LECTURE 04

Facade, **Composite** and **Decorator**

Software Development Principle 58~76

Facade (外观模式)

Intent

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.

Motivation

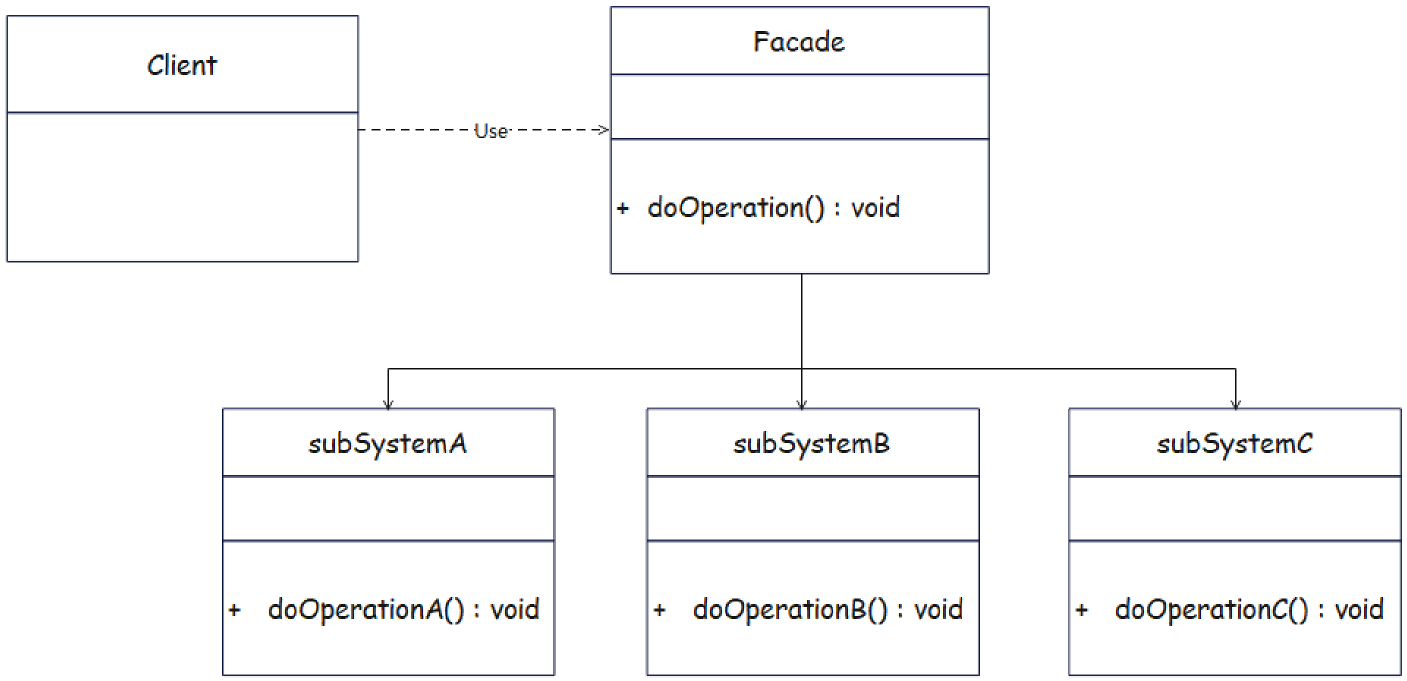
Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a **facade** object that provides a single, simplified interface to the more general facilities of a subsystem.

Applicability

Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem.
- there are many dependencies between clients and the implementation classes of an abstraction.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

Class Diagram



Participants

Façade (Compiler)

- knows which subsystem classes are responsible for a request.
- delegates client requests to appropriate subsystem objects.

subsystem classes (Scanner\Parser\ProgramNodes)

- implements the operations to create concrete product objects.
- handle work assigned by the Facade object.
- have no knowledge of the facade; that is, they keep no references to it.

Collaborations

- Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s).
- Clients that use the facade don't have to access its subsystem objects directly.

