Course: Soft Design

LECTURE 06

Proxy and **Bridge**

Software Development Principle 73~86

Proxy (代理模式)

Intent

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

Also known as

Surrogate

Motivation

Use an object in place of the real function object, and Zuze instantiates the replacement object when needed.

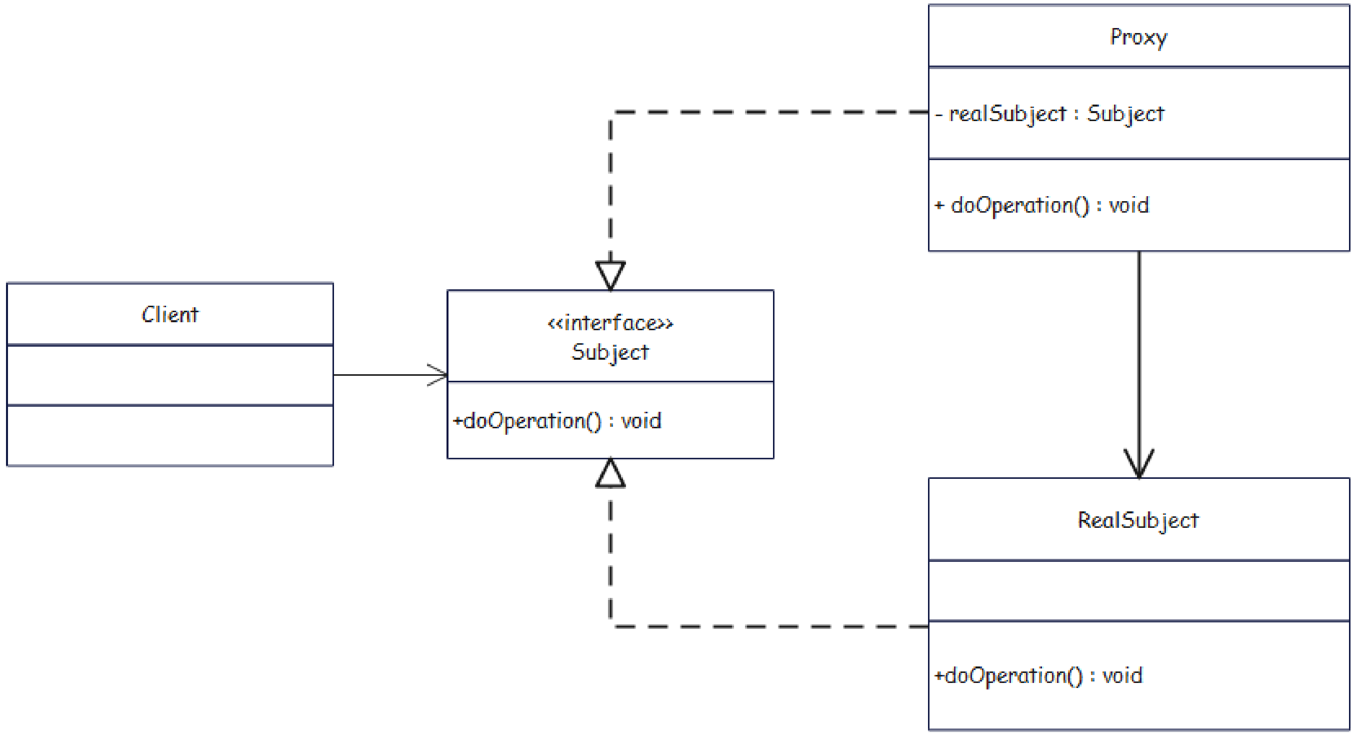
Course: Soft Design

Applicability

Here are several common situations in which the Proxy pattern is applicable:

- A **remote proxy** provides a local representative for an object in a different address space.
- A **virtual proxy** creates expensive objects on demand.
- A **protection proxy** controls access to the original object.

* Class Diagram



Participants

Proxy (ImageProxy)

- maintains a reference that lets the proxy access the real subject.
- provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
- controls access to the real subject and may be responsible for creating and deleting it.

Subject (Graphic)

defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

RealSubject (Image)

defines the real object that the proxy represents.

Collaborations

Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

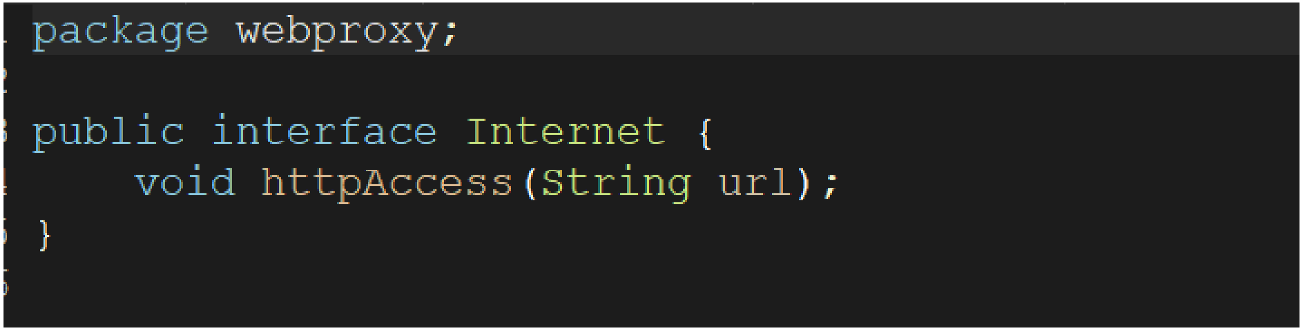
Consequences

The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:

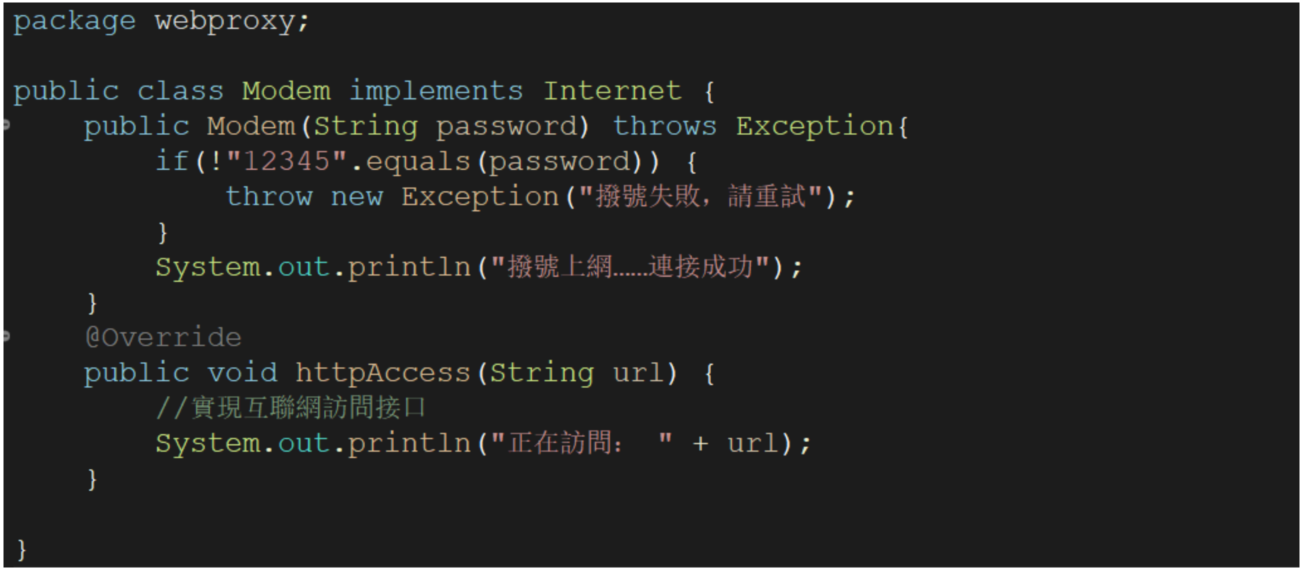
a) A remote proxy can hide the fact that an object resides in a different address space.
b) A virtual proxy can perform optimizations such as creating an object on demand.
c) Protection proxies allow additional housekeeping tasks when an object is accessed.

