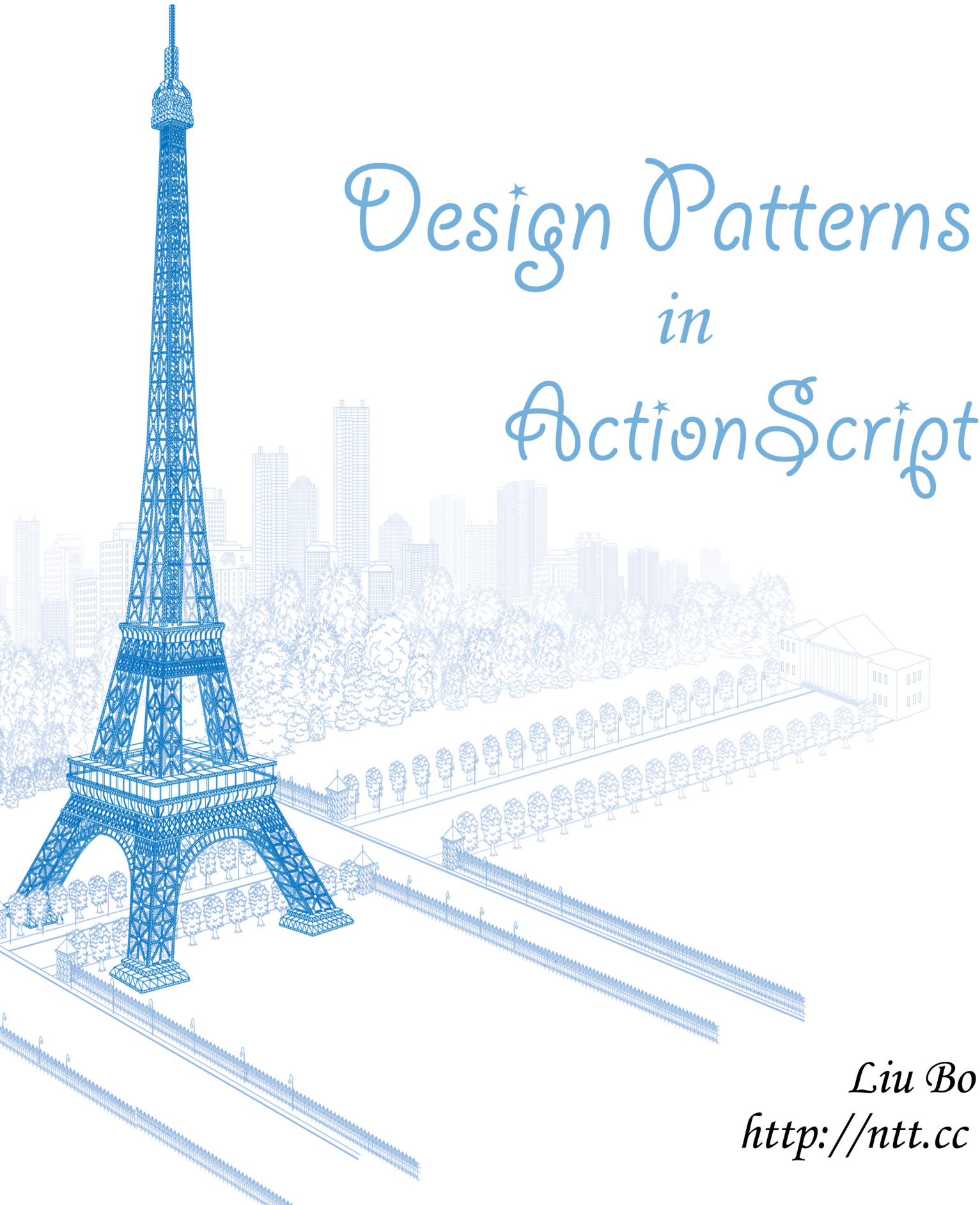


Design Patterns in ActionScript

Liu Bo
http://ntt.cc

Contents

Preface

About this series

Last summer, when I was busy finding the internship opportunity, I got a message from my classmate. It said that, someone was looking for a writer to write some articles about Flex/Flash. Though I just had little experience on Flex and Flash, I sent a message to the email address given on the message. Then I got this job, and began to write something about FlexUnit.

Minidxer is a very kind person, when he knew that I'm not familiar with the unit test framework, he gave me a new subject, the 23 design patterns from GoF book. In fact, I just knew few simple patterns, but I wanted to learn all the 23 patterns. So, I prepared to write the articles about this.

During the preparing time, I got an internship offer from Ericsson. As you think, I began my intern life at Ericsson, and used my spare time writing the articles. You know, it's not easy to write something you're not familiar with. And I don't have much time after finish my job. Further more, I began my job hunting in September. All those things result that I only finished five patterns in about four months, from August to November.

After finish my internship and job hunting, I began to write the left patterns. Though I spend many times preparing, there are still many mistakes in the articles. I'm just a fresh man in this field :)

The main reason I wrote these articles is that I want to learn all the 23 patterns. And it would be my honor that these articles can give you some help.

Some suggestions

When you begin to learn a new pattern, carefully look at its intent, every pattern has its corresponding problem field.

Then, try to understand the example, understanding how this pattern affects the current design, what problems it solves. If the example is not so clear, use Google to find more examples about this pattern.

In general, you may find many examples write in C++ or Java; try to transform them into ActionScript. It's a good way for practice. And with these practice, you'll know the pattern clearly, not only the implementation, but also its nature.

During you learning of patterns, some books are necessary. You should take the <Design Patterns> by GoF on your desk. And other books, such as <Head First Design Patterns>, are also useful for you, which depend on your level, pick up some books that fit you.

Remember, design is a compromise, when it solves some problems, it brings new problems. So, there is no best pattern, but suitable one.

Liu Bo

2009-3-13

About me

I'm a senior student of SCUT, and my interest including programming, swimming and billiards. The following picture is me :) If you have any problems, contact me via the email.

Liu Bo(刘 毫)

liubo.cs@hotmail.com

Strategy

Today, we're going to talk about the design patterns in ActionScript. You may be familiar with those patterns if you're familiar with OO(object oriented). But I won't suppose you have much knowledge about it, all you need to know is just some basic OO principles, such as encapsulation, inheritance and polymorphism.

This is the first topic of design patterns, and you'll see all the 23 design patterns in the following days. Hope you will enjoy these topics.

Let's begin the first one now.

Suppose you need to write some codes to mimic some birds, the first is eagle, and the other is penguin. Those two animals have two methods named migration() and fly(). You may realize you need a super class named bird, and two methods in it. Then eagle and penguin can inherit from the super class, and override the methods, 'cause those birds have different behaviors. It seems to be perfect. You quickly write down the code.

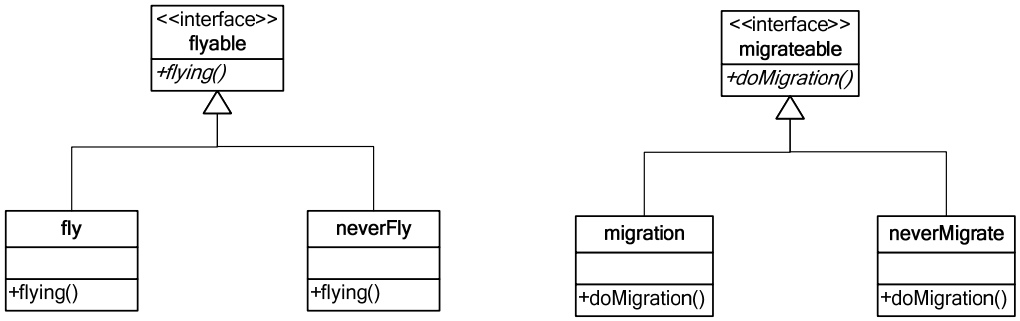
```
class bird {
    public function  migration():void{
    }
    public function fly():void{
    }
}

class eagle extends bird {
    public  override function  migration():void{
        trace("Sorry, I don't migrate");
    }
    public  override function fly():void{
        trace("flying");
    }
}

class penguin extends bird {
    public  override function  migration():void{
        trace("migrating...");
    }
    public  override function fly():void{
        trace("Sorry, I can't fly...");
    }
}
```

OK, it does what you want. But what would you do when you want

to add a new kind of birds into your program. Suppose you need to add a duck into your program now, you can inherit from the super class bird and override the methods again. It seems that the super class doesn't save your time, reuse your code, you need to write every methods of subclasses.

How to fix this problem? I think you need this design pattern—Strategy. And here is the definition from the GoF book. "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." And what're the algorithms here? The methods, migration() and fly(). Every bird has it's own behavior about migration and fly, it migrates or not, it flies or not.

Let me say it more clearly, we just need to implement two situations about migration and two about fly. We need to implement each situation on a single class. And we'd better to have two interfaces named flyable and migrateable, and the classes just need to implement those interfaces. The UML diagram is showing below.



Now, the super class bird needs to handle two variables of those interfaces. Using polymorphism, the super class bird can do all things we do in the subclasses. When you want to ask a eagle to fly, you can simply call the function doFlying() of class bird. You can write the code like this:

```
var littleEagle:bird = new eagle();
littleEagle.doFlying();
```

I know you may be confused now, let me show you the code of the super class bird.

```
class bird {

    var flyInterface:flyable;
    var migrateInterface:migrateable;

    public function bird(_fly:flyable,_migrate:migrateable){
        flyInterface = _fly;
        migrateInterface = _migrate;
    }
```

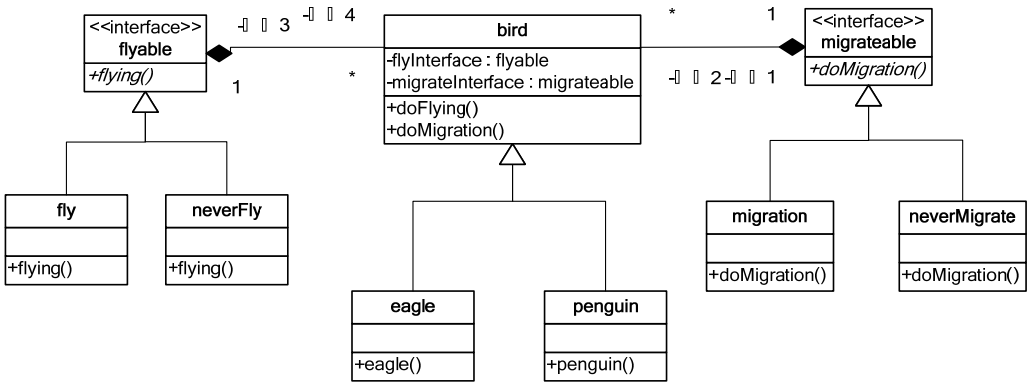
```
        public function   doMigration():void{
            migrateInterface.doMigration();
        }

        public function doFlying():void{
            flyInterface.flying();
        }
    }
```

From the code, we can see the super class do all the things simply by call the methods of the interfaces. We don't have to know which instance the interface points to, we only need to know it works. That's it. Well, we'll use the constructor to specify the instances of each interface. And here comes the code of each subclass.

```
    class eagle extends bird {
        public function eagle(){
            super(new fly(), new neverMigration());
        }
    }

    class penguin extends bird {
        public function penguin(){
            super(new neverFly(),new migration());
        }
    }
```

If you want to add a new kind of birds, all you need to do is just write a new class, and specify the constructor. And that's it!

If you want to change the behavior of fly or neverFly, you just need to change the code of the specific class. And all the birds' behavior will get changed. So, the code will be more easily to maintain. In this sample the behavior is easy to change, so we need to encapsulate the behavior. The UML of all classes is showing below.



Let's back to the definition of this pattern, "Define a family of algorithms, encapsulate each one", we have done it. And the result is we can change the behavior or add a new bird more easily. The algorithm means the way you solve a problem, when you have many ways to solve a problem, encapsulate it, and make it interchangeable.

This sample is not good enough to show the power of this pattern, if you want to learn more about this pattern, you can look up the GoF book.

Factory Method

In our last topic, we talk about the strategy pattern. And it helps us to encapsulate the change of algorithm. Today, we go on talking about the birds. We will consider how to generate the birds, eh, I mean the classes.

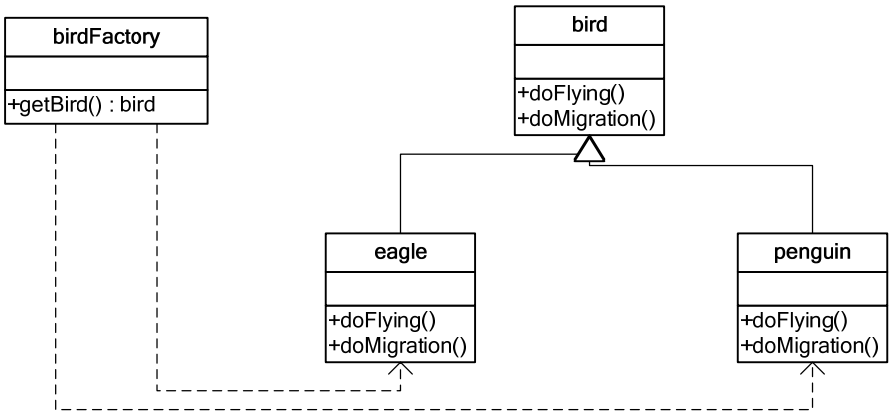
Now, we have the classes of eagle and penguin. We can use them anywhere we want. Further more, we can write down the following code to decide which class should be initialized.

```
public class birdFactory {

    public function birdFactory(){
    }

    public function getBird(type:String):bird{
        if(type == "eagle")
            return new eagle();
        else
            return new penguin();
    }
}
```

Now, we just need to look at the type, and we can decide which kind of birds to be initialized. This is a pattern called simple factory, it seems that everything goes all right.

I don't want to talk too much about this pattern, because it is not our topic today. And I want to introduce you a principle of OOD, called Open-Close-Principle, which means we should open for extension, and closed for modification.

Let's take a sight at the UML diagram of the above code. Then discussing where the problem is.