Consequences

The Facade pattern offers the following benefits:

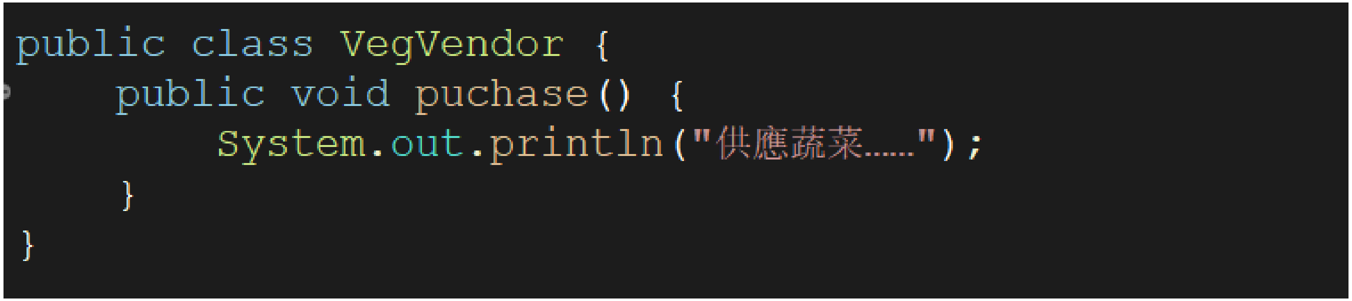
1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
2. It promotes weak coupling between the subsystem and its clients. Often the components in a subsystem are strongly coupled.
3. It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Implementation

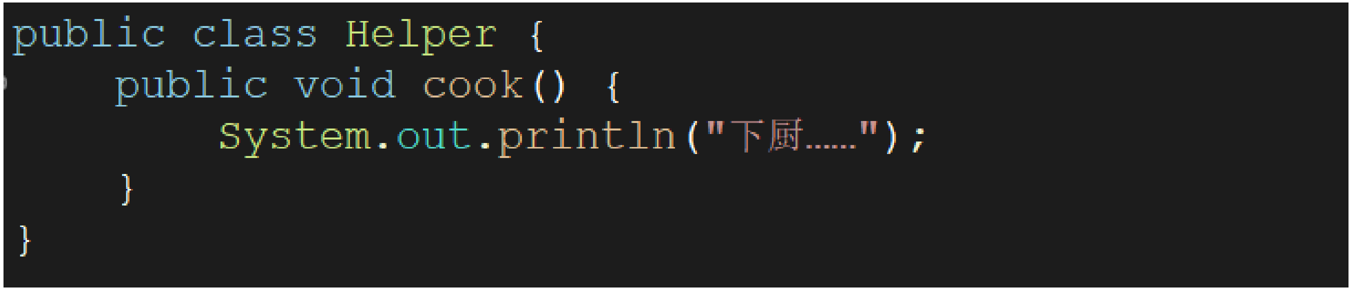
Reducing client-subsystem coupling.

Public versus private subsystem classes.