Course: Soft Design

Course: Soft Design

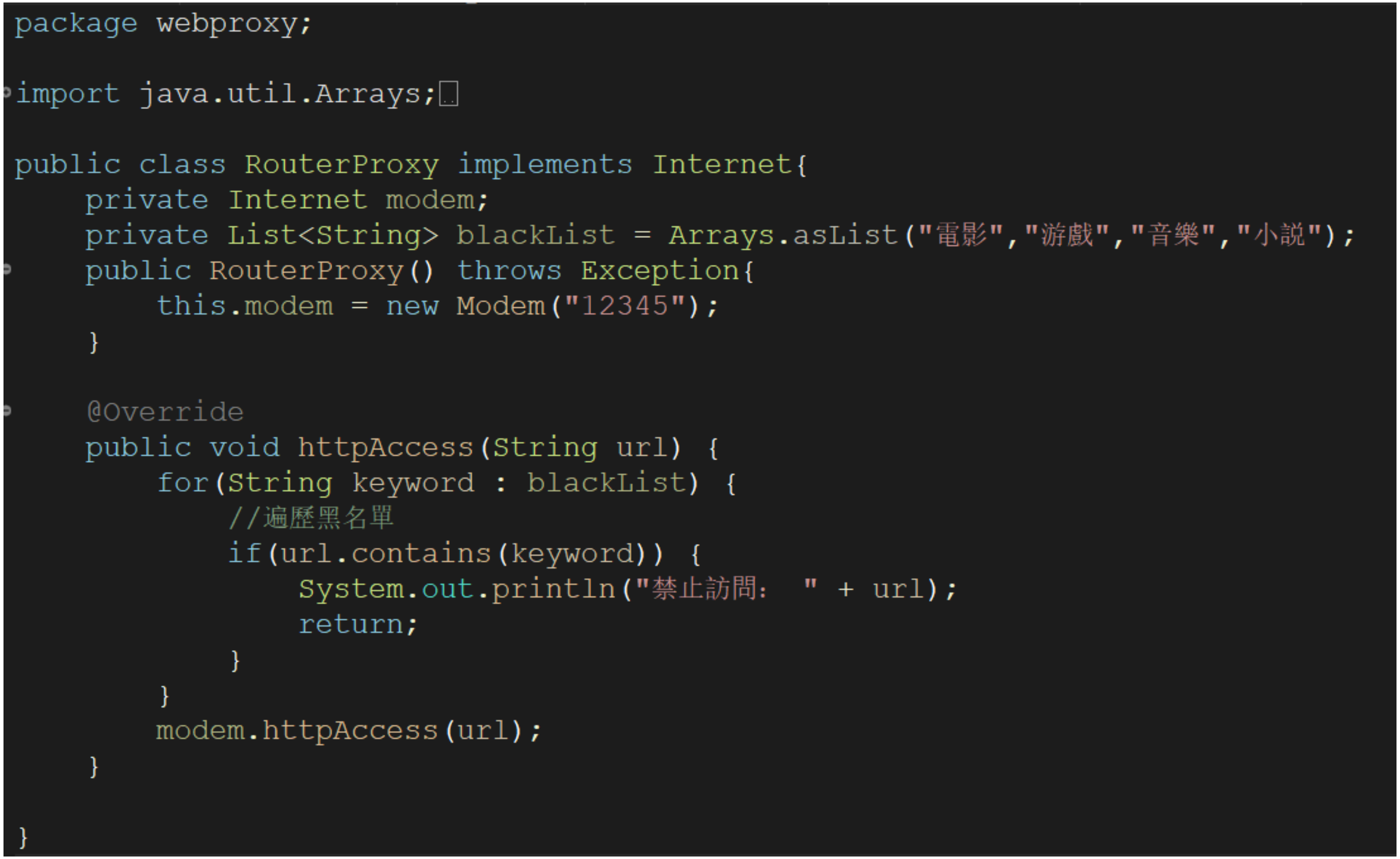


```java
package webproxy;

public interface Internet {
      void httpAccess(String url);
 }
```



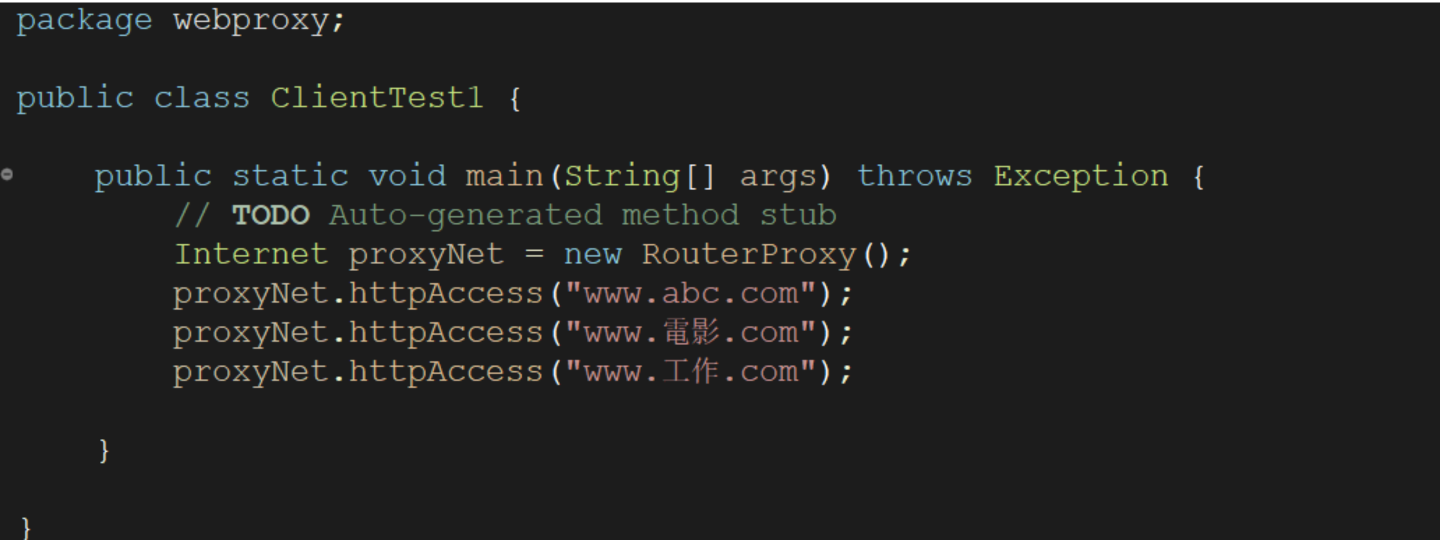
```java
package webproxy;

public class Modem implements Internet {
    public Modem(String password) throws Exception{
        if(!"12345".equals(password)) {
            throw new Exception("撥號失敗，請重試");
        }
        System.out.println("撥號上網......連接成功");
    }
    @Override
    public void httpAccess(String url) {
        //實現互聯網訪問接口
        System.out.println("正在訪問： " + url);
    }

}
```

Course: Soft Design

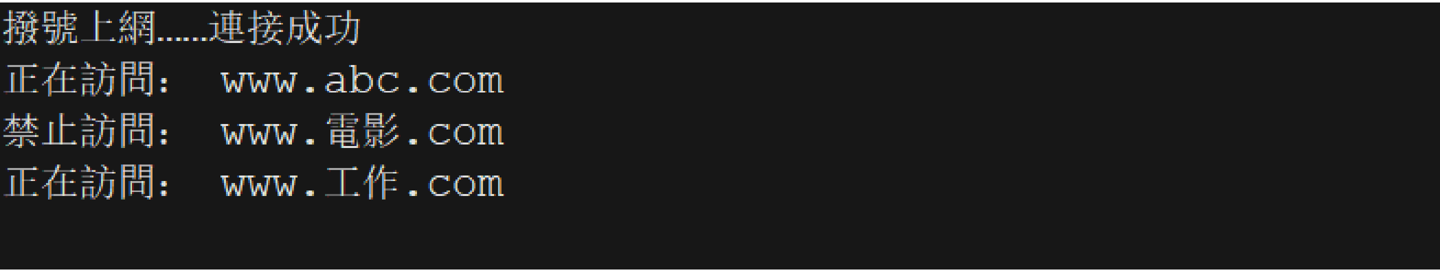


```java
package webproxy;

import java.util.Arrays;

public class RouterProxy implements Internet{
    private Internet modem;
    private List<String> blackList = Arrays.asList("電影","游戲","音樂","小說");
    public RouterProxy() throws Exception{
        this.modem = new Modem("12345");
    }

    @Override
    public void httpAccess(String url) {
        for(String keyword : blackList) {
            //遍歷黑名單
            if(url.contains(keyword)) {
                System.out.println("禁止訪問:  " + url);
                return;
            }
        }
        modem.httpAccess(url);
    }

}
```

Course: Soft Design

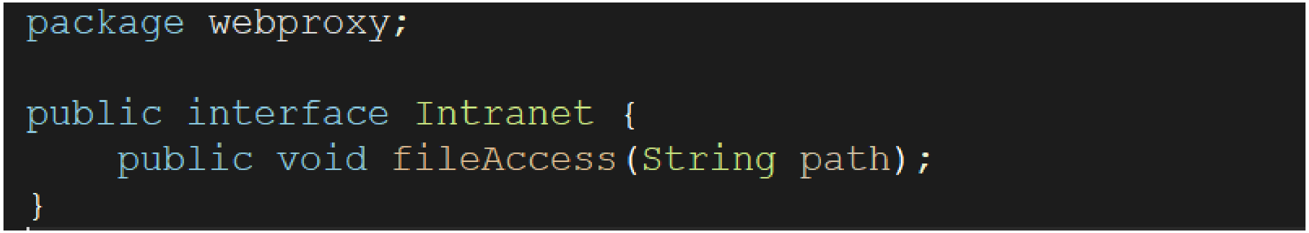


```java
package webproxy;

public class ClientTest1 {

    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        Internet proxyNet = new RouterProxy();
        proxyNet.httpAccess("www.abc.com");
        proxyNet.httpAccess("www.電影.com");
        proxyNet.httpAccess("www.工作.com");

    }

}
```