Eh, Here comes the problem. Look at the code I just write down, what'll happen if I want to add a new kind of bird. It seems I need to

change the method of **getBird().** That's right, this is the problem of this pattern. It doesn't fit the O-C-P. So, we need another solution.

Remember the Open-Close-Principle. We will follow this principle to get the result.

We want a factory to produce the class, and when we also want the factory doesn't need to be modified, if we want to add a new kind of bird.

Then we'll need another two classes of factories, named eagleFactory and penguinFactory, paired to the birds.

The birdFactory will change into interface.

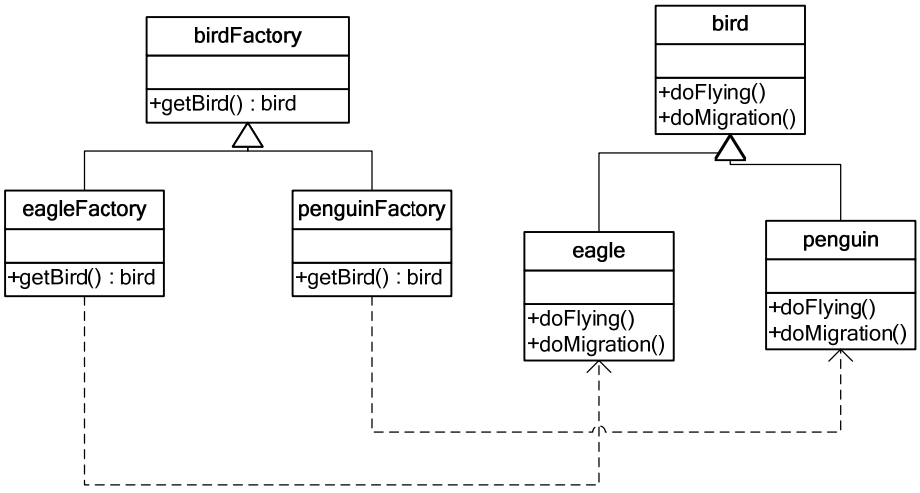
```
public interface birdFactory {
    function getBird():bird;
}
```

And the concrete factory will implement the birdFactory. Here is the code of eagleFactory.

```
public class eagleFactory implements birdFactory{
    public function getBird():bird{
        return new eagle();
    }
}
```

It looks like very simple and clean, isn't it? Another concrete factory, penguinFactory is much the same.

Now the UML diagram will change into:



One factory, one kind of birds. If you want to add a new kind of birds, you need to add two more classes. It sounds a little red-tape, but you'll get the rewards. You don't need to change the code, and the code is easy to extension. It means when the requirement is changed, you can easily fit for it. And that's why we need design patterns, it saves our time.

That's all about this pattern, and we will talk about another related pattern next time.

Abstract Factory

Do you still remember the factory method pattern we talked about last time? If you're already forgot it. May be you can take a look at the following intent, which was defined by the GOF book.

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

-- By THE GOF BOOK

In out last example, we use eagleFactory and penguinFactory to decide which bird should be instantiated. And we can easily extend this program only by adding some new classes; we don't need to change the main framework.

OK, now let's do something more about the last example.

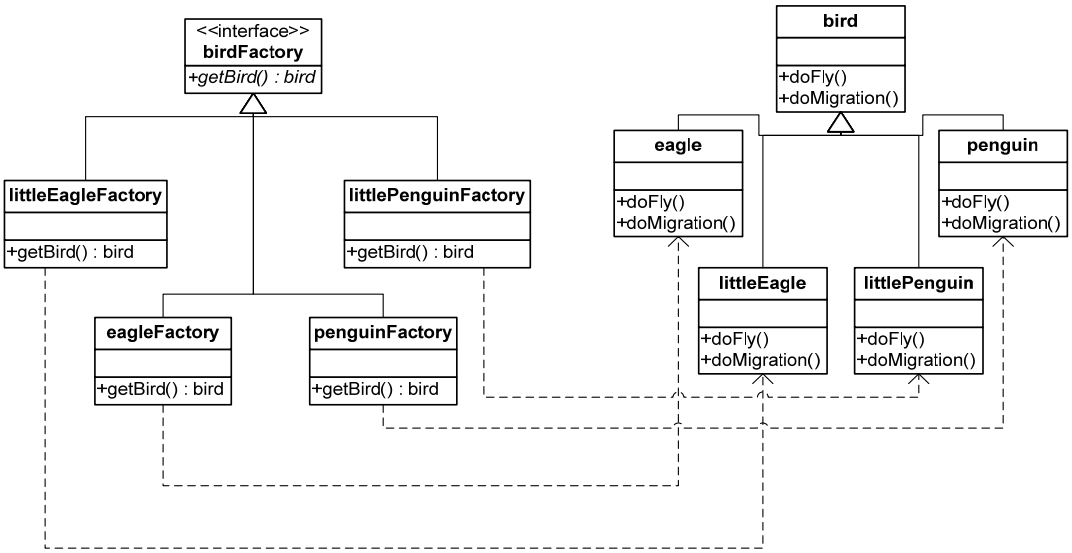
First, let's add a new class named littleEagle, and the little eagle doesn't have the ability to fly, and not migration. So, we can write the code as follows.

```
class littleEagle extends bird {
    public function littleEagle(){
        super(new neverFly(), new neverMigration());
    }
}
```

Then, we need to add a bird related to penguin, eh, I wonder that whether there is some kind of penguin flies and not migration. It seems that god doesn't make that kind of creature. So, we need to make it, and we can call it littlePenguin. Of course, it was faked :) The code is as below.

```
class littlePenguin extends bird {
    public function littlePenguin(){
        super(new fly(),new neverMigration());
    }
}
```

According to the factory method pattern, we need to add two more factories to produce these two new products. And the UML diagram becomes like this.



How's your feeling about this diagram?

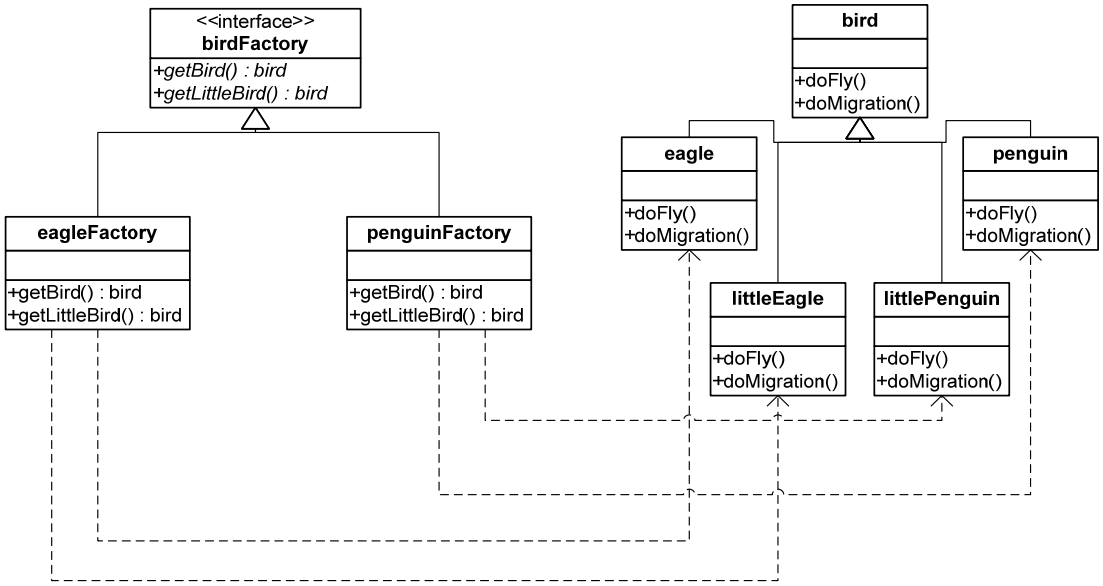
Four factories correspond to four birds. Looking deep inside, we can found that littleEagle is much the same as eagle, and the same to the other two classes. If we want to add bigEagle and oldEagle, we'll need another two more factories, and it sounds unreasonable. The eagles can be looked upon as series products. Maybe they should be produced by the same factory.

Now, I should show you the new pattern, named abstract factory pattern. Its intent was defined as follows.

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

-- By THE GOF BOOK

Using this pattern to change our existing code, we can get the following UML diagram.



Of course, the code has to be changed. Let me show you the code of eagleFactory.

```
public class eagleFactory implements birdFactory{
    public function getBird():bird{
        return new eagle();
    }
    public function getLittleBird():bird{
        return new littleEagle();
    }
}
```

Abstract Factory and Factory Method are both about generating the new object, while abstract factory cares more about a family of object.

Enjoy!

Adapter

Our company is building a new project recently. This project is about simulated flight. The total solution will include the software module and hardware module. Unfortunately, our company just a software company, we don't have the ability to design the hardware. So, our company wants to buy the hardware solution from other company, we only need to work out the software.

The software part looks like just a piece of cake to us. So, we quickly build a team to handle this task.

After many and many overtime, the team has finished the software module. And all the operation with hardware is reserved as an interface.

At the same time, our boss has met some hardware suppliers and decided to make a deal with one of them.

Sooner after the deal, we receive the hardware driver library. Things not always go that easy. This time, we found out that the hardware driver library is not fit for the interfaces reserved by our software.

This is a big problem. If we can't fix it, we'll only have two ways to go, one is rewriting our code to fit for the hardware driver library; the other is buying a new hardware solution. Eh, maybe we have the third choice, that's we quit our jobs.

We were very depressed by this situation. But life still goes on. We check the driver library carefully, and we found that many functions can provide the function we need, but it has a different name. We can't change the driver library. Does it mean that we need to change our code?

Obviously, we don't want to do that, it's not an easy job, further more, we're all lazy guys. So, we need a tricky solution.

Genius programmers, what will you do if you're in this situation?

Finally, a genius guy in our team gives a suggestion. This solution is much like building a bridge from the interfaces reserved in the software module and the functions provided by the driver library.

The solution is that, we build a new class to wrap driver library, and the new class will derived from the interfaces reserved by the software.

The original code is showing below.

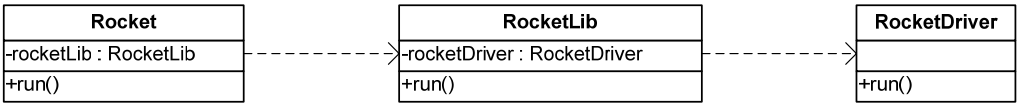
```
public class Rocket{
    private var rocketLib:RocketLib;
    public function Rocket(){
        rocketLib = new RocketLib();
    }
    public function run():void{
        rocketLib.run();
    }
}

public class RocketDriver{
    public function run():void{
        trace("The Rocket Runs!");
    }
}
```

As you see, the Rocket class is in our software module, and the RocketDriver class is in the library they provided. And the solution is as follows.

```
public class RocketLib{
    private var rocketDriver:RocketDriver;
    public function RocketLib(){
        rocketDriver = new RocketDriver();
    }
    public function run():void{
        rocketDriver.run();
    }
}
```

Now, the software module doesn’t need to be changed. And everything goes well.

The UML diagram is as below.



Do you like this solution? Of course, this is a pattern named adapter pattern. The intent of this pattern is as follows.

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

--By GOF BOOK

Hope you have a sensitive feeling about this pattern now.

Enjoy!

Decorator

When we build some flash applications, we spend most of the time on two things. One is image, the other is text. Eh, I don't want to talk about the image. Because in most cases, I don't need to deal with it, it belongs to others. So, I want to talk something about the text.

Sometimes, we want to capitalize all the text, while sometimes we want all of them be lowercased. Sometimes, we want the text color is red, while sometimes is blue. We need to care the case, the color, and more over, the font, or something else.

I'm tried of this thing. I don't know whether you guys have any idea to solve this trouble. If you have any, please tell to me via the email.

Now, I'll introduce my solution. Firstly, we need a basic class to implement the basic function.

The basic class's code is as follows.

```
class OutputText {
    public function write(s:String):void{
        trace(s);
    }
}
```

After we finish the basic class, let's consider the other classes. We need to deal with the format in the other classes. So, we can write some classes to wrapper the basic operation. The concrete format class will inherited from the basic class, and override the basic operation. The override function will do some other thing to format the text.

Here is a concrete format class code.

```
class OutputAppendText extends OutputText {
    private var output:OutputText;
    public function OutputAppendText(output:OutputText){
        this.output = output;
    }
    public override function write(s:String):void{
        s+=" {APPEND TEXT}";
        output.write(s);
    }
}
```

Here is another concrete format class code.

```
class OutputLowercaseText extends OutputText {
    private var output:OutputText;
    public function OutputLowercaseText(output:OutputText){
        this.output = output;
    }
    public override function write(s:String):void{
        output.write(s.toLowerCase());
    }
}
```

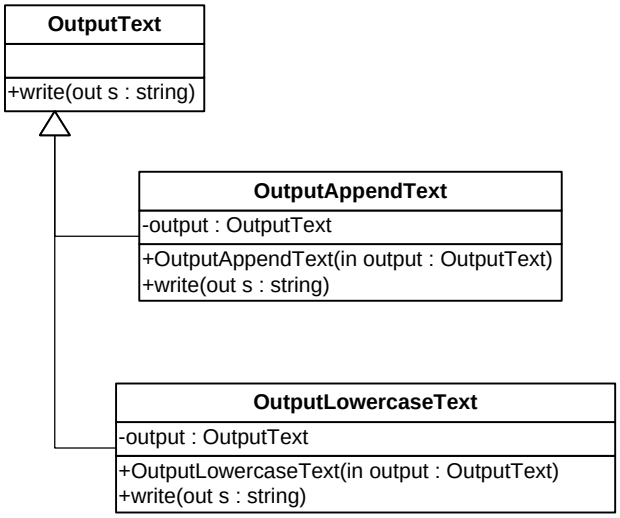
As you see, two concrete classes both inherited from the basic class. So, you can use one concrete class to wrap another concrete class without considering the sequence.