Course: Soft Design

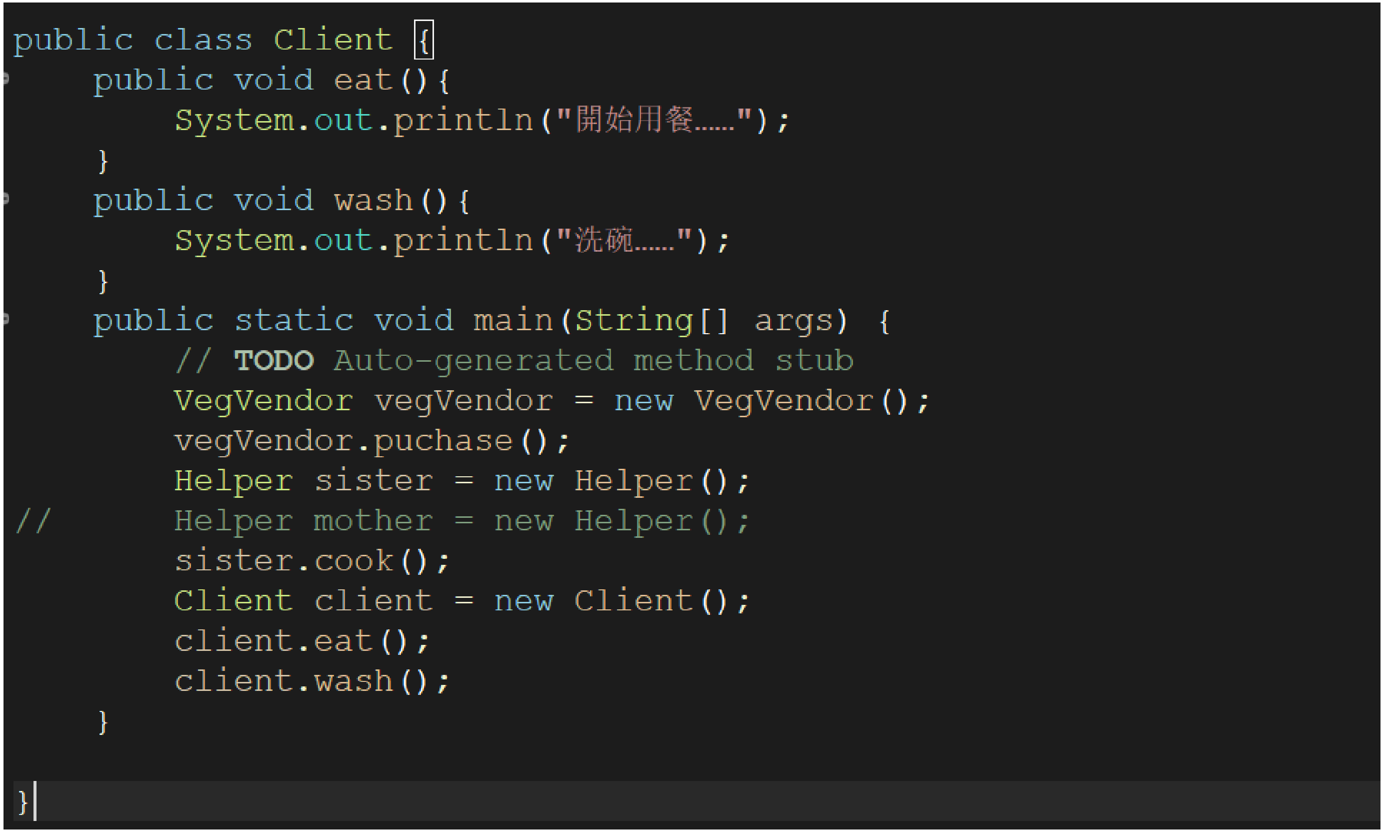


```java
public class VegVendor {
    public void puchase() {
        System.out.println("供應蔬菜......");
    }
}
```

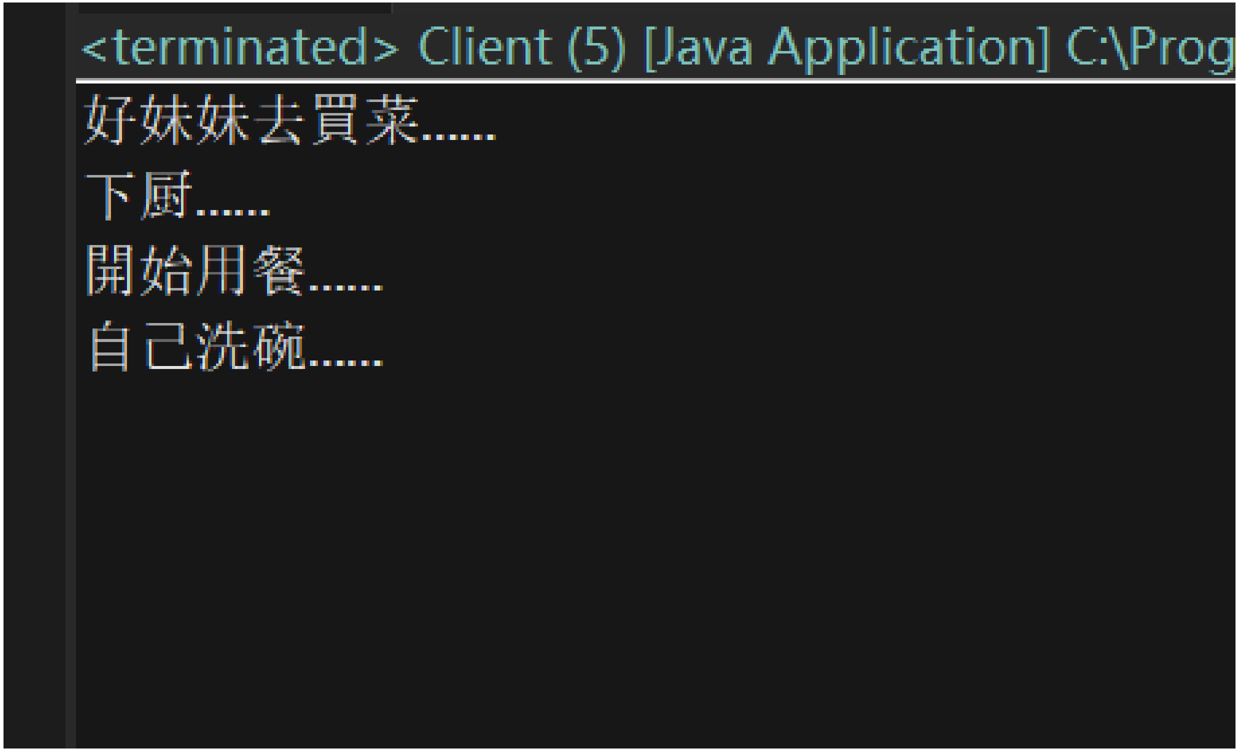


```java
public class Helper {
    public void cook() {
        System.out.println("下厨......");
    }
}
```