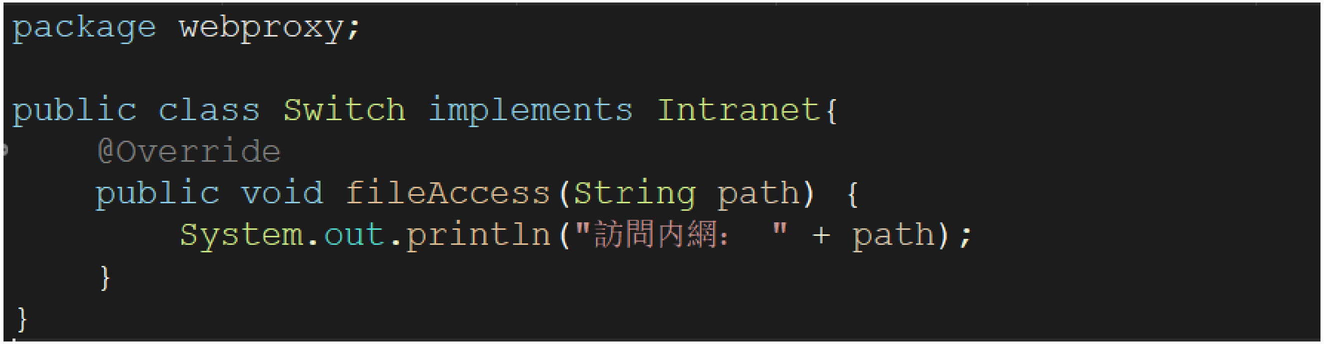


```
撥號上網......連接成功
正在訪問:   www.abc.com
禁止訪問:   www.電影.com
正在訪問:   www.工作.com
```

Course: Soft Design

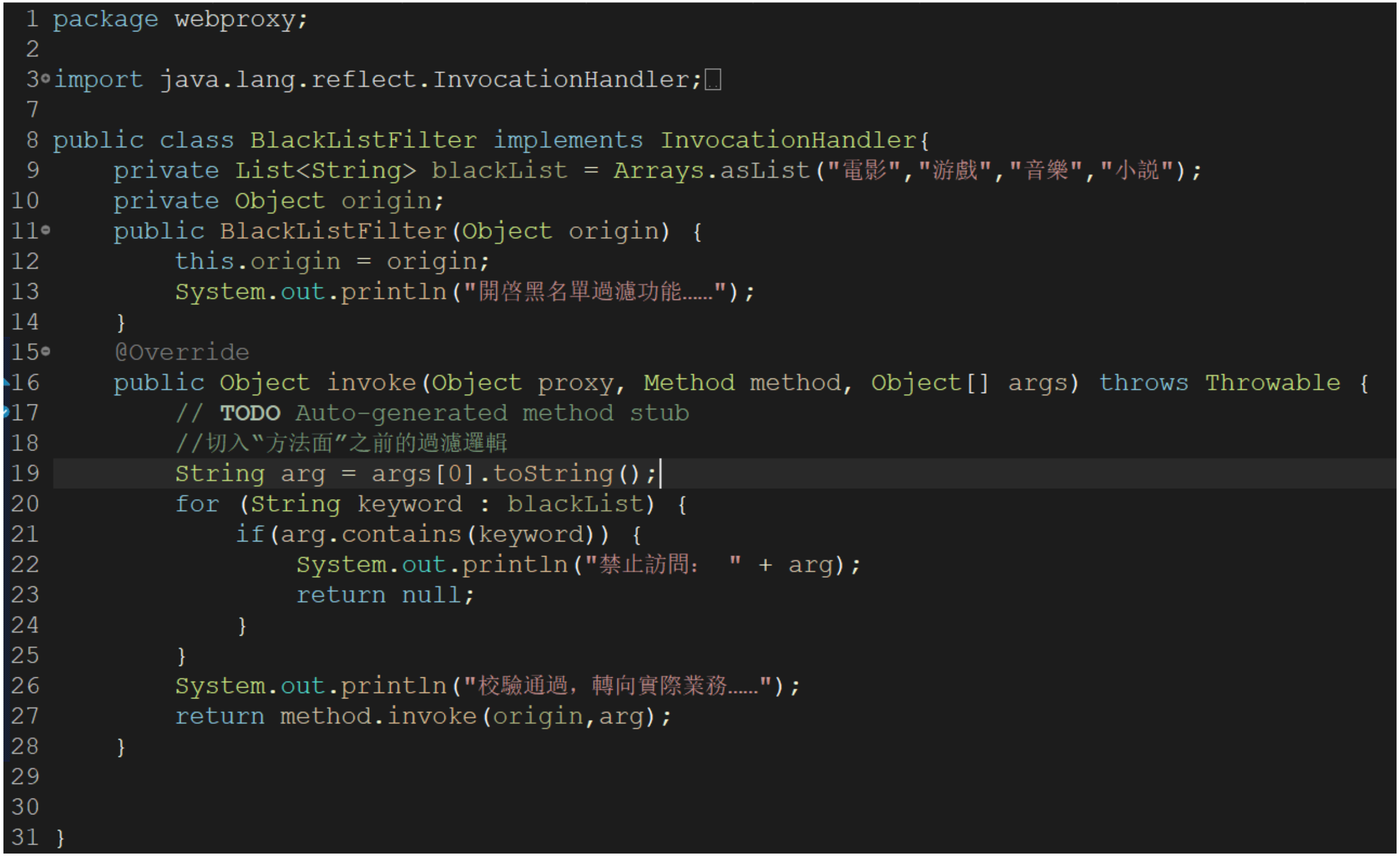


```java
package webproxy;

public interface Intranet {
    public void fileAccess(String path);
}
```

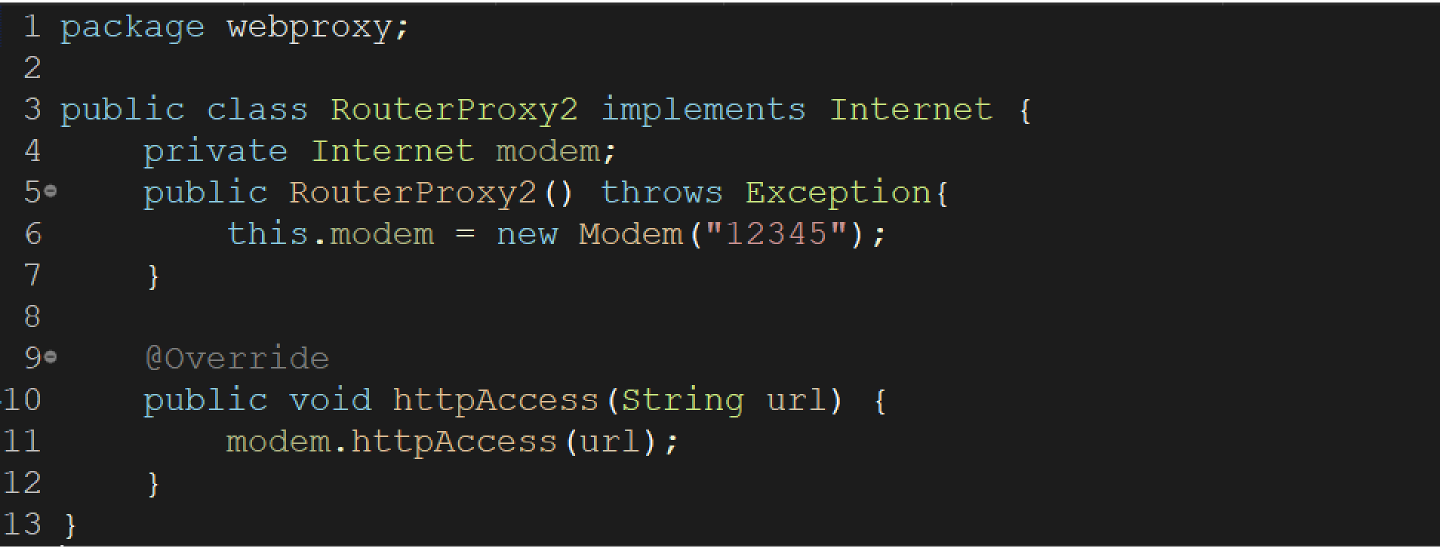


```java
package webproxy;

public class Switch implements Intranet{
    @Override
    public void fileAccess(String path) {
        System.out.println("訪問內網:  " + path);
    }
}
```