Here is the test code.

```
var output:OutputAppendText = new OutputAppendText(
                                        new OutputLowercaseText(
                                                new OutputText()));
output.write("skfjaslf");
```

If you want to add some other format, you just need to write some other class just like the classes mentioned.

And the UML diagram is as follows.



In this class hierarchy, there is only one basic class, which defines the basic operations. All the other classes will be inherited from the basic class, and override the basic operations. Every sub class will do something in the override function to implement the concrete operation.

This pattern is called decorator. The GoF's definition is as follows.

Attach additional responsibilities to an object dynamically. Decorators

provide a flexible alternative to subclassing for extending functionality.

--By GOF BOOK

This pattern is very useful when you deal with the input/output or formatting the text. The java I/O system is a famous example of this pattern.

Enjoy!

Facade

Have you ever heard Xerox PARC (Palo Alto Research Center)? This research center has many great innovations. GUI (Graphical User Interface) is one of them.

The history of GUI is an interesting story. After the PARC invent the GUI, they didn't put this into business immediately. Sometimes later, a young man visited this center, and was shocked by the great innovations including the GUI. When the young man backed to his company, he put those ideas into their product, Macintosh, the first commercially successful product using GUI. And the young man, the CEO of that fruit company, began his legend in Silicon Valley.

OK, let's turn to our topic today. Have you ever think about something under the GUI? Such as, when you click a button, how the computer will do? I mean the way, not the result. For example, when you click "open", maybe it will show you a file chooser dialog. The file chooser dialog is the result, and the way to get the result maybe very complex. I think this is the great of GUI, you needn't know the way, and you just need to know the result. All the details were covered by the system.

This situation was very common in our daily life. When you drive your car, you just put your foot on the accelerator pedal. Of course, you don't know the details how the cat started, except you've ever studied it.

There is a pattern corresponding to these situations. It was called Façade.

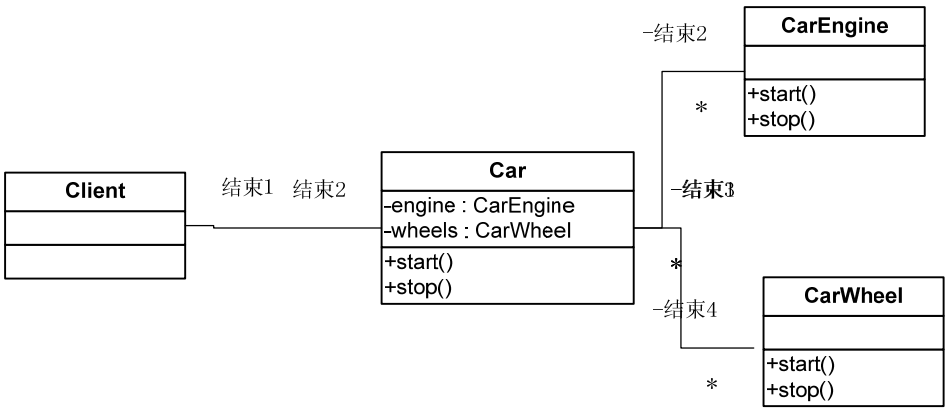
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

-- By THE GOF BOOK

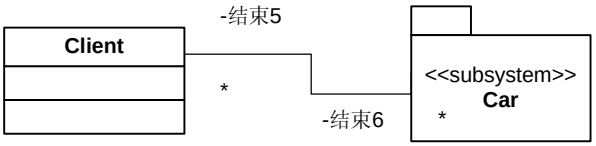
In the car example, you don't know the inner structure of the car. And the car supplies you a higher-level interface, the accelerator pedal. Let's mimic this example.

In order to make it easier to understand, we will only use three classes to illustrate this example, the Car, CarEngine and CarWheel.

You can see the diagram below.



Also, it can be described as blew.



From the client side, the car is a subsystem. And the client doesn't need to know the internal structure of the car. All he need is the high-level interfaces.

You can see the source code for more information.

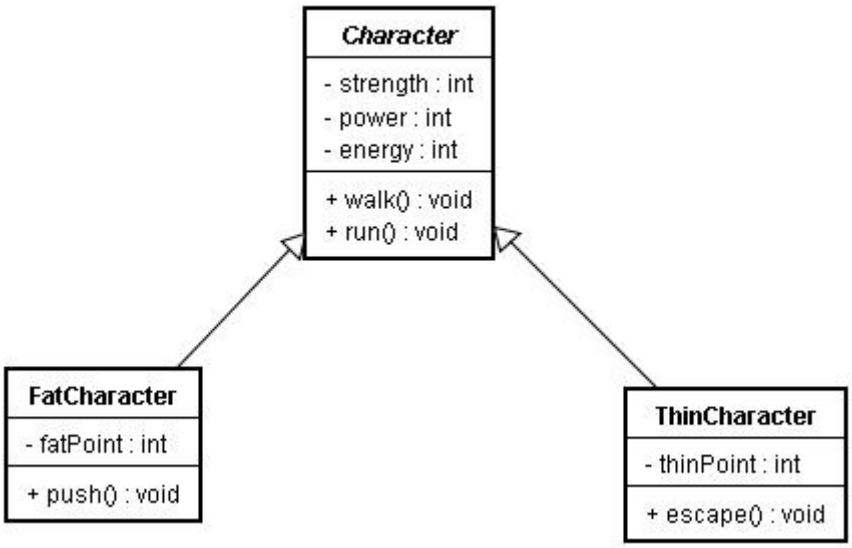
As you see, this pattern is mainly use for reduce the complex of the subsystem, when you want to use some function provide by a subsystem. You don't need to know the details about the subsystem, if you use this pattern. You just need to know the interfaces. It's very helpful when you cooperate with others.

Enjoy!

Bridge

I think many of you are the fans of Diablo. I'm one of you :) In Diablo series, you can choose different characters to begin your Diablo journey. As a programmer, maybe you'll write a base class, named Character. Suppose there are only two characters, named FatCharacter and ThinCharacter.

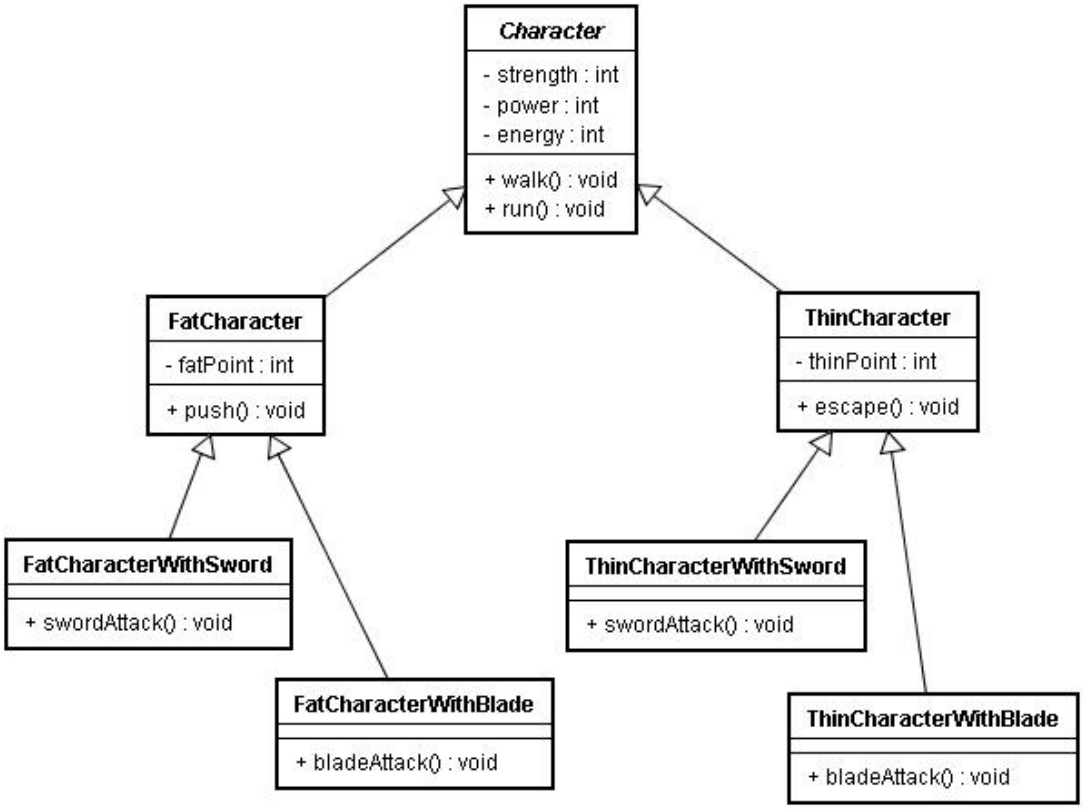
And we will define some common operations in the base class Character, define special skills in the derived classes.



Now, you can use the characters to walk around. However,, it's dangerous to walk around in Diablo, because the monsters are everywhere, so the characters need to protect themselves. The FatCharacter can push the monsters, and the ThinCharacter can escape from the monsters' attack. Maybe it's not enough; we'd better give the characters some weapons, like sword or blade.

Now, our implementation is not suitable, so we need to change. A solution is, add more classes, such as FatCharacterWithSword, FatCharacterWithBlade and so on, and let these classes inherited from the FatCharacter and the ThinCharacter.

The class hierarchy will be as follows.



Now, the characters can use the weapon to beat the monsters.

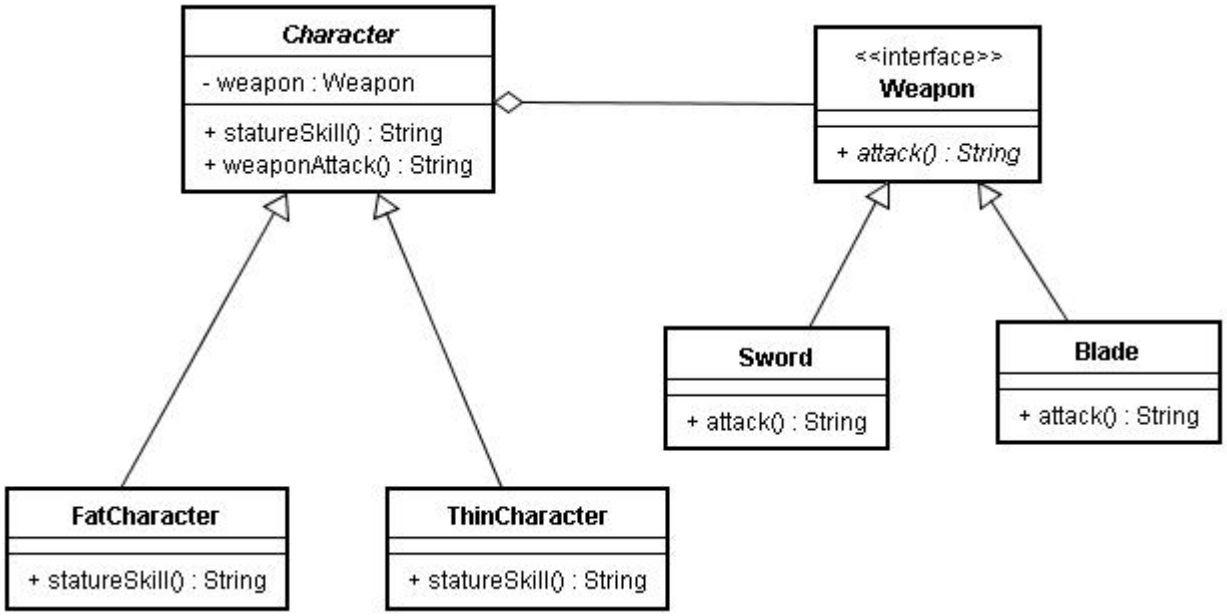
Do you like this implementation? In this class hierarchy, you can see some classes are exactly the same, especially in level 3. If you want to add some other weapons, you may need to write the classes twice, one for the FatCharacter, and another for ThinCharacter. If you want to add a normal stature character, it would be worse and worse. Class explosion!

Watch the implementation carefully. There are two varying points, one is the character's stature, and the other is the weapon. These two should be varying independently, not together. So, we need a new pattern to solve this, it is bridge pattern.

Decouple an abstraction from its implementation so that the two can vary independently.

-- By THE GOF BOOK

In order to let the two vary independently, we should use composite instead of inheritance. Let the weapon be a part of the character, and the sword and the blade implement the weapon interface. Now the class hierarchy will be as follows.



Now, if you want to add a normal stature character, you just need to write a new class, not two or more, also, if you want to add an axe, you just need to write a new class and implement the weapon interface. You character can use the new weapon to attack the monsters.

In general, composite brings more flexibility than inheritance. Another important thing is separate the varying point, let them vary independently.

For more information you can take a look at the GoF book!

Enjoy!

Singleton

In our real world, many things are one and only. For example, there is only one god in our world, and only one president in the USA, eh, I mean the current president. One and only is very important to our world, and so does to our program.

In some case, we may only need one instance of some classes. If you have any experience on developing the GUI application, you can feel it. For example, there's only one display object in J2ME application, and only one form in windows application. During those situations, we need to control the number of the class, and it was forbidden that using the new operator to generate a new instance. Further more, we need to supply an interface for accessing the one and only instance.

That's all about the singleton pattern. And the intent defined by GOF is as follows.

Ensure a class only has one instance, and provide a global point of access to it.

--By GOF BOOK

Now, we come to the operation layer.

In general, we will meet two problems here, one is when to initialize the instance and the other is how to ensure thread-safe in the multithread environment. Because Action Script 3 doesn't support multithread, so we don't need to care the thread-safe.

Before we solve the first problem, we need to provide a global access point for getting the instance. Of course, we don't except the users use the new operator to get a new instance. Maybe we could change the constructor modifier from public to private. Eh, if you really do so, you'll get a complier error, because in Action Script 3, the modifier of a constructor can only be public. So, we need to change our method.