Course: Soft Design

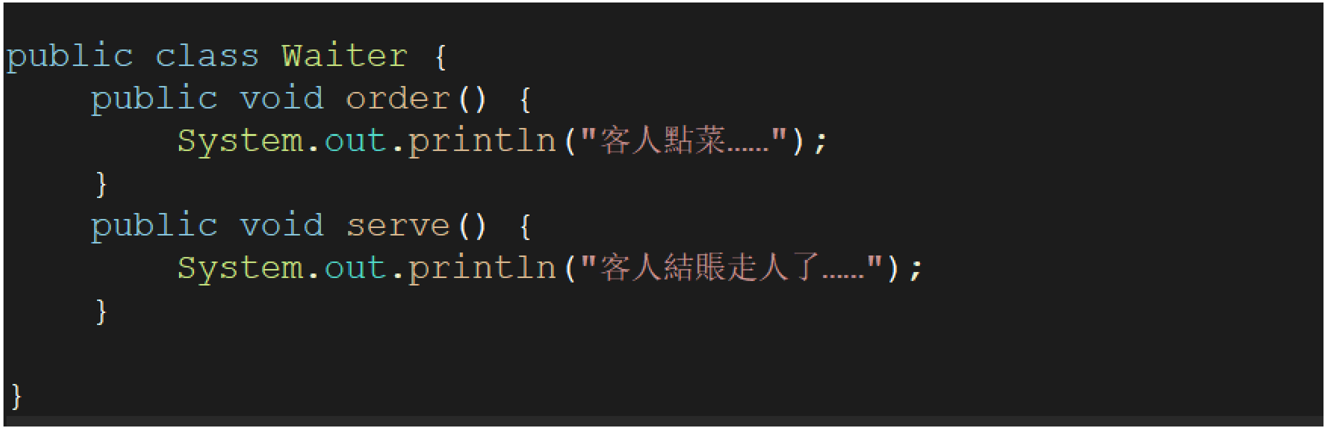


```
public class Client {
	public void eat(){
		System.out.println("開始用餐......");
	}
	public void wash(){
		System.out.println("洗碗......");
	}
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		VegVendor vegVendor = new VegVendor();
		vegVendor.puchase();
		Helper sister = new Helper();
//		Helper mother = new Helper();
		sister.cook();
		Client client = new Client();
		client.eat();
		client.wash();
	}

}
```