Course: Soft Design

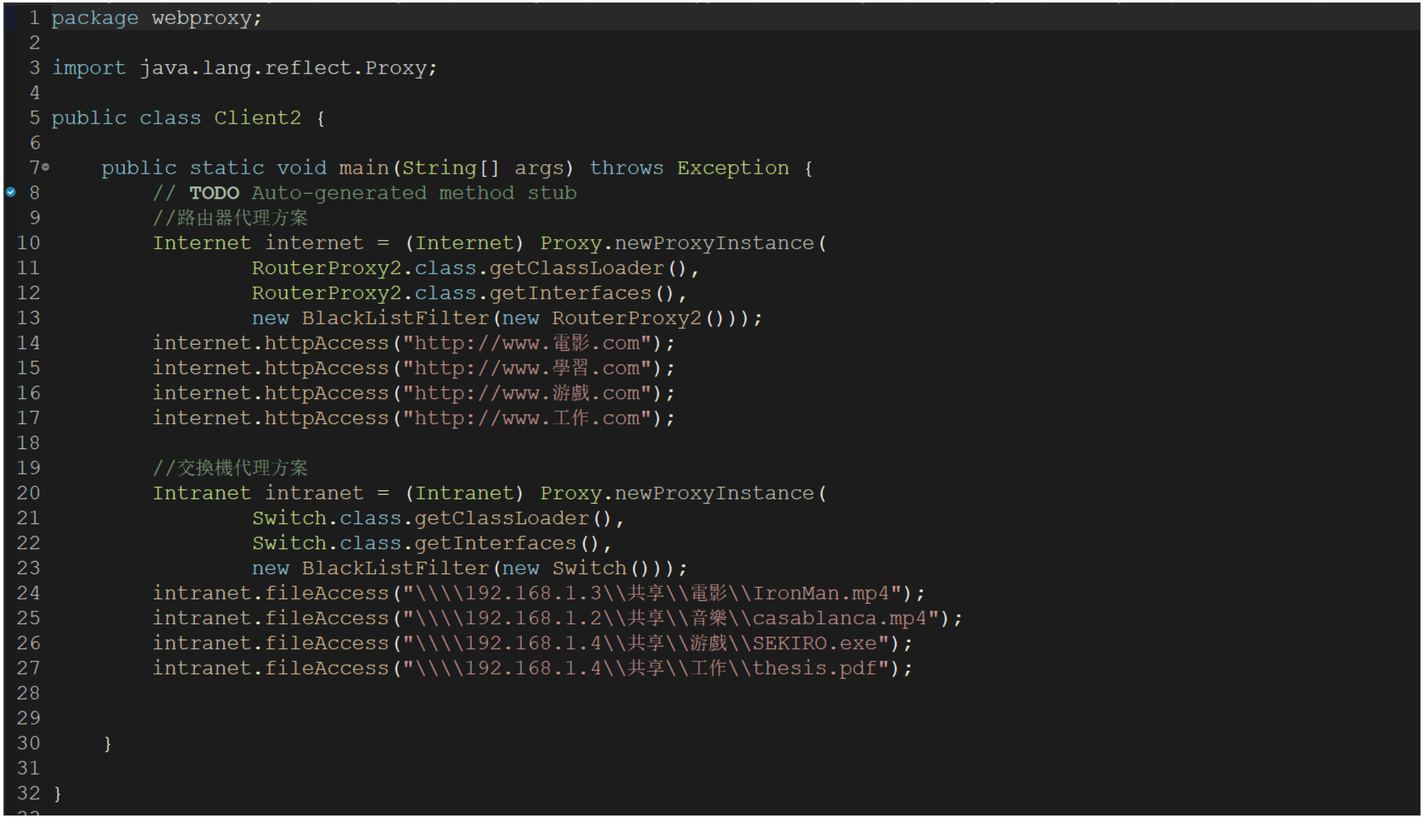


```
package webproxy;

import java.lang.reflect.InvocationHandler;

public class BlackListFilter implements InvocationHandler{
    private List<String> blackList = Arrays.asList("電影","游戲","音樂","小説");
    private Object origin;
    public BlackListFilter(Object origin) {
        this.origin = origin;
        System.out.println("開啓黑名單過濾功能......");
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // TODO Auto-generated method stub
        //切入"方法面"之前的過濾邏輯
        String arg = args[0].toString();
        for (String keyword : blackList) {
            if(arg.contains(keyword)) {
                System.out.println("禁止訪問:  " + arg);
                return null;
            }
        }
        System.out.println("校驗通過，轉向實際業務......");
        return method.invoke(origin,arg);
    }


}
```

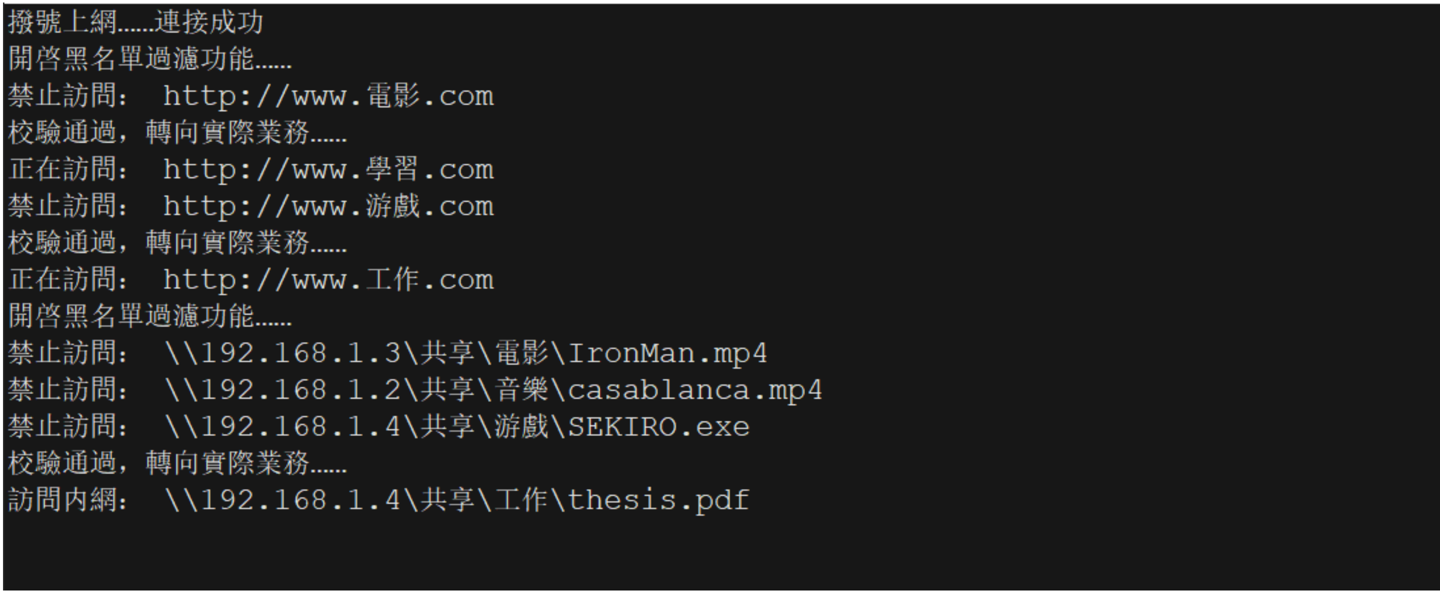


```java
package webproxy;

public class RouterProxy2 implements Internet {
	private Internet modem;
	public RouterProxy2() throws Exception{
		this.modem = new Modem("12345");
	}

	@Override
	public void httpAccess(String url) {
		modem.httpAccess(url);
	}
}
```

Course: Soft Design



```java
package webproxy;

import java.lang.reflect.Proxy;

public class Client2 {

    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        //路由器代理方案
        Internet internet = (Internet) Proxy.newProxyInstance(
                RouterProxy2.class.getClassLoader(),
                RouterProxy2.class.getInterfaces(),
                new BlackListFilter(new RouterProxy2()));
        internet.httpAccess("http://www.電影.com");
        internet.httpAccess("http://www.學習.com");
        internet.httpAccess("http://www.游戲.com");
        internet.httpAccess("http://www.工作.com");

        //交換機代理方案
        Intranet intranet = (Intranet) Proxy.newProxyInstance(
                Switch.class.getClassLoader(),
                Switch.class.getInterfaces(),
                new BlackListFilter(new Switch()));
        intranet.fileAccess("\\\\192.168.1.3\\共享\\電影\\IronMan.mp4");
        intranet.fileAccess("\\\\192.168.1.2\\共享\\音樂\\casablanca.mp4");
        intranet.fileAccess("\\\\192.168.1.4\\共享\\游戲\\SEKIRO.exe");
        intranet.fileAccess("\\\\192.168.1.4\\共享\\工作\\thesis.pdf");


    }

}
```



```
撥號上網......連接成功
開啓黑名單過濾功能......
禁止訪問:  http://www.電影.com
校驗通過, 轉向實際業務......
正在訪問:  http://www.學習.com
禁止訪問:  http://www.游戲.com
校驗通過, 轉向實際業務......
正在訪問:  http://www.工作.com
開啓黑名單過濾功能......
禁止訪問:  \\192.168.1.3\共享\電影\IronMan.mp4
禁止訪問:  \\192.168.1.2\共享\音樂\casablanca.mp4
禁止訪問:  \\192.168.1.4\共享\游戲\SEKIRO.exe
校驗通過, 轉向實際業務......
訪問内網:  \\192.168.1.4\共享\工作\thesis.pdf
```

Bridge (橋接模式)