If we can't change the modifier, the constructor can be called outside. One solution is we can throw an error in the constructor, just like below.

```
public function President()
{
```

```
            if(president != null)
                throw new Error("You shouldn't use the new operator!");

        }
```

If there is already have an instance, we need to throw an error. If not, let it go, and then we can get an instance.

This is not a good way, but it works :). Now, let's come to the first problem. In general, there will be two kinds solution. One is called early initialization, the other is lazy initialization. The first one initializes the instance when the program first runs, and the other initializes the instance when the getInstance () method called.

The code below shows early initialization. The instance will get initialized when the class gets initialized.

```
        static var   president:President = new President();

        public static function getInstance():President
        {
            return president;

        }
```

The code below shows lazy initialization. As you see, after the getInstance() method was called, the instance will get initialized.

```
        static var   president:President = null;

        public static function getInstance():President
        {
            if(president == null)
                president = new President();
            return president;

        }
```

Which way should be taken depends on your demand. Actually, you'll need to consider these two situations only when you using this pattern to get a big object in the memory limit environment.

Enjoy!

Observer

In GUI programming, event-driven is an important concept. Such as, when you programming in Flex, you may put a button on the stage, then you'll add a callback function to the button. The function will be called when there is some action play on the button, such as click. This is almost the same with the observer pattern.

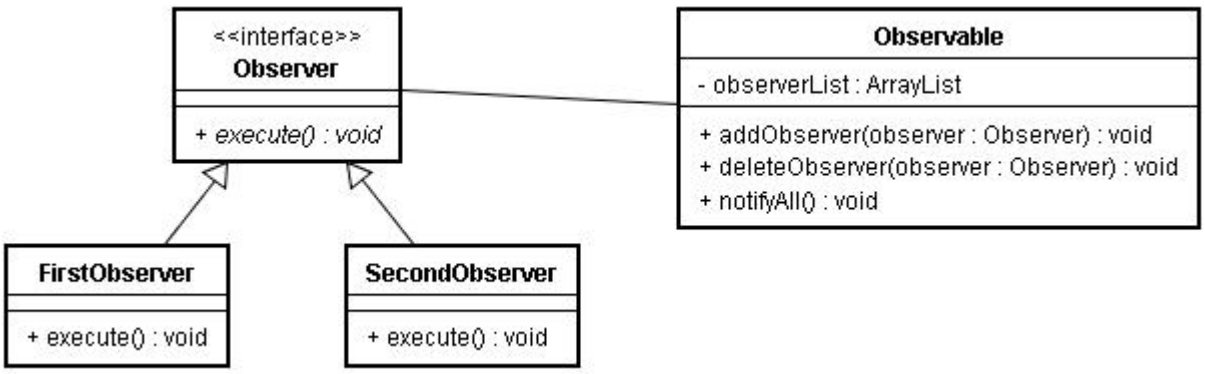
Make it clearer. Firstly, we care the state of the button; we want know when the button was clicked. Secondly, when the button state was changed, we want to do something, that's the callback function. Actually, you can consider as the function cares the state of the button. When the button state change, the function will do something. The button is observable, and the function maybe the observer.

Now, let's come to the intent of this pattern.

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

-- By THE GOF BOOK

I'll give you a new example to illustrate this pattern.
The class diagram is as follows.



As you see, there are only two parts in our example. One is Observable, and the other is Observer.

The role of Observable is holding the observers in a list, and notifies all the observers when the state was change. Here, we don't have the state variable, and when you click the Notify All button in the demo the notifyAll() operation will be execute.

The Observer's role is much simpler, just execute or update. Of course, you need to figure out which subject you'd like to focus on,

and you may need to call the Observable.addObserver(yourself) to let you appear in the observer list. Then when the state is changed, execute or update method will be called automatically.

And this is pattern is also called Publish/Subscribe pattern. It means if you subscribe something, such as newspaper, you'll get it sooner after the newspaper was published.

This pattern is very useful in dealing with GUI programming. Take it with you!

Template Method

Do you like playing cards? If your had ever played, you may noticed that everyone has their own way arranging the cards. And in most cases, people will put the cards in order, maybe from the biggest one to the smallest one. Eh, this is a way of sorting.

We can write down the following code to mimic this action.

```
public function sort(array:Array):void
{
    for(var i:int = 0; i < array.length; i++)
    {
        for(var j:int = 0; j < array.length-i; j++)
            if( (int)(array[j]) < (int)(array[j+1]) )
                swap(array,j,j+1);
    }
}
```

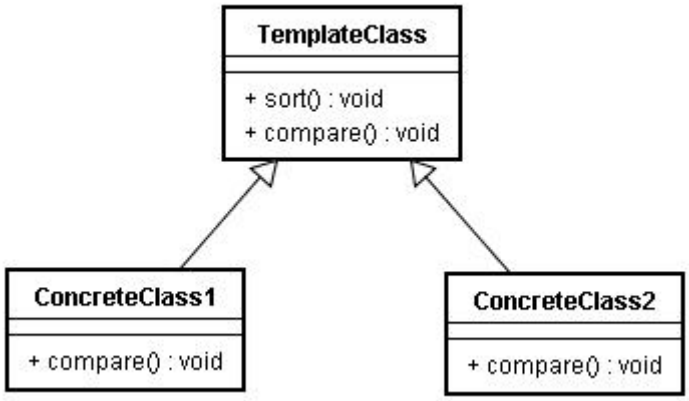
Here, I use the bubble sort, not insertion sort. Because I think the bubble sort is easy to understand. Using this sort will make you cards arrange from the biggest to the smallest, and the sequence is from left to right. Eh, maybe you don't like to this sequence, you'd like to hold the cards from the smallest to the biggest, and also it's from left to right.

Of course, you can change the code with your sorting logic. But, you'll find that you just need to change the compare logic. Maybe, we can extract the compare logic to another function. And the code will be as follows.

```
for(var i:int = 0; i < array.length; i++)
{
    for(var j:int = 0; j < array.length-i; j++)
        if( compare(array,j,j+1) )
            swap(array,j,j+1);
}
```

Now, we can rewrite the compare function to get a different result. Let's go further, we don't need to implement the compare method now, we can delay it to the subclass. Then, each subclass can get its own way of sorting by override the compare function.

And the class diagram will be.



Aha, a new pattern! And it's called template method.

The intent is here.

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

-- By THE GOF BOOK

As you see the skeleton of our sorting algorithm is defined in the template class, and the compare function is use for the subclass to redefine.

This pattern is very useful especially when your high level design is stable, but the detail needs to change frequency.

Enjoy!

Iterator

There is a famous saying in computer science, "Program = Data Structure + Algorithm". It figures out the importance of how to organize the data and how to deal with the data.

Maybe, in your applications, you care more about how to show the data, because this has much to do with the user experience. However, how to organize the data and deal with the data is also or much more important, because this has much to do with the performance.

Eh, I don't want to talk about the performance, it is too big. I just want to talk something has a little to do with how to organize the data. In our daily programming, we will use array, list, or set to store the data. If you want to know the differences between these structures, you'd better to find a textbook in this field.

Array, list and set have their own way to store the data. The array uses continual space to store the data, while the other two using disjunction space to store the data. The difference in storing the data, leads the difference on locating the data. Find the n-th element in array will be much different from the same operation in list or set.

In general, we will write some concrete class for these structures. But, sometimes, we need to access the data without considering the concrete data structure. Or sometimes, we need to define another method for traversing the data.

If you have the requirements like mentioned above, I think you can take a look at this pattern, Iterator. The intent is as follows.

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

■ **By THE GOF BOOK**

Here, I will show you a simple example to illustrate this pattern. In this example, I will just use the array, and provide a backward traverse.

The definition of Iterator is as follows.

```
interface Iterator
{
    function first():void
```

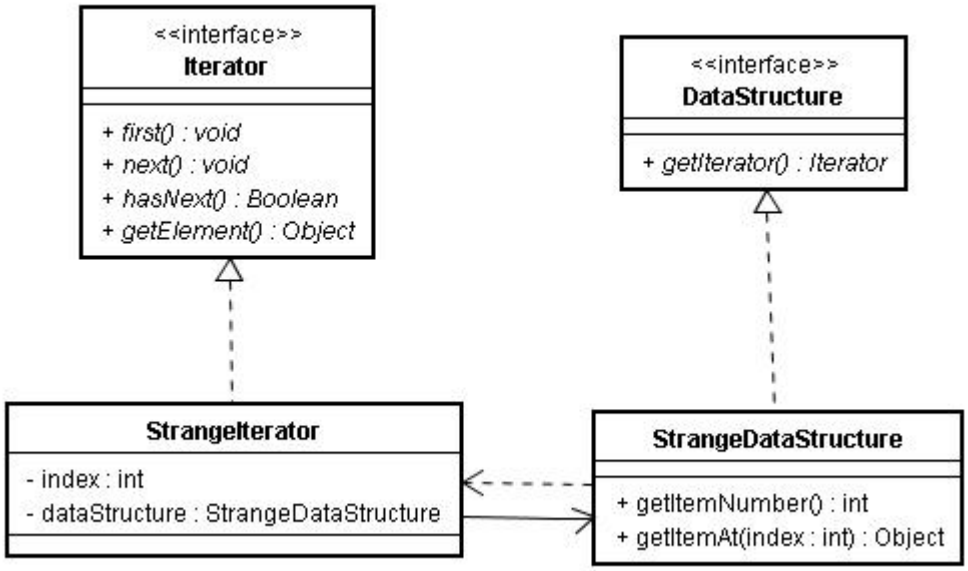
```
        function next():void
        function hasNext():Boolean
        function getElement():Object

    }
```

Also, we define the interface for data structure.

```
    public interface DataStructure
    {
        function getIterator():Iterator

    }
```

Now, we need to implement the concrete class of the data structure and the iterator. Actually, the concrete iterator knows the details of the specific data structure. And how to traverse the data is depends on the concrete iterator's implementation. See the code for more information, 'coz I don't want to put so much code here :)

Here is the class diagram.



In this pattern, we can traverse the data without considering the concrete data structure by using the same interface, further more, we can change the way of traversing the data by changing the concrete Iterator class.

Enjoy!

Prototype

When I want to write the Prototype pattern, I firstly look up the ActionScript 3.0 manual. I want to find out whether there is a clone method in Object class. If so, I will use this as an example. But, unfortunately, I can't find this method in the Object class. Then I found the prototype attribute, but the explanation confused me. So, I decide to show you this pattern in my own way without using the Object class.

Firstly, you need to know the intent of this pattern. You can read the following text.

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

--By GOF BOOK

This pattern is about how to instantiate a new object. You may say, we could use the new operator to instantiate the object. Of course, you can do it in this way. But there are still some another things you need to considering, such as the new object can't get any information from the current object. It just likes in some fantasy movies, the magician copy himself, and two or more magicians looks the same appears in the screen.

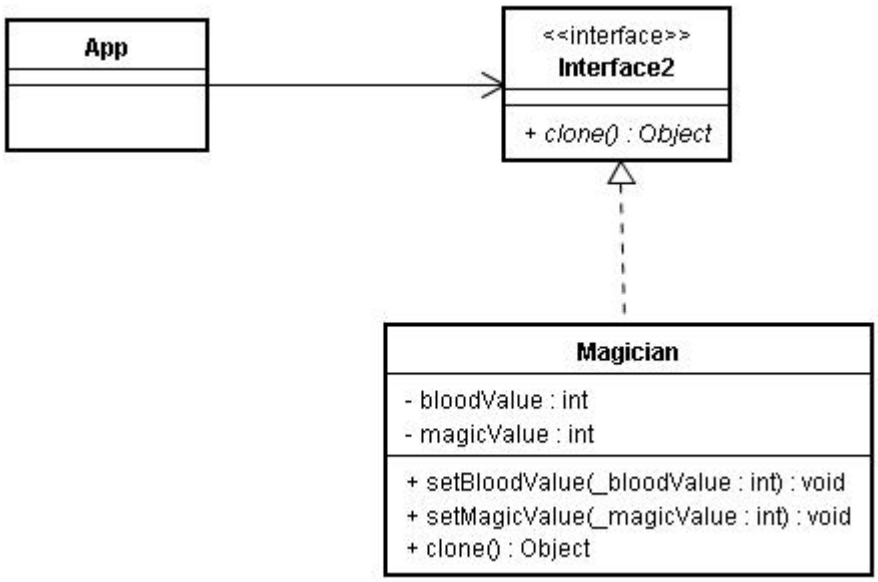
When the object has much the same with the existing object, you need to consider this pattern. By using this pattern, you can get a copy of the existing object.

Do it the simplest way, we provide an interface named Cloneable, and the concrete class will implement this interface.

```
public interface Cloneable
{
    function clone():Object

}
```

And the concrete class named Magician. It implements the clone method, and returns and new magician with the current state.

The class diagram is as follows.



Now, you can use Magician.clone() to get a new instance. And the new instance will be the same as the existing one. If you want the classes be the same, this is a good way for you.

When you want to apply this pattern, I should tell you some thing more about this pattern. Have you ever heard deep copy and shallow copy? If you have ever heard that, I think you're already what I will say next. If not, you'd better to get some information about those things.

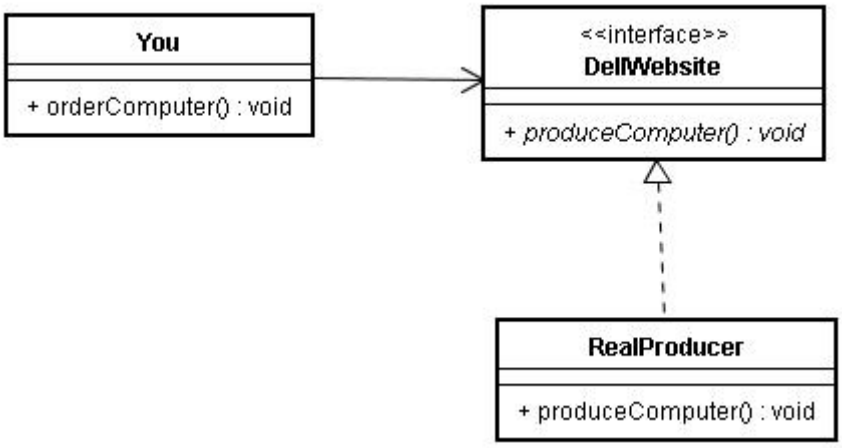
If you use the shallow copy, the reference field of new object and old object will points to the same object. So, when you change the object via the old object. The new object will also be affected. So, be careful about the shallow copy.