Intent

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

Also known as

Handle/Body

Motivation

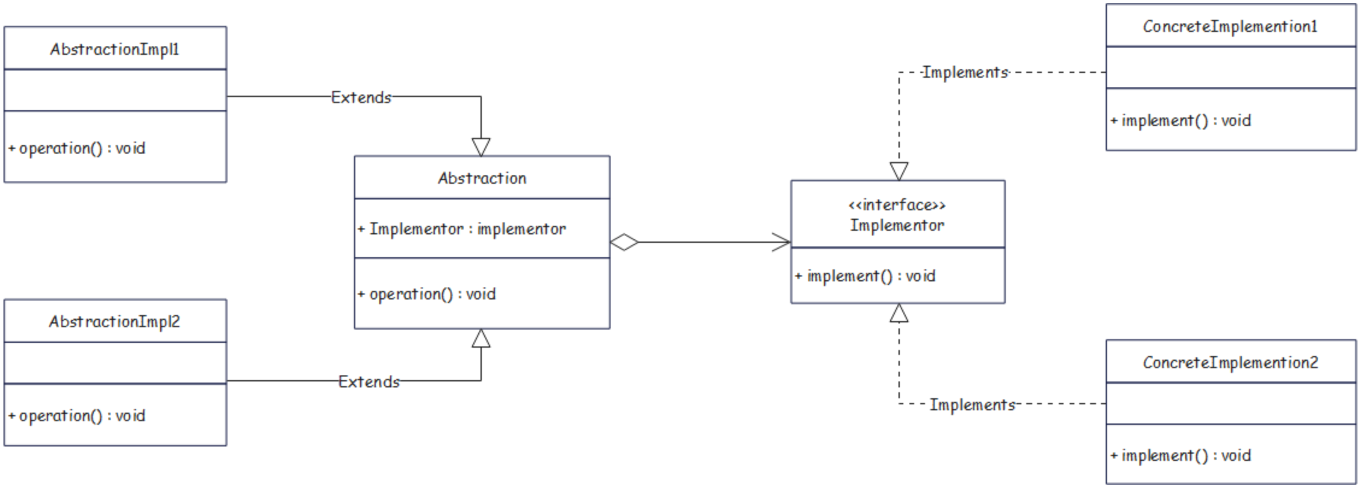
It makes the use of abstract classes and subclass methods more flexible, and fundamentally solves the independent modification, expansion, and reuse of abstract parts and implementation parts.

Applicability

Use the Bridge pattern when

- you want to avoid a permanent binding between an abstraction and its implementation.
- both the abstractions and their implementations should be extensible by subclassing.
- changes in the implementation of an abstraction should have no impact on clients
- you want to hide the implementation of an abstraction completely from clients.
- you want to share an implementation among multiple objects, and this fact should be hidden from the client.

* Class Diagram



Participants

Abstraction

- defines the abstraction's interface.
- maintains a reference to an object of type Implementor.

RefinedAbstraction

- Extends the interface defined by Abstraction.

Implementor

- Define the interface of the implementing class, which provides the basic operations

ConcreteImplementor

- implements the Implementor interface and defines its concrete implementation.

Collaborations

- Abstraction forwards client requests to its Implementor object.

Consequences

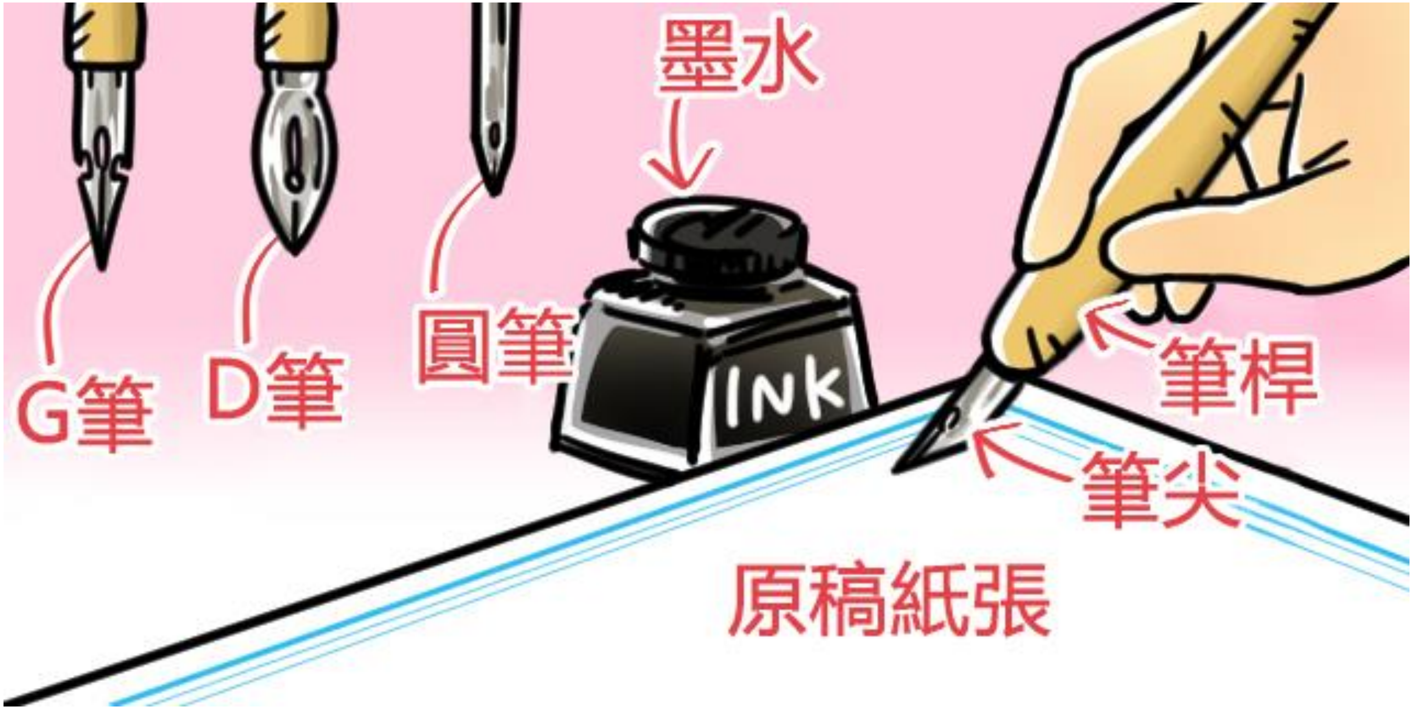
The Bridge pattern has the following consequences:

- Decoupling interface and implementation.
- Improved extensibility.
- Hiding implementation details from clients.

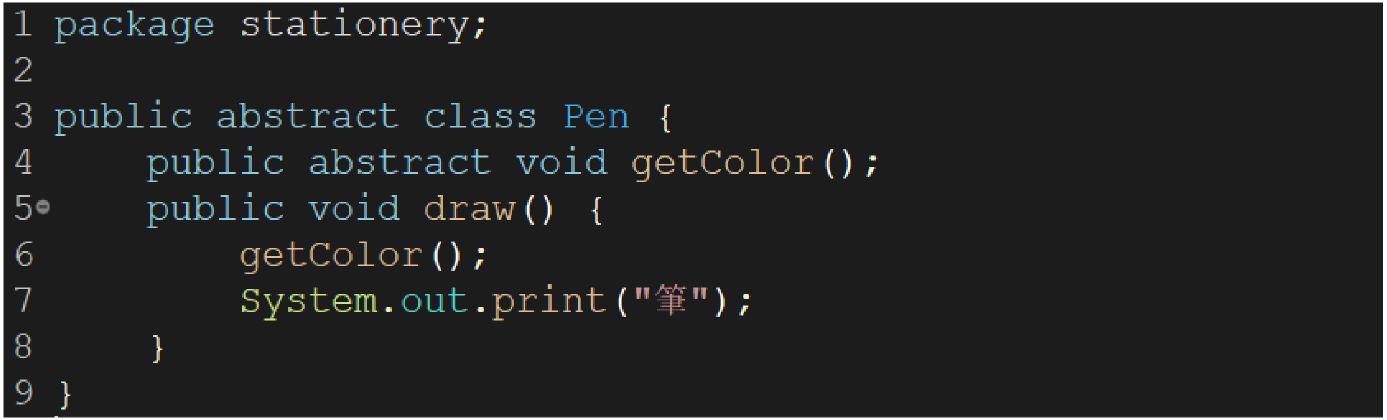
Implementation

- *Only one Implementor.*
- *Creating the right Implementor object.*
- *Sharing implementors.*
- *Using multiple inheritance.*