Enjoy!

Builder

Have you ever buy a computer online, maybe from dell? In dell's website, we just need to follow its process to order the accessories you need, then you can get your own computer configuration. Of course, you can't get the real computer until you pay it :)

Here, you direct the producer to produce your own computer through the dell website. Ok, three roles here, you, dell website and the real producer.



If we change the You to Director, DellWebsite to Builder and RealProducer to ConcreteBuilder, then this class diagram will be totally like that in a pattern called builder, and this is the topic today.

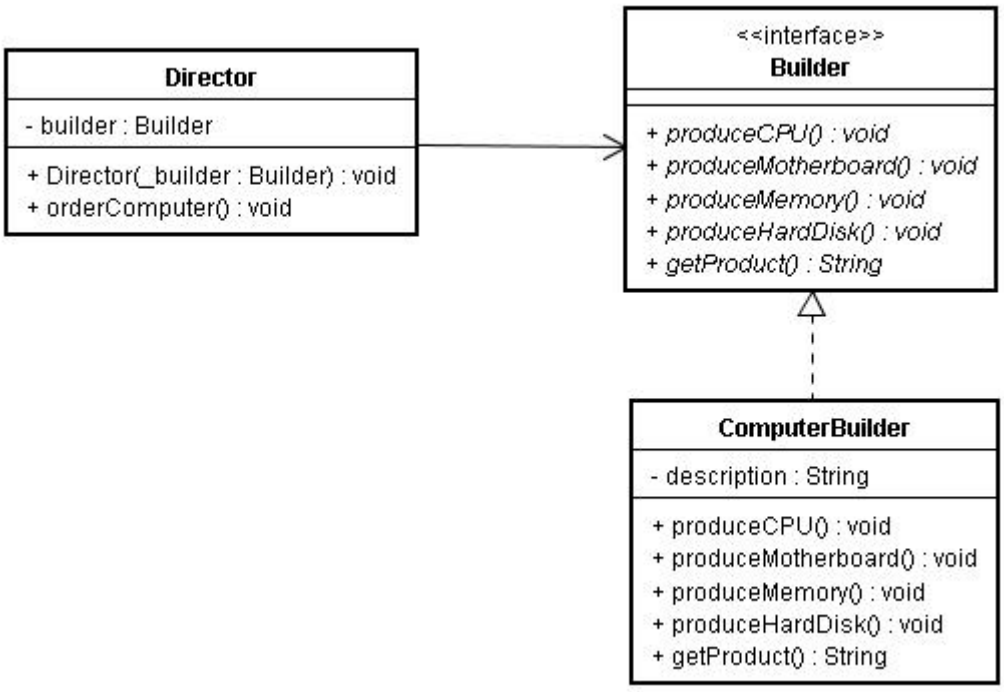
The intent of this pattern is as follows.

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

--By GOF BOOK

In our example, the computer is a complex object, and we use the accessories' description for its representation, and the construction process is the same.

So, our class diagram will be as follows.



Here, we use the description for the return value of getProduct() method in ComputerBuilder.

Now, the Director controls how to produce the computer in orderComputer() method, and the ComputerBuilder is the real action taker, it produces the real computer you want. If you want to get another kind of computer, you need to add another concrete builder class to build your dream computer :)

In our code, we can call produce the computer in this way.

```
var builder:Builder = new ComputerBuilder();
var director:Director = new Director(builder);

director.orderComputer();
```

Further more, you can compare this pattern with the template method. When you join the Director and the Builder interface together, it looks like the template method, isn't it?

Enjoy!

State

Yesterday, when I was on my way home, I suddenly met an old friend. I haven't met him since I began to write these articles. :) We stop at a little cafe, and began to talk.

"I've just changed my job", he said, "and now I join the Orange".

"WOW, Orange, you mean the biggest MP3 manufacturer company," I answered.

"Yeah, and I join the new product team, we want to surprise the whole world by our new product." He said proudly.

……. (The rest of our conversation is very boring, so, let's stop here)

When I back home, I can't help to imagine the new product, which may surprise the whole world. Actually, I don't think this product can give a surprise to me. From my old friend's description, I don't think this product can defeat another fruit company's product.

"This product will have nothing but just one button", it was the description of the new product. It sounds Orange goes further more than other fruit company. Just one button, it confuses me. How to control it just by one button? I don't have the answer. But it remains me something that I don't want to tell you now. Let's write some code to mimic this product now.

The code maybe look like this, we can give some rules to this product. When you press the button the first time, It plays music, and it will be paused by your second touch, the third touch will cause it to stop, and then, the first. So, we can get the code like below.

```
if(currentState == play)
    currentState = pause;
else if(currentState == pause)
    currentState = stop;
else if(currentState == stop)

        currentState = play;
```

Let's check our code carefully, eh, it works well. But, how can I start this machine or stop it? Aha, we have a new mp3 player now, but we can't even start it. I admire that it surprises me.

Eh, maybe for this kind of product, you can start the product just by press the button down with a fixed interval, and stop it in the same way. So, we can add these two rules to our code.

Another more question, how to add these two rules to our code, just by add another two more if-else? Of course, it makes the code looks ugly. Genius programmers, what will you do to refactor the code?

Aha, this is what I want to talk today. Firstly, let me show you a new pattern, named State. The intent is as follows.

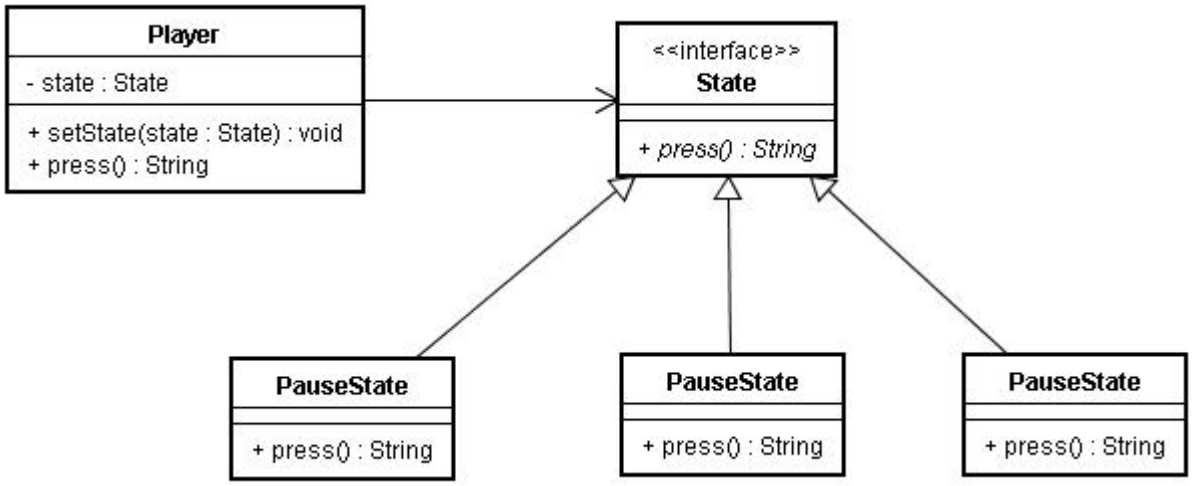
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

--By GOF BOOK

Now, let's begin.

Firstly, we need a super class to define some basic operation. Then, let all the other states inherit from the super class.

And the class diagram is as follows.



Does it look more clearly? If you want to add other rules, you just need to add the concrete state class; you don't have to maintain so much if-else statements.

Let's take a look at the super class, and a concrete state class.

```
public interface State
```

```
{
    function press(player:Player):String
}

public class PlayState implements State
{
    public function press(player:Player):String
    {
        player.setState(new PauseState());
        return "Playing :)";
    }
}
```

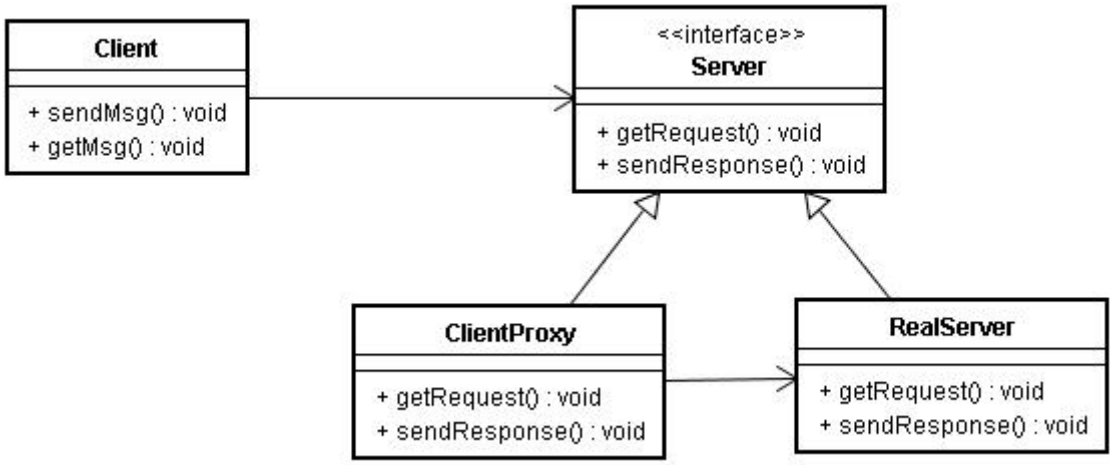
With this pattern, we leave the state-related operation to the concrete state class. And every concrete state class corresponds to a concrete state.

This pattern is very helpful when you modeling the objects that have state-related operations. You don't need to write so much if-else statements, and maintain the variable to record the current state. In fact, if-else block maybe causes some trouble, especially when the block becomes larger. So, you'd better use this pattern to construct you code.

Much for today, Enjoy!

Proxy

Have you ever use HTTP-proxy or some other proxy? When you're in a relatively isolated environment, such as the LAN in your company, maybe you'll need it. Actually, when I was an intern in an IT company, I always used the HTTP-proxy to login the MSN and surf on the internet. When I used MSN or surf on the net, I can't feel the existence of the proxy. And this is the role of a proxy. And this can be express as follows.



Actually, the client proxy is just like a middle layer between the client and the real server. And in this layer, the network administrator can do many things, such as, he can group people by using authorization mechanism that different group will get different service, or he can allow or forbid you to surf some sites. All those can be done in the proxy layer.

Further more, layer is an important concept in computer science; you can see it everywhere, such as the HAL (Hardware Abstraction Layer) in the OS, or the N-tier architecture website and so on. With layer, we can do many things, for example, we move all the platform-dependent properties in to the HAL, and then when we want to move the OS from one platform to the other, we just need to rewrite the HAL.

And here we use the proxy layer to solve some problems which happen during we access the subject directly, such as the security, the spending or something else.

The intent of the corresponding pattern in the GoF is as follows.

Provide a surrogate or placeholder for another object to control access to it.
-- By THE GoF BOOK

Now, the example time :)

We put the RealServer and the ClientProxy into a package named outer and put the Client into the default package. And then, change the modifier of the RealServer to default, and the ClientProxy to public, so that, the Client can't access the RealServer Directly.

If you try to do this, server = new RealServer("server") in the Client, then you'll get a complier error, it means the Client can't access the RealServer as you want.

Now, we can use the proxy to bridge the Client and the RealServer.

```
private var server:Server;
server = new ClientProxy(serverName);
server.getRequest();
server.sendResponse();
```

If the Client wants to visit the server, it can only through the proxy. So you can do what you want to control the visit :)

Enjoy!

Interpreter

In web programming, we often use regular expression for validating the e-mail address or phone number. Regular expression is a powerful tool in validating the specific format field. However, the interpretation of regular expression is not an easy job.

In the GoF's design patterns, there is a corresponding pattern named Interpreter.

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
--By GOF BOOK

Maybe, from the intent, you'll know that you won't touch this pattern in your future career :) That's what I think.

To illustrate this pattern, I'll show you a demo for calculating an arithmetic expression. And the expression will only support addition and subtraction, so the expression may looks like "11 + 2 + 3 - 4 – 5 + 8".