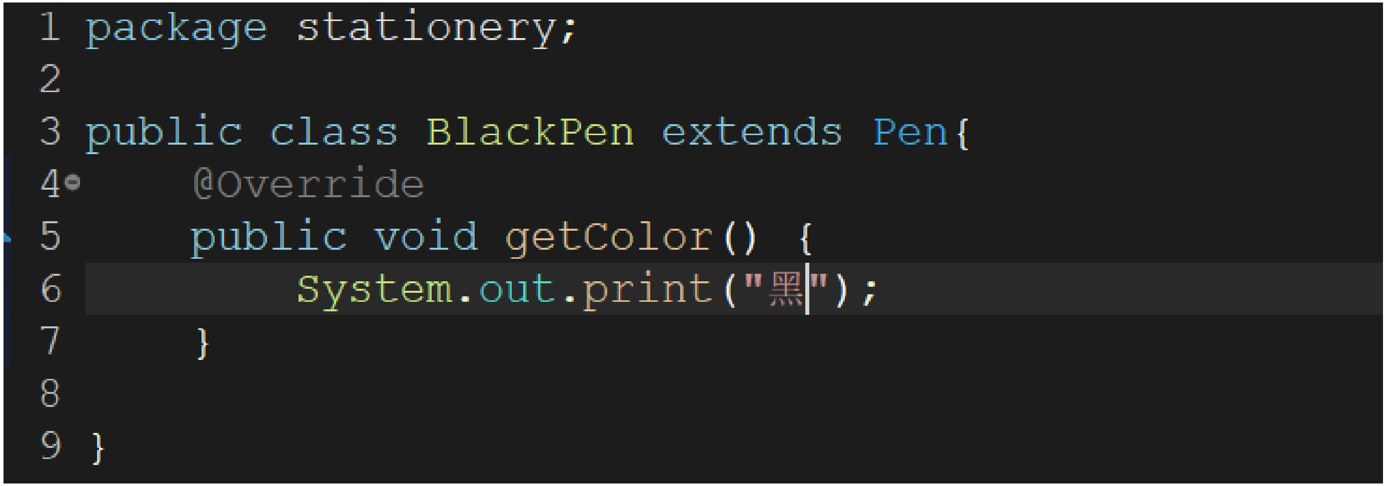
Course: Soft Design



Course: Soft Design

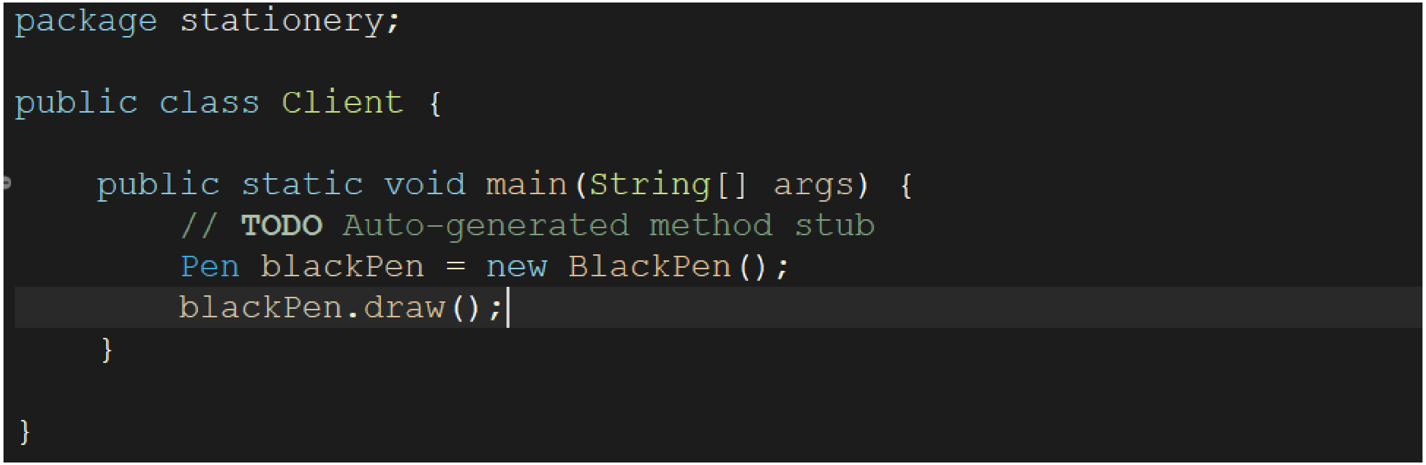


```java
package stationery;

public abstract class Pen {
	public abstract void getColor();
	public void draw() {
		getColor();
		System.out.print("筆");
	}
}
```

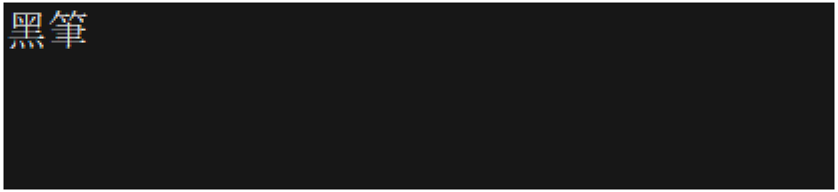


```java
package stationery;

public class BlackPen extends Pen{
	@Override
	public void getColor() {
		System.out.print("黑");
	}

}
```

Course: Soft Design

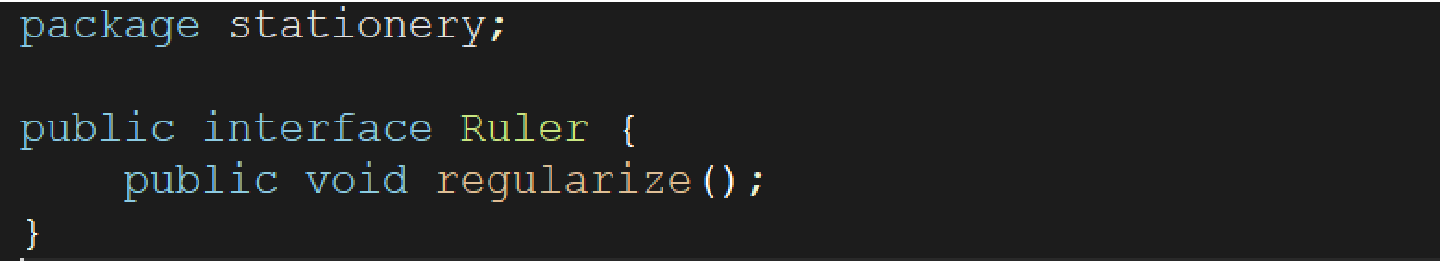


```
package stationery;

public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Pen blackPen = new BlackPen();
        blackPen.draw();
    }

}
```

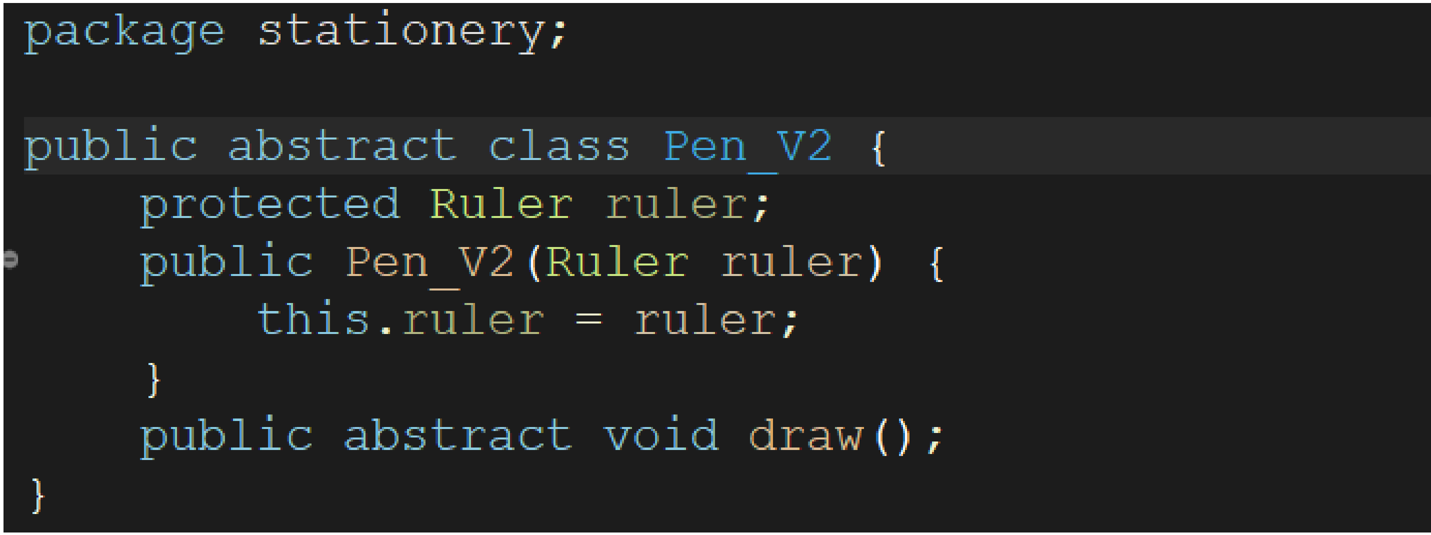


```
黑筆
```

Course: Soft Design

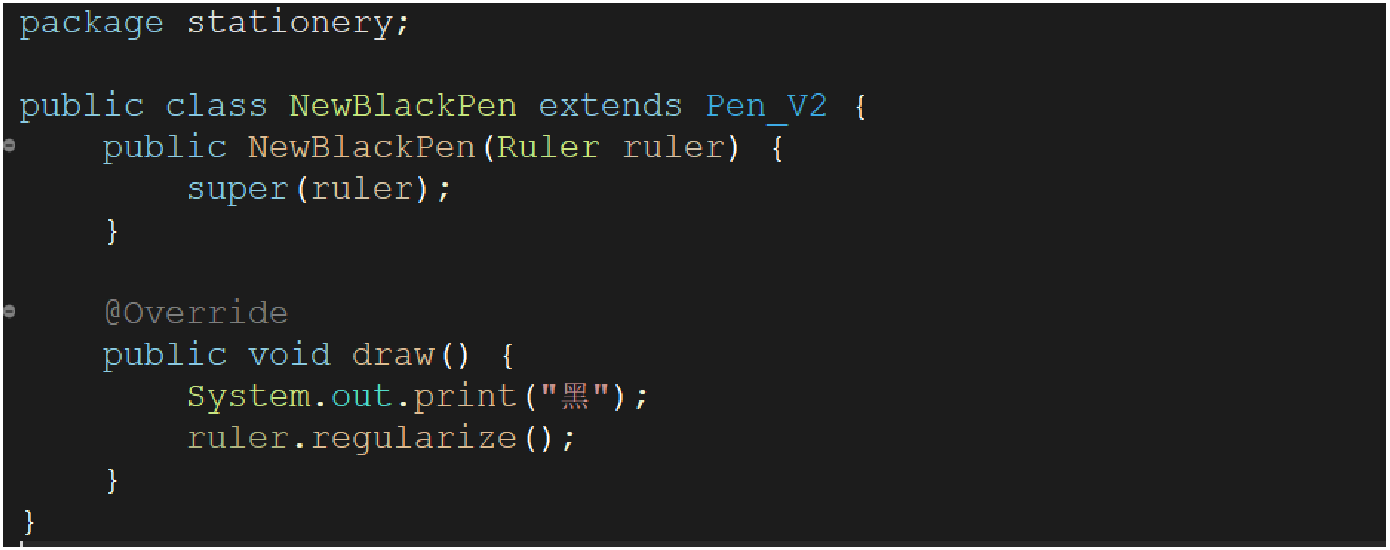


```
package stationery;

public interface Ruler {
    public void regularize();
}
```