And the grammar we define as follows:

GeneralExpression => AddExpression | SubExpression | NumberExpression

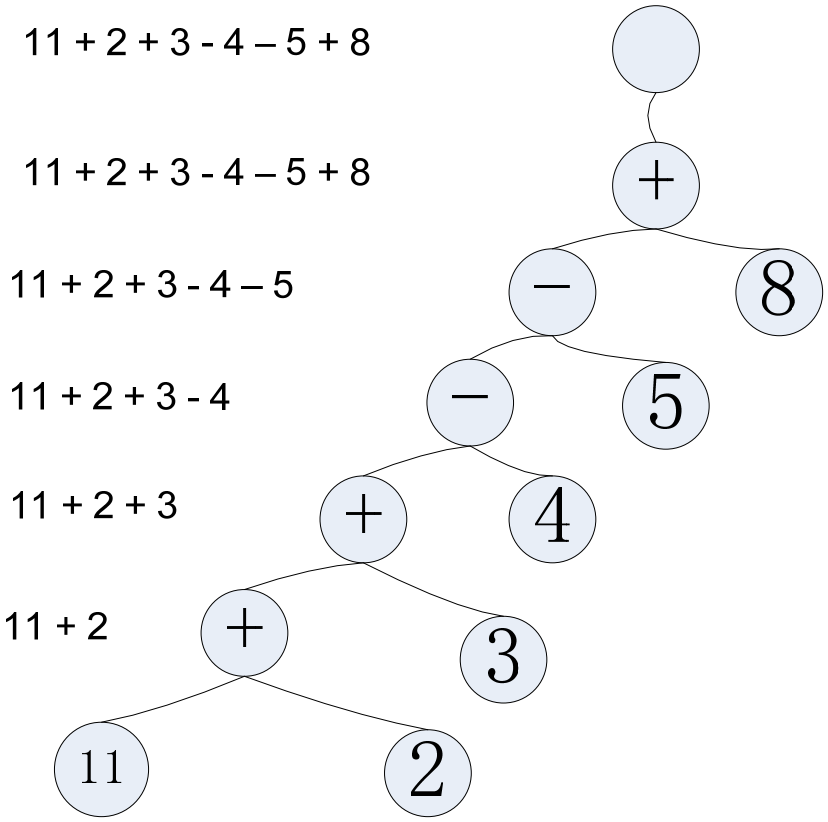
AddExpression => GeneralExpression + NumberExpression

SubExpression => GeneralExpression - NumberExpression

NumberExpression => 0 | ([1-9][0-9]*)

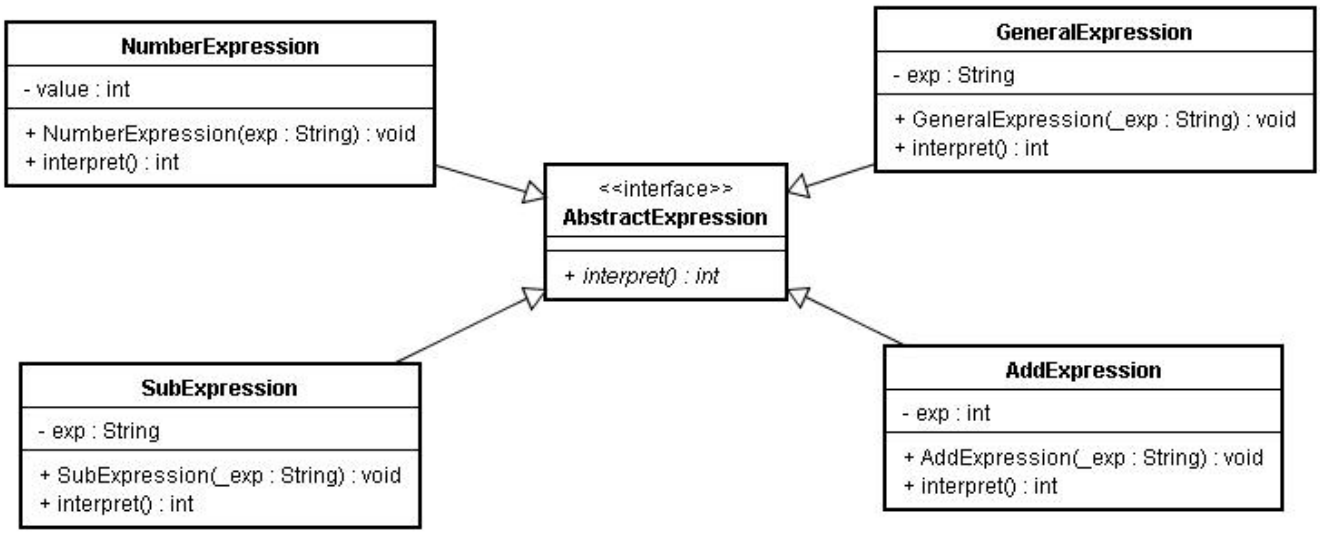
Note: "|" means or.

Now, the expression "11 + 2 + 3 - 4 – 5 + 8"'s grammar tree can be describe as follows.



As you see, we can calculate the expression by parsing the expression from top to down. Now, we need to design the classes.

The class diagram is as follows.



When we get an expression, it'll be passed to the GeneralExpression, then the interpret method will be called. The source code is as follows.

```
public function interpret():int
{
    if(exp.lastIndexOf("+") > exp.lastIndexOf("-"))
        return new AddExpression(exp).interpret();
    else if(exp.lastIndexOf("+") < exp.lastIndexOf("-"))
        return new SubExpression(exp).interpret();
    else
        return new NumberExpression(exp).interpret();

}
```

And the interpret method of SubExpression or AddExpression maybe called, and here is the source code of interpret method in SubExpression.

```
public function interpret():int
{
    var index:int = exp.lastIndexOf(SUB);
    var generalExp:String = exp.substr(0,index);
    var numberExp:String = exp.substr(index+1,exp.length);
    return new GeneralExpression(generalExp).interpret()
                - new NumberExpression(numberExp).interpret();

}
```

In the interpret method of each class will parsing the expression from top to down, then calculate the value and returns it.

As you see, this pattern is, eh, all about the complier :) I don't like this pattern, because it's uneasy to implement when the grammar is not so simple. Maybe it's all because I haven't learned the complier principle well.

Enjoy!

Memento

Now, I'm using Microsoft word to write these articles. These word processor programs are very useful when you writing something down. Sometimes, we heard someone was very familiar with these programs, such as word, can remember many short-cuts. Though I use this program frequently, I can't remember many short-cuts. I can remember some short-cuts, such as "Ctrl+Z", which means UNDO.

The UNDO command is widely use in today's program. You can see it almost everywhere. Besides the word processor program, you can find it in photoshop, in flash, or even in games. Eh, in games, this command isn't called UNDO, it use another name SAVE.

I think you should be familiar with this command. But, have you ever implemented this command in your application? Maybe, your application doesn't need this kind of command, but you won't deny the importance of this command. So, it's worth to consider how to implement this command.

In the GoF book, there is a pattern, which has much to do with this command, called Memento. The intent is as follows.

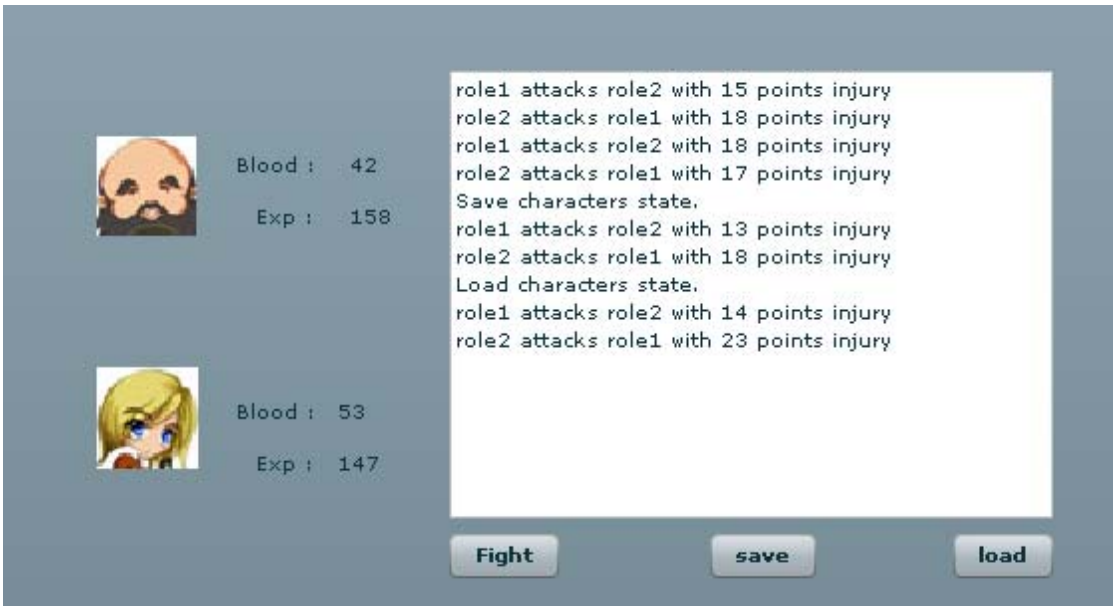
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

--By GOF BOOK

In my opinion, without practice, the definition is nothing. So, let's write some code to implement this pattern.

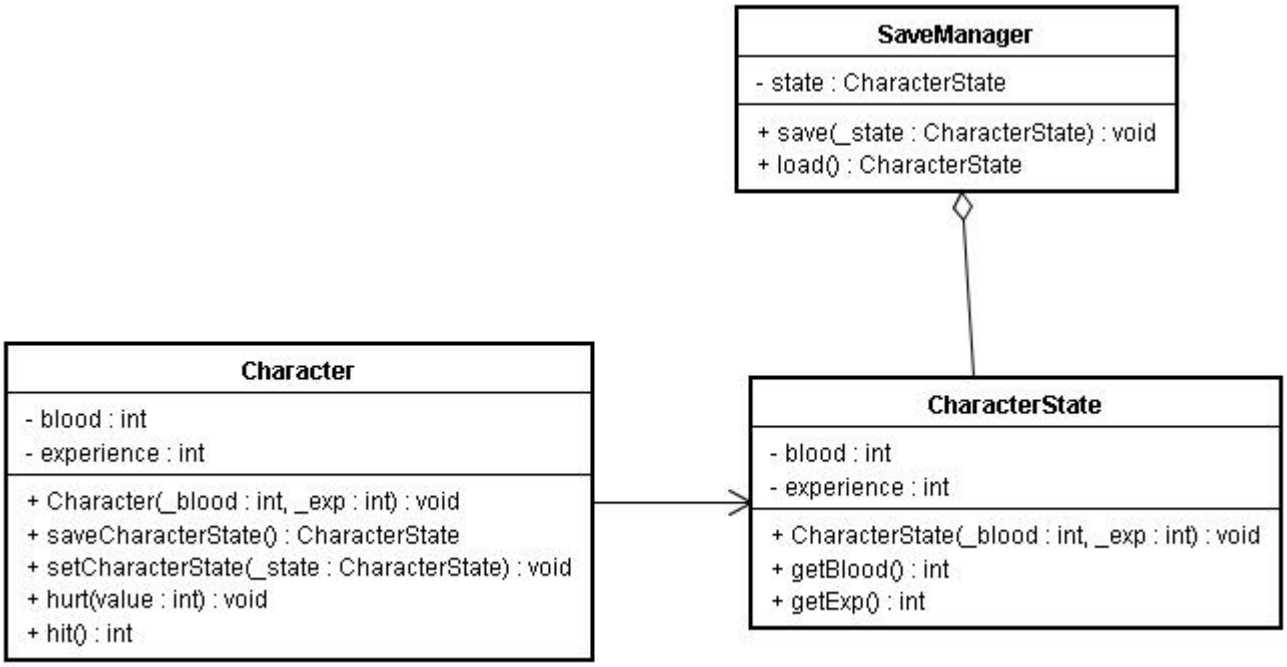
This little program will mimic the game command SAVA. Generally, we need to store some basic information about the role, such as the blood, the experience and so on. Do it the easy way, we will only store the blood and the experience, nothing more.

Our demo will be looks like follows.



This is a little game with no rules, just fight :) And you can save the characters state, including blood and experience, with the save button. Of course, you can load the states you've just saved. One more thing, the experience here can't help you level up; you can take a look at the source code for its effect.

The class diagram is as follows.



The SaveManager class is used for manage the save state, and all the state of Character will be put into the CharacterState. When we want to save the character's state, we just need to call the

SaveManager.save, and pass the character's state into it. Then, when we want to load it, call the load method of SaveManager.

This is a basic application of this pattern. If you want to implement the UNDO and REDO commands, you may need a memento stack. How to use the stack depends on you program.

Enjoy!

Visitor

It's our winter holiday now, and I spend many times in playing games. In last week, I soaked myself in <Prince of Persia: the sands of time>. I love this game, but it's a pity for me that when I fight to the monsters, I can't control the heroine. Because of that sometimes I need her help to shot somebody exactly not the other one. OK, I'm crazy :) Controlling two roles will increase the difficulty of manipulation, and it's not a RTS game.

In RTS game, such as Warcraft, you can control many units. When your team attacks someone, the team members will use their own skills. Maybe the Dwarven Sniper will use his gun to shot, while the Mountain King uses his hammer. So, in action script, we may express these in this way.

```
Var team : Array = new Array();
team.push(new DwarvenSniper());
team.push(new MountainKing());
team.push(new Priest());

function attack(team:Array):void
{
    for(var i:int = 0; i < team.length; i++)
    {
            If (team[i] instanceof DwarvenSniper)
                DwarvenSniper(Team[i]).gunShot();
            Else if(team[i] instanceof MountainKing)
                MountainKing(Team[i]).hammerShot();
            Else if(team[i] instanceof Priest)
                Priest(Team[i]).priestHit();
    }
}
```

Note: the above code is directly type in Word, so don't try to complier it, it may contain many grammar mistakes :)

Now, take a look at the attack function. When you pass the team array into it, it may works well. Actually, the team member maybe more than ten, eh, I mean the type of members. So, we need to distinguish them in the iteration. This will cause many if-else. And this is the "bad smell" of our code :)

Here, our problem is that, in an array that may contains many objects, and the action it takes depends on the object type. Further more, the object type is fixed, and the operations of each type are also known, just as the units in Warcraft. What we want to do is organize their basic operations to form a series operation, such as a team in Warcraft to attack the creatures with their normal skills, or attack the buildings with siege skills.