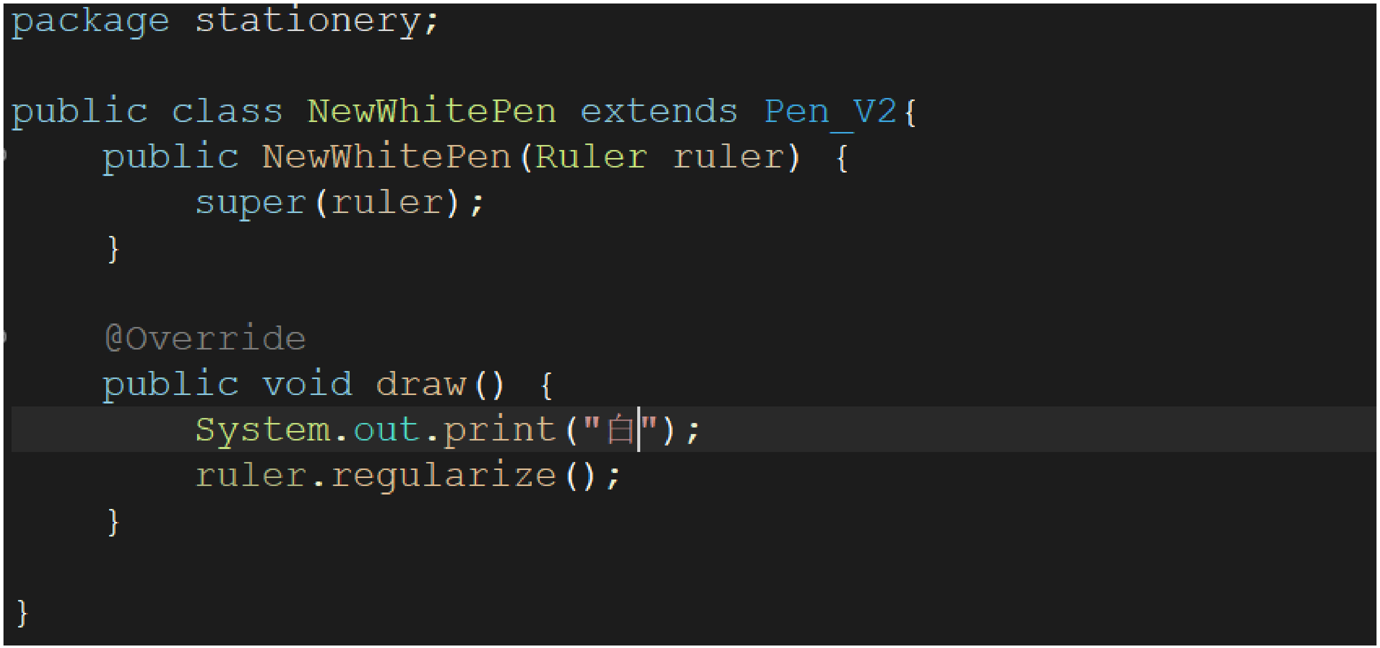


```
package stationery;

public abstract class Pen_V2 {
    protected Ruler ruler;
    public Pen_V2(Ruler ruler) {
        this.ruler = ruler;
    }
    public abstract void draw();
}
```

Course: Soft Design

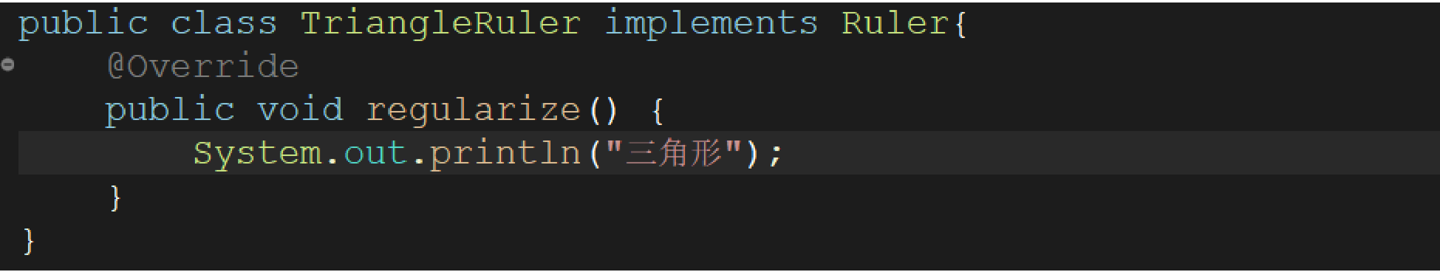


```
package stationery;

public class NewBlackPen extends Pen_V2 {
    public NewBlackPen(Ruler ruler) {
        super(ruler);
    }

    @Override
    public void draw() {
        System.out.print("黑");
        ruler.regularize();
    }
}
```

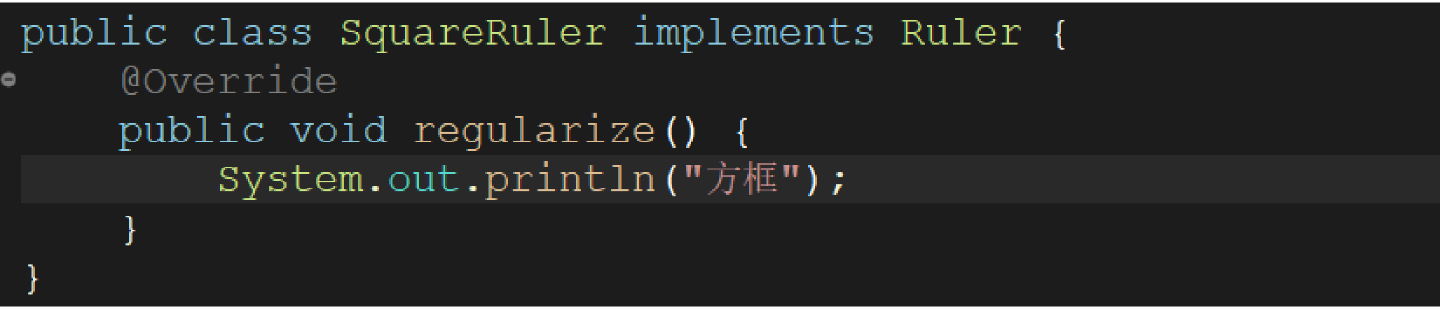


```
package stationery;

public class NewWhitePen extends Pen_V2{
    public NewWhitePen(Ruler ruler) {
        super(ruler);
    }

    @Override
    public void draw() {
        System.out.print("白");
        ruler.regularize();
    }

}
```

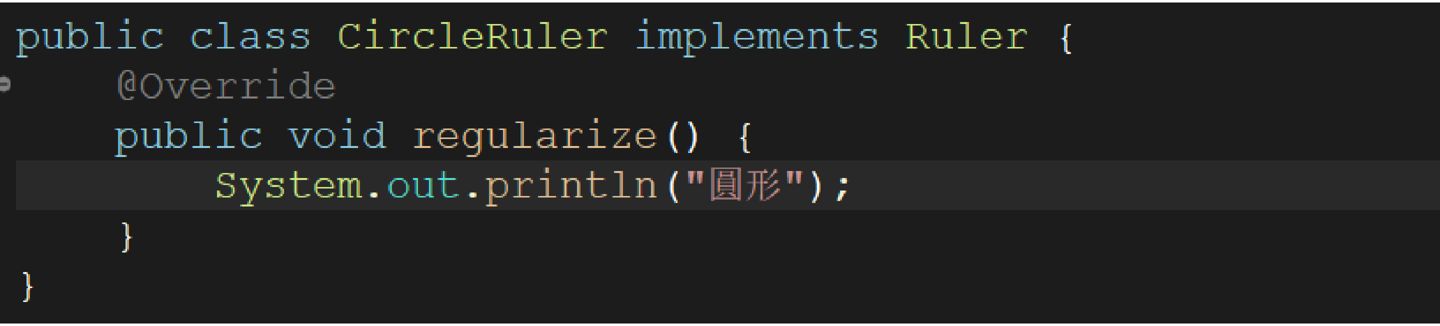
Course: Soft Design



```java
public class TriangleRuler implements Ruler{
    @Override
    public void regularize() {
        System.out.println("三角形");
    }
}
```

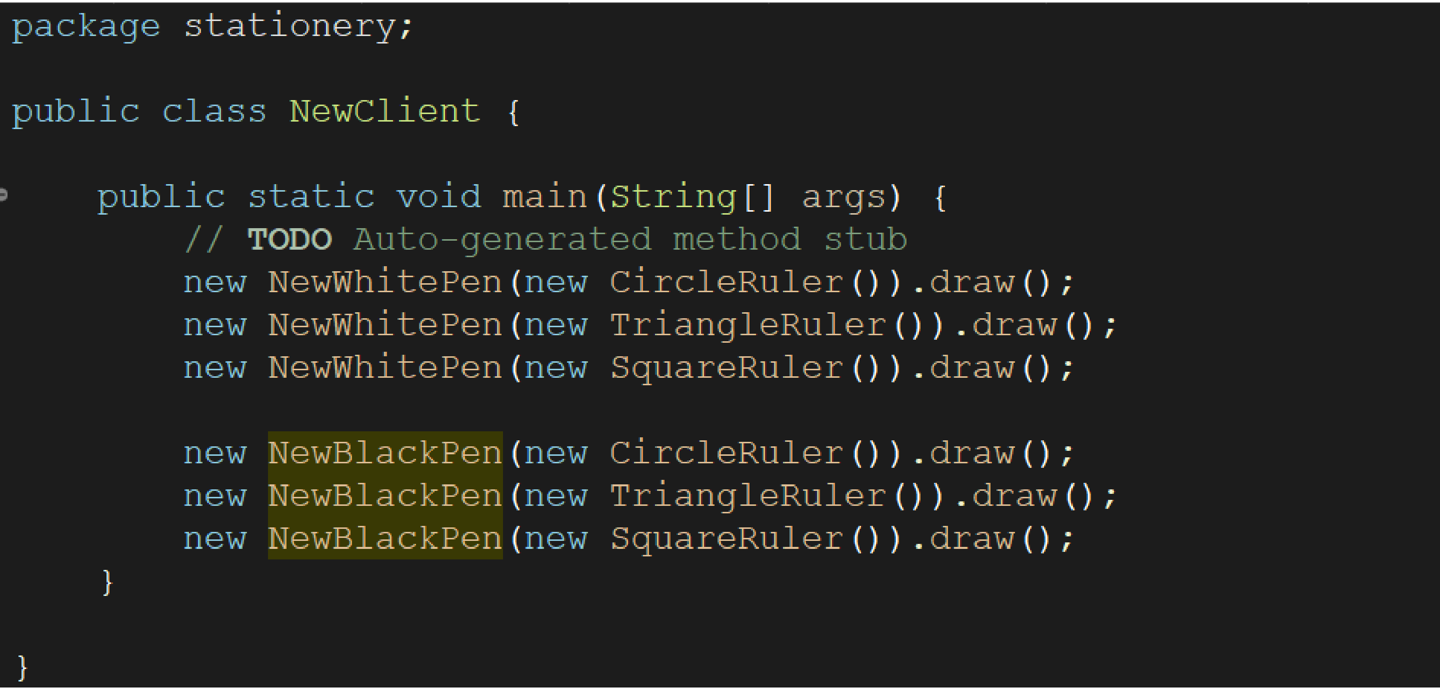


```java
public class SquareRuler implements Ruler {
    @Override
    public void regularize() {
        System.out.println("方框");
    }
}
```



```java
public class CircleRuler implements Ruler {
    @Override
    public void regularize() {
        System.out.println("圓形");
    }
}
```

Course: Soft Design



```
package stationery;

public class NewClient {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        new NewWhitePen(new CircleRuler()).draw();
        new NewWhitePen(new TriangleRuler()).draw();
        new NewWhitePen(new SquareRuler()).draw();

        new NewBlackPen(new CircleRuler()).draw();
        new NewBlackPen(new TriangleRuler()).draw();
        new NewBlackPen(new SquareRuler()).draw();
    }

}
```

Principle 73

Use coupling and cohesion

Coupling and cohesion were defined in the 1970s by Larry Constantine and Edward Yourdon. They are still the best ways we know of measuring the inherent maintainability and adaptability of a software system. In short, coupling is a measure of how interrelated two software components are. Cohesion is a measure of how related the functions performed by a software component are. We want to strive for low coupling and high cohesion. High coupling implies that, when we change a component, changes to other components are likely. Low cohesion implies difficulty in isolating the causes of errors or places to adapt to meet new requirements.

Principle 74

Design for change

To accommodate change, the design should be: Modular, that is, it should be composed of independent parts that can be easily upgraded or replaced with a minimum of impact on other parts (see related Principles 65,70,73,and 80). Portable, that is, it should be easily altered to accommodate new host machines and operating systems. Malleable, that is, flexible to accommodate new requirements that hadnot been anticipated. Of minimal intellectual distance (Principle 69). Under intellectual control (Principle 70). Such that it exhibits conceptual integrity (Principle 71).

Principle 75

Design for maintenance

The largest postdesign cost risk for nonsoftware products is manufacturing. The largest postdesign cost risk for software products is maintenance. In the former case, design for manufacturability is a major design driver. Unfortunately, design for maintainability is not the standard for software. It should be.

Principle 76

Design for errors

No matter how much you work on your software, it will have errors. You should make design decisions to optimize the likelihood that:

1. Errors are not introduced.
2. Errors that are introduced are easily detected.
3. Errors that remain in the software after deployment are either noncritical or are compensated for during execution so that the error does not cause a disaster.

Principle 77

Build generality into software

When decomposing a system into its subcomponents, stay cognizant of the potential for generality. Obviously, when a similar function is needed in multiple places, construct just one general function rather than multiple similar functions. Also, when constructing a function needed in just one place, build in generality where it makes sense for future enhancements.

Principle 78

Build flexibility into software

A software component exhibits flexibility if it can be easily modified to per-form its function (or a similar function) in a different situation. Flexible software components are more difficult to design than less flexible components. However, such components(1) are more run-time-efficient than general components (Principle77) and (2) are more easily reused than less flexible components in diverse applications.

Principle 79

Use efficient algorithms

Knowledge of the theory of algorithm complexity is an absolute prerequisite for being a good designer. Given any specific problem, you could specify an infinite number of alternative algorithms to solve it. The theory of "analysis of algorithms" provides us with the knowledge of how to differentiate between algorithms that will be inherently slow(regardless of how well they are coded) and those that will be orders of magnitude faster. Dozens of excellent books exist on this subject. Every good undergraduate computer science program will offer a course on it.

Principle 80

Module specifications provide all the information the user needs and nothing more

A key part of the design process is the precise definition of each and every software component in the system. This specification will become the"visible" or"public" part of the component. It must include everything a user needs, such as its purpose, its name, its method of invocation, and details of how it communicates with its environment. Anything that the user does not need should be specifically excluded.In most cases, the algorithms and internal data structures used should be excluded.

Principle 81

Design is multidimensional

The same is true for software design. A complete software design includes at least:

1. Packaging.
2. Needs hierarchy.
3. Invocation.
4. Processes.

Principle 82

Great designs come from great designers

The difference between a poor design and a good design may be the result of a sound design method, superior training, better education, or other factors. However, a really great design is the brainchild of a really great designer. Great designs are clean, simple, elegant, fast, maintainable, and easy to implement. They are the result of inspiration and insight, not just hard work or following a step-by-step design method. Invest heavily in your best designers. They are your future.

Principle 83

Know your application

No matter how well the requirements have been written, the selection of optimal architectures and algorithms is very much a function of knowing the unique characteristics of an application. Expected behavior under stress situations, expected frequency of inputs, life-critical nature of response times, likelihood of new hardware, impact of weather on expected system performance, and so on are all application-specific and often demand a specific subset of possible alternative architectures and algorithms.

Principle 84

You can reuse without a big investment

Chances are that the most effective way to reuse software components is from a repository of crafted, hand-picked items that were tailored specifically for reuse. However, this requires considerable investment in both time and money. It is possible to reuse in the short term through a technique called salvaging. Simply stated, salvaging is asking others in the organization,"Have you ever built a software component that does x?" You find it, you adapt it, you employ it. This may not be efficient in the long term, but it certainly works now; and then you have no more excuses not to reuse.

Principle 85

"garbage in, garbage out" is incorrect

Many people quote the expression"garbage in, garbage out" as if it were acceptable for software to behave like this. It isn't.If a user provides invalid input data, the program should respond with an intelligent message that describes why the input was invalid. If a software component receives invalid data, it should not process it, but instead should return an error code back to the component that transmitted the invalid data. This mind-set helps diminish the domino effect caused by software faults and makes it easier to determine error causes by (1) catching the fault early and (2) preventing subsequent data corruption.

Principle 86

Software reliability can be achieved through redundancy

In hardware systems, high reliability or availability (Principle 57) is often achieved through redundancy. Thus, if a system component is expected to exhibit a mean-time-between-failures of x, we can manufacture two or three such components and run them in either:

1. Parallel. For example, they all do all the work and, when their responses differ, one is turned off with no impact on overall system functionality.
2. Or cold standby. A backup computer might be powered on only when a hardware failure is detected in the operational computer.