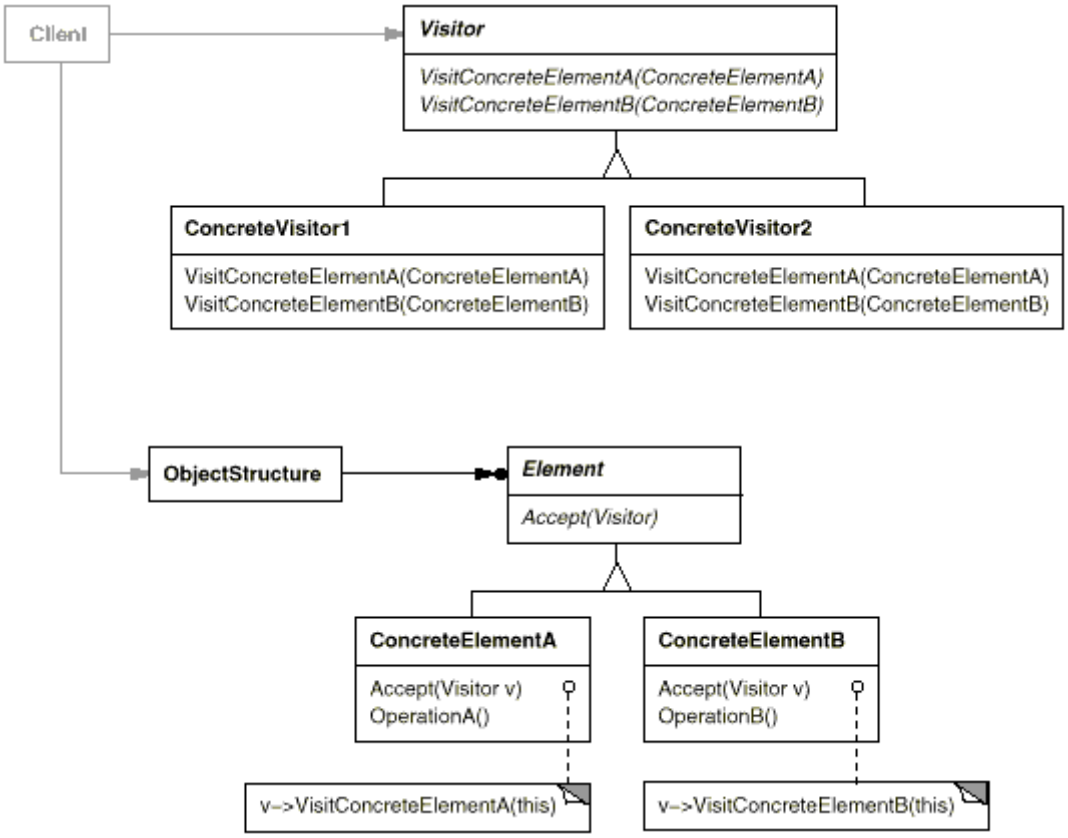
```
Function creatureAttack(team:Array):void
{
    If(……)
        //Using it's skill here
    Else if(……)
    .
    .
    .
    .
    Else if(……..)
        //Using it's own skill here
}
Function buildingAttack(team:Array):void
{
    …..//the same if-else clauses with different skills
}
```

Actually, our aim is to refactoring the if-else clauses. And that's what the Visitor pattern does. The intent is as follows.

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

--By GOF BOOK

And here is the static diagram of this pattern from the GoF book.



In the Warcraft example, the member is corresponding to the Element, eh, one for each concrete element. And the array is the ObjectStructure. The attacks is the visitor, the concrete visitor is the creature attack and building attack.

So, we can refactor the code by this pattern, let all the members implements the Element interface, and let the attacks implements the Visitor interface. And in the iteration, we can do it in this way.

```
Var creatureAttack: AttackVisitor = new creatureAttack();
For(var i:int = 0; I < team.length; i++)
    (teamElement)team[i].accept(creatureAttack);
```

Here, one accept method replace all the if-else clauses and the concrete operations. Eh, you should implement every accept method in the concrete element like this.

```
Function accept(visitor:AttackVisitor):void
{
    Visitor.visitElementX(this);
}
```

If you’re interested in this pattern, find more information by Google :) And you can take a look at the example code in the attach file.

Flyweight

In Action Script 3.0 we have the following ways to define a String.

```
var str1:String = new String("foo");
var str2:String = "foo";
var str3:String = String("foo");
```

I don't know which way is your way, but they work the same way. Actually, all of the three String variables point to the same String object in memory. That's to say there is only one copy of "foo" object in the memory, but three references. In other languages, such as java, are almost the same. It saves the memory. This is called object pool. And there is a similar pattern called Flyweight.

Let's take a look at its intent.

> Use sharing to support large numbers of fine-grained objects efficiently.
> -- By THE GOF BOOK

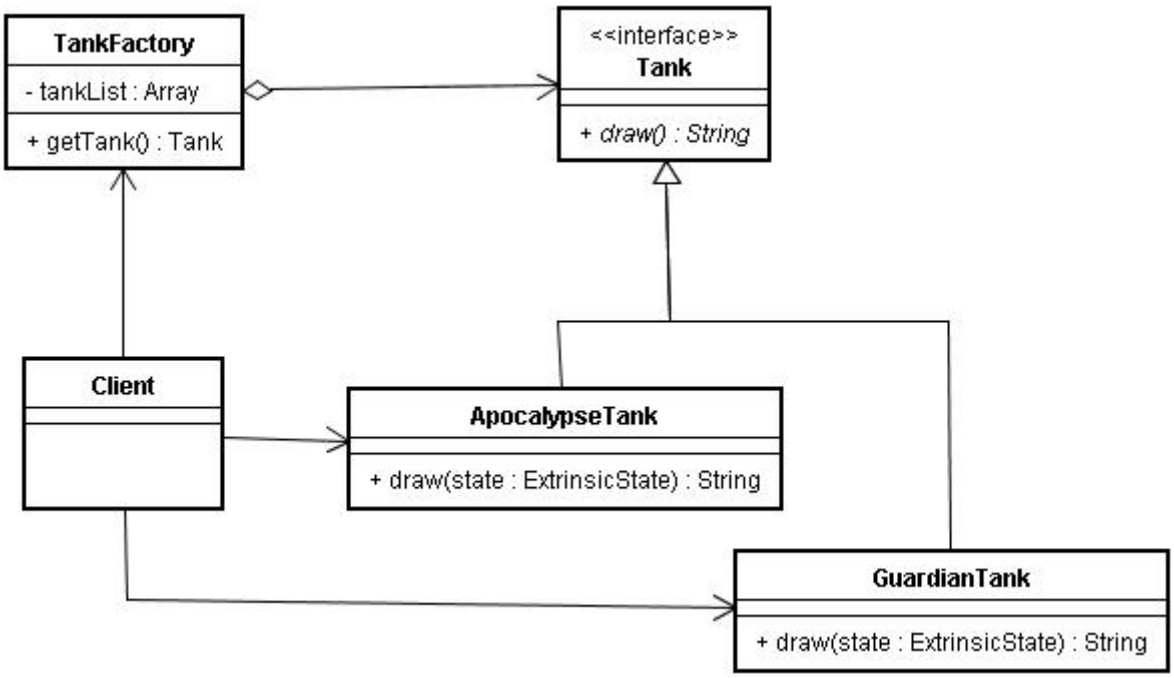
Do you remember the Red Alert, when I played this game with my friends; we often made many tanks, and just one or two types.

If you're the designer of the Red Alert, what will you do? Write a tank class, and let the concrete tank inherit it. Then when the player made one, get a corresponding object in the memory. This is a solution, but maybe not a good one, because it will take many memories and the memory is a scarce resource, so, we need to fix it.

The concrete tank shares the same model. The difference is just the coordinate and the direction. If we abstract these attributes, then a kind of concrete tank can share the same object, which just including the model.

So, we create a ExtrinsicState class for the extrinsic attributes, and create a Tank interface, and let the concrete tank implements it. The concrete tank class will only have the intrinsic attributes. Note that some operations will need the extrinsic state.

So, the class diagram will be as follows.



And in the tank factory, we produce the concrete tank, but not always create the concrete tank; you can see the following code.

```
public static function getTank(key:String):Tank
{
    if(tankList[key] == null)
    {
        if(key == "Guardian")
            tankList[key] = new GuardianTank();
        else if(key == "Apocalypse")
            tankList[key] = new ApocalypseTank();
    }
    return Tank(tankList[key]);
}
```

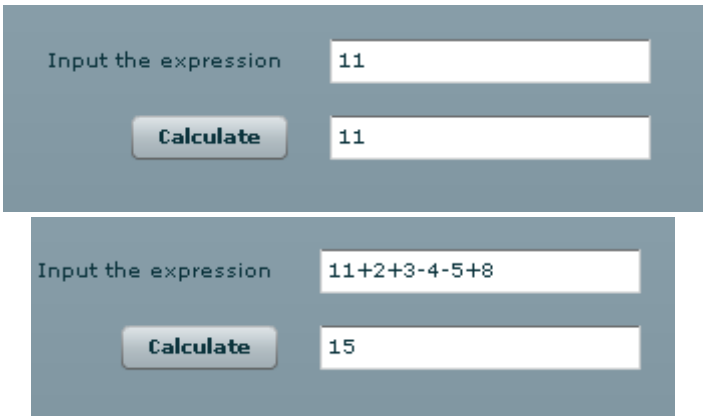
We use a tankList to hold the objects we have created, so, when the user wants a tank that we've already created, just return it, not need to create again.

Of course, when we do some operations, we may need the extrinsic state, so, pass the extrinsic state as a parameter, and then we get different appearance of each tank.

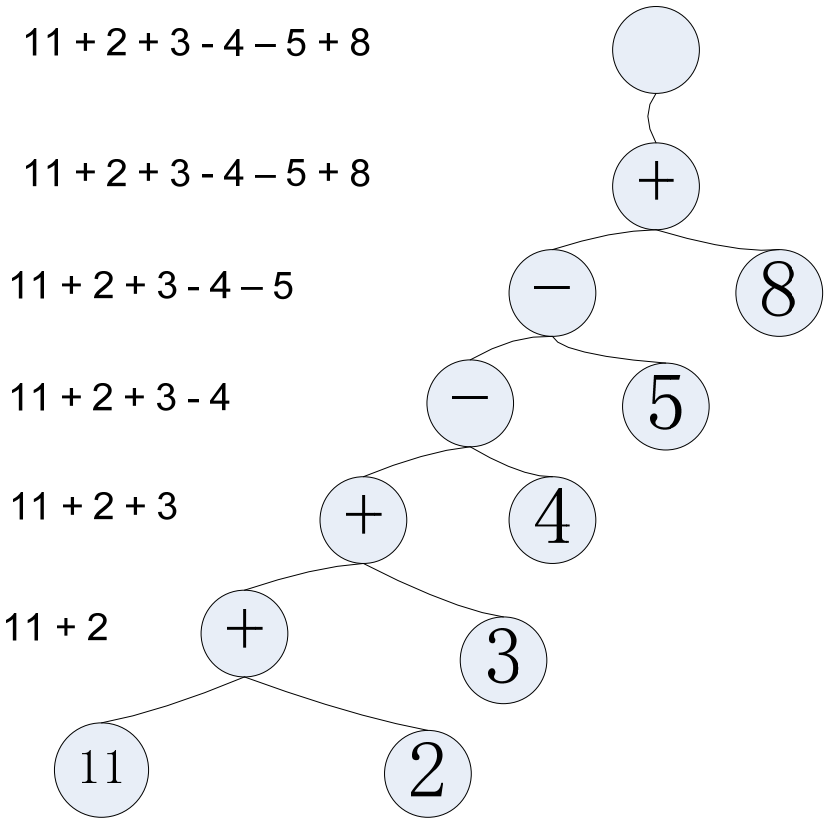
Enjoy!

Composite

Still remember the Interpreter example? No matter what you type is a number or an expression; it can calculate the result just by the clause" new GeneralExpression(inputText.text).interpret()"



It means that we treat the number the same with the expression. Actually, the expression is form by the numbers. So, you can take the expression as a container, it contains the numbers, and the number is a special kind of expression, it calculates itself.



It this picture, you can consider it as a tree. And all the inner nodes are containers, and the leaf node is the Number. We treat all the nodes in the same way, just use the interpret() method. So, the client is very convenient to deal with the expressions. This is what we introduce today, the composite pattern.

The intent of this pattern is as follows.

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

--By GOF BOOK

In the interpreter example, we put the differences of dealing with the inner node and leaf node into the concrete class. And leave a common interface for the client to deal with. This is what the composite pattern does.

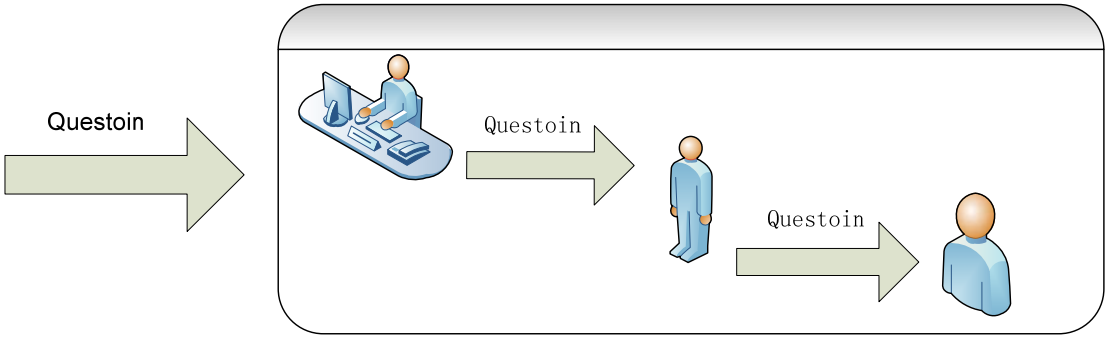
This pattern is very useful, especially when you building a user interface. You can treat display object and display container object the same way, without considering the differences between the two objects. It's very common in the GUI building environment.

Of course, it has its own disadvantages. For example, you need to carefully avoid two or more composites to form a cycle. Remember, be a tree, not a cycle!

Enjoy!

Chain of Responsibility

When you need some help in a hotel, you may ask the attendant firstly, if the attend can't help you, then the attendant may pass you question to the assistant manager, if the attendant still can't help you, then the question maybe pass to the lobby manager.



The picture above shows the way your question passes through. In fact, you don't need to care the path your question pass through; you just need to know, some one in the chain will give you an answer. And it has a corresponding pattern called Chain of Responsibility.

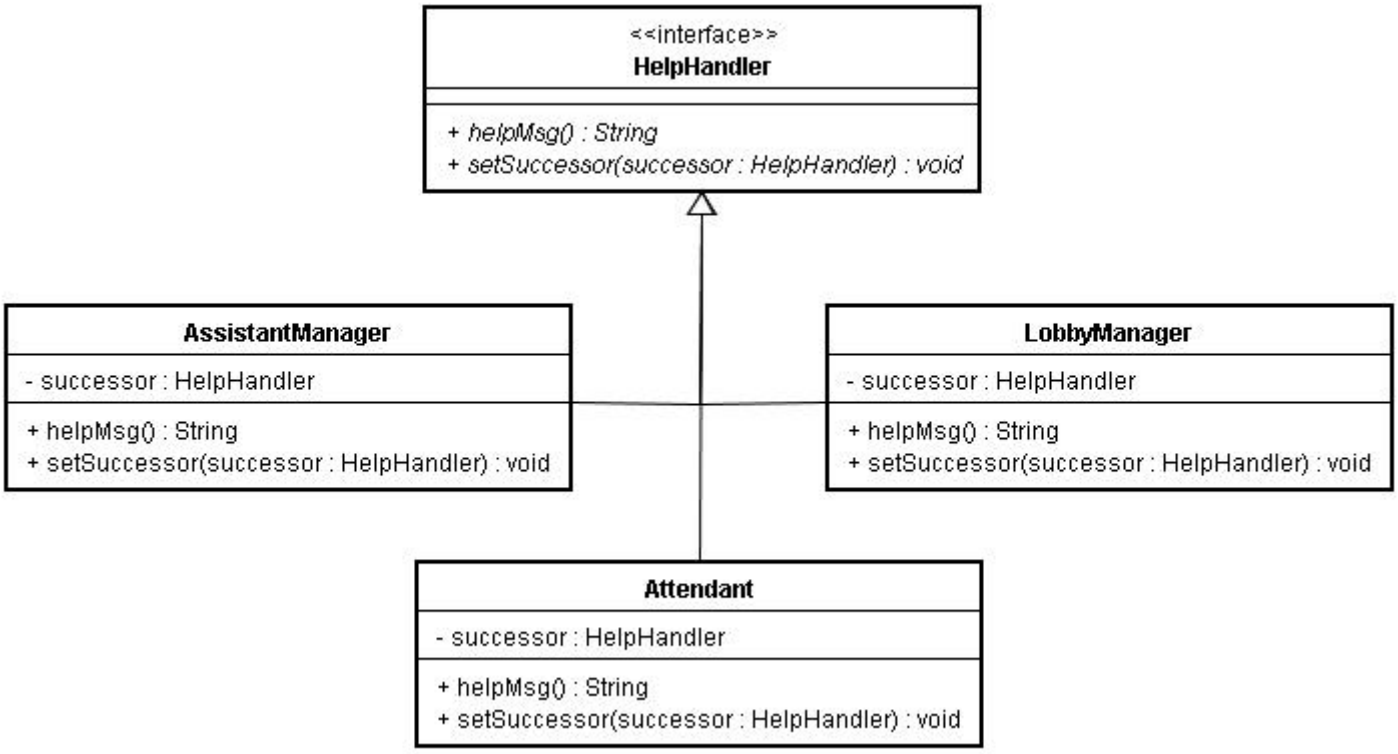
The intent of this pattern is as below.

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

-- By THE GOF BOOK

In the scene above, we may need three classes to handle the help message; further more, we need a common interface for these classes.

The class diagram will be as follows.



And if you want to use these classes, you need to specify each successor just as below.

```
attendant = new Attendant();
assistantManager = new AssistantManager();
lobbyManager = new LobbyManager();
attendant.setSuccessor(assistantManager);
assistantManager.setSuccessor(lobbyManager);
```

When a client ask a question, you can simply call the attendant.helpMsg(), and in the helpMsg() method of each concrete helper, it can decide help or not depends on it's own state. Eh, here I simply use the random number to decide pass the request or handle it. If you want to change the sequence, just change the successor by calling the setSuccessor() method.

The above introduction is a use of this pattern; someone said that it's pure CoR (Chain of Responsibility). There is another use of this pattern, eh, it's not pure CoR. In the not pure way, all the classes in the chain will work together to finish a job. Such as the pipeline, you can let each class take one step, then pass the request from the first one till the last one, then all the steps will finish, so the job is done. That's all for this pattern.

Command

A few months ago, I was an intern of a company. I joined a group which builds a LBS application, and our target platform including J2ME and Android. I was asked to make a J2ME demo.

This demo was just like the Google map on J2ME. Eh, actually, we did something more than that; but you don't need to know, just considering we're going to build a Google map on J2ME platform.

As our project going, we can fetched the map tile and display it on the screen. When the user pressed the direction key, we created a new connection to fetch the map tile, and then display it on the screen. The code is maybe as follows. I hope you can understand the following code :)

```
Var directionBtn:Button = new Button();

directionBtn.addMouseEventListener(Mouse.Click,
"downloadTile");

function downloadTile():void

{

    Var conn:Connection =
ConnectionFactory.getConnection();

    DrawOnScreen(conn);

}
```

Note: the above code is pseudo code. Don't try to complier it.

If you ever use some similar application on your phone, you may realize where the problem is. Let me figure it out. When the user wants to drag the map, the application will seems to be as no response. It means, when the user holds a direction, the application will produce many connections, and only 5 connections can be executed at the same time, and each connection may be finished in 3 or 5 seconds. (Note: 5 connections at one time are depending on our test, and each connection's time spending is depends on the network.)

So, our problem is that, when the user drags the map, the application will download all the tiles. Actually, not all the tiles are needed, the tiles the user wants is where the user drags to not the way it passes.

Wow, genius, maybe you have your own solution now. And our solution is use a stack to hold the commands. That is, when the user presses a direction key, it puts a common into the stack, and we execute the commands from the top one. So, the most recent request will execute first. And now, our code will be as below.

```
directionBtn.addMouseEventListener(Mouse.Click,
"downloadTile");

function downloadTile():void

{

    requestStack.push(new Request());

}
```

The most difference between the two is that we put a method into an object. And this is the Command pattern.

See the intent now :)

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

-- By THE GOF BOOK

With this pattern, you can make a request queue, or implement a logger system or the UNDO / REDO command.

If you want, you can take a look at the example code :)

Enjoy!

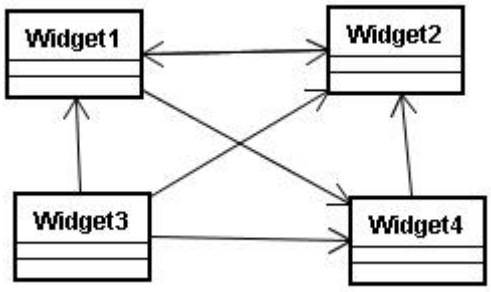
Mediator

Ok, the last pattern now. Let's take a look the intent directly.

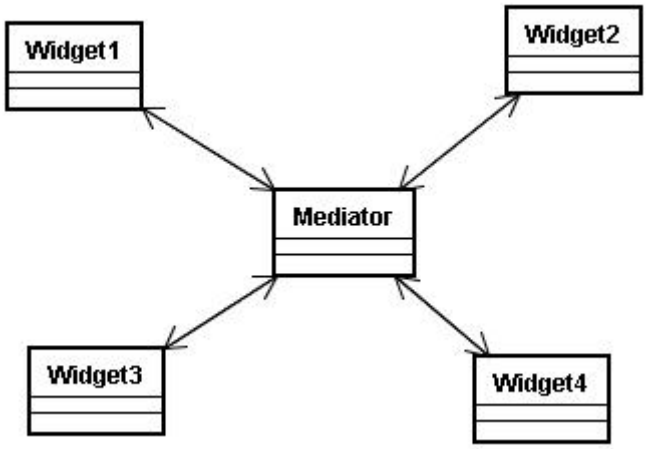
> Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
>
> -- By THE GOF BOOK

From this intent, we can get that this pattern is use for encapsulating the interaction of objects.

For example, when you build a GUI application, you may enable or disable the button when the user input something, you may add a listener for the text change event. Then, when the event happens, you change the state of that button. It means that you must know the button object, and some details about the button. As you application going further, you may do many things in that function, also, you need to know many classes' detail. When the requirement changed, you need to fix many things. And your classes maybe look like as follows.



With the help of Mediator pattern, you reduce the complexity of each widget, they don't even need to know the existence of other widgets, and they only need to know the mediator. When an object changes, it tells the mediator, then the mediator notify the other objects. In this pattern, you put the complexity from the widget to the mediator. The mediator needs to know everything. And the classes changes to below.



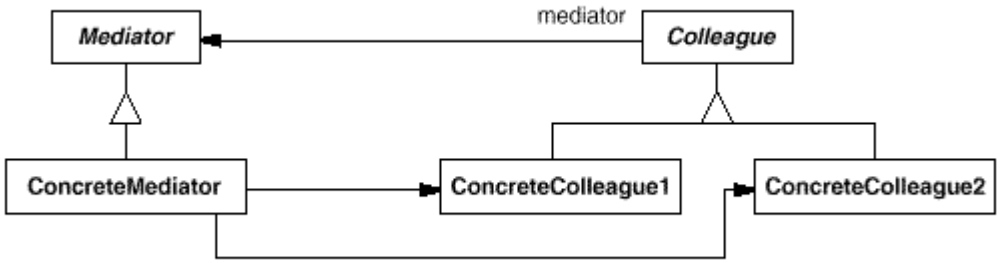
In GoF, all the widgets will inherit a super class Colleague, and the widgets is called concrete colleague. When the widget's state changes, it calls the widgetChanged() method. My simple implementation of colleague is as follows.

```
public class Colleague
{
    private var mediator:Mediator;
    public function Colleague(mediator:Mediator)
    {
        this.mediator = mediator;
    }
    public function hasChanged():String
    {
        return mediator.detectChanged(this);

    }

}
```

In the concrete mediator, we need to distinguish which objects call the method, and here is my implementation.

```
public function detectChanged(colleague:Colleague):String
{
    if(colleague == colleague1)
        return colleague2.operation2();
    else if(colleague == colleague2)
        return colleague1.operation1();
    return null;

}
```

And the class diagram below is from the GoF.



Hope this helps. That's all for this pattern.

Thanks for you kindness in reading those articles, especially for those who make comments. Actually, I'm a fresh man in design patterns; the main reason that I wrote those articles is to help me learn more quickly. If there is any wrong that troubles you, I've to say sorry here, sincerely! If you have any questions, contact me, liubo.cs@hotmail.com.

Final Note

In this series, we talk about the 23 design patterns. We use some examples to illustrate how and when to use the patterns, but there is still something I've to say here.

Why design patterns?

In my view, design pattern is just a solution to some types of problems, and just a general solution. We use this pattern is just because this pattern is proved to be useful or efficiency to this kind of problem. In general, it's a better solution, maybe not the best one, but it's enough in most cases.

With these patterns, you can communicate with your colleagues more easily during the development. You just need to say the pattern name, such as "Singleton", and your colleagues will understand, "Oh, it's a singleton, I can't use the new operator." You don't need to explain to your colleagues what the pattern it is.

How and when to use the patterns?

It depends on your experience and what the problem is. Understanding the problem is the first step and the most important step when you build an application.

The following is the conclusion from the GoF book, maybe it helps you.

Creational Patterns

Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

http://ntt.cc/2008/10/19/gang-of-four-gof-design-patterns-in-actionscript-abstract-factory.html

Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

http://ntt.cc/2009/01/22/gang-of-four-gof-design-patterns-in-actionscript-builder.html

Factory Method

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

http://ntt.cc/2008/10/08/gang-of-four-gof-design-patterns-in-actionscript-factory-method.html

Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

http://ntt.cc/2009/01/21/gang-of-four-gof-design-patterns-in-actionscript-prototype.html

Singleton

Ensure a class only has one instance, and provide a global point of access to it.

http://ntt.cc/2009/01/12/gang-of-four-gof-design-patterns-in-actionscript-singleton.html

Structural Patterns

Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

http://ntt.cc/2008/10/28/gang-of-four-gof-design-patterns-in-actionscript-adapter.html

Bridge

Decouple an abstraction from its implementation so that the two can vary independently.

http://ntt.cc/2009/01/11/gang-of-four-gof-design-patterns-in-actionscript-bridge.html

Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

http://ntt.cc/2009/01/30/gang-of-four-gof-design-patterns-in-actionscript-composite.html

Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

http://ntt.cc/2008/11/01/gang-of-four-gof-design-patterns-in-actionscript-decorator.html

Facade

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

http://ntt.cc/2009/01/10/gang-of-four-gof-design-patterns-in-actionscript-facade.html

Flyweight

Use sharing to support large numbers of fine-grained objects efficiently.

http://ntt.cc/2009/02/05/gang-of-four-gof-design-patterns-in-actionscript-flyweight.html

Proxy

Provide a surrogate or placeholder for another object to control access to it.

http://ntt.cc/2009/01/25/gang-of-four-gof-design-patterns-in-actionscript-proxy.html

Behavioral Patterns

Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

http://ntt.cc/2009/02/06/gang-of-four-gof-design-patterns-in-actionscript-chain-of-responsibility.html

Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

http://ntt.cc/2009/02/09/gang-of-four-gof-design-patterns-in-actionscript-command.html

Interpreter

Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

http://ntt.cc/2009/01/27/gang-of-four-gof-design-patterns-in-actionscript-interpreter.html

Iterator

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

http://ntt.cc/2009/01/18/gang-of-four-gof-design-patterns-in-actionscript-iterator.html

Mediator

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

http://ntt.cc/2009/02/10/gang-of-four-gof-design-patterns-in-actionscript-mediator.html

Memento

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

http://ntt.cc/2009/02/01/gang-of-four-gof-design-patterns-in-actionscript-memento.html

Observer

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

http://ntt.cc/2009/01/13/gang-of-four-gof-design-patterns-in-actionscript-observer.html

State

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

http://ntt.cc/2009/01/25/gang-of-four-gof-design-patterns-in-actionscript-state.html

Strategy

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

http://ntt.cc/2008/10/07/gang-of-four-gof-design-patterns-in-actionscript-strategy.html

Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

http://ntt.cc/2009/01/16/gang-of-four-gof-design-patterns-in-actionscript-template-method.html

Visitor

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

http://ntt.cc/2009/02/03/gang-of-four-gof-design-patterns-in-actionscript-visitor.html