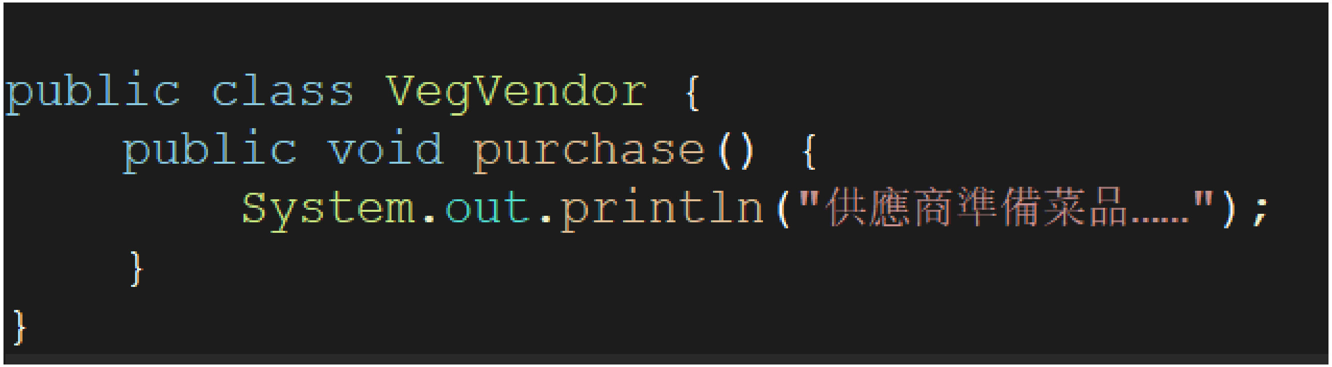


```
<terminated> Client (5) [Java Application] C:\Prog
好妹妹去買菜……
下厨……
開始用餐……
自己洗碗……
```

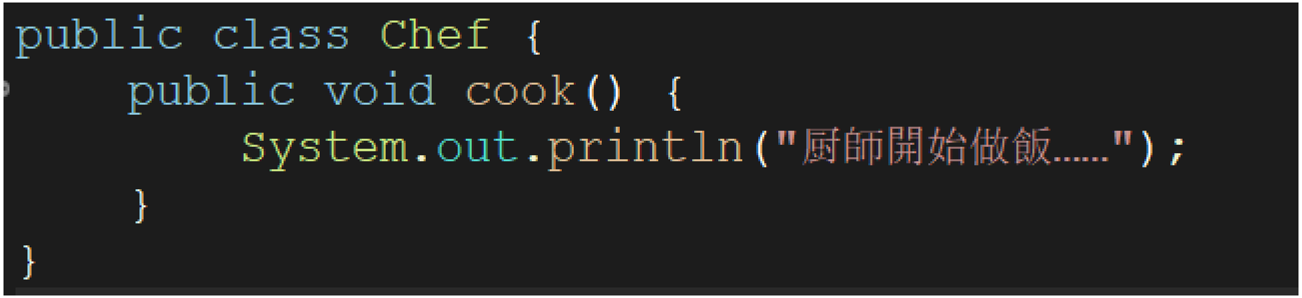
Course: Soft Design



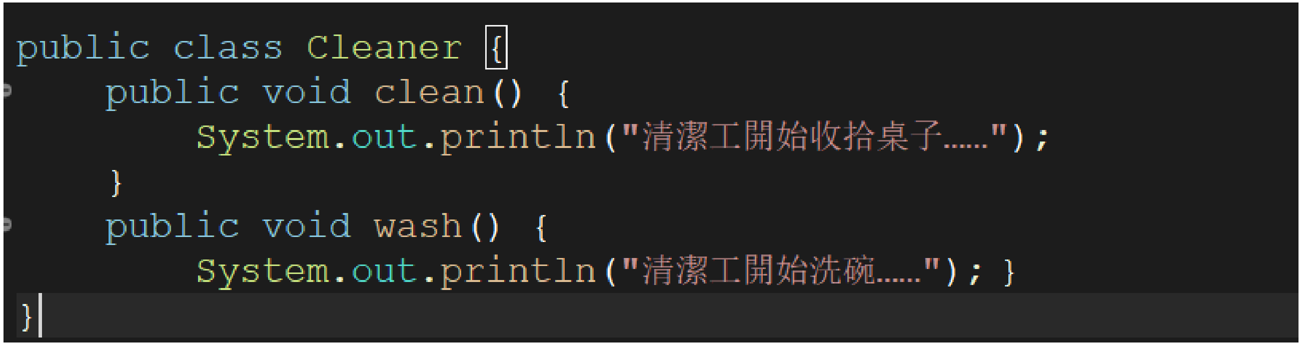
```
public class Waiter {
    public void order() {
        System.out.println("客人點菜......");
    }
    public void serve() {
        System.out.println("客人結賬走人了......");
    }

}
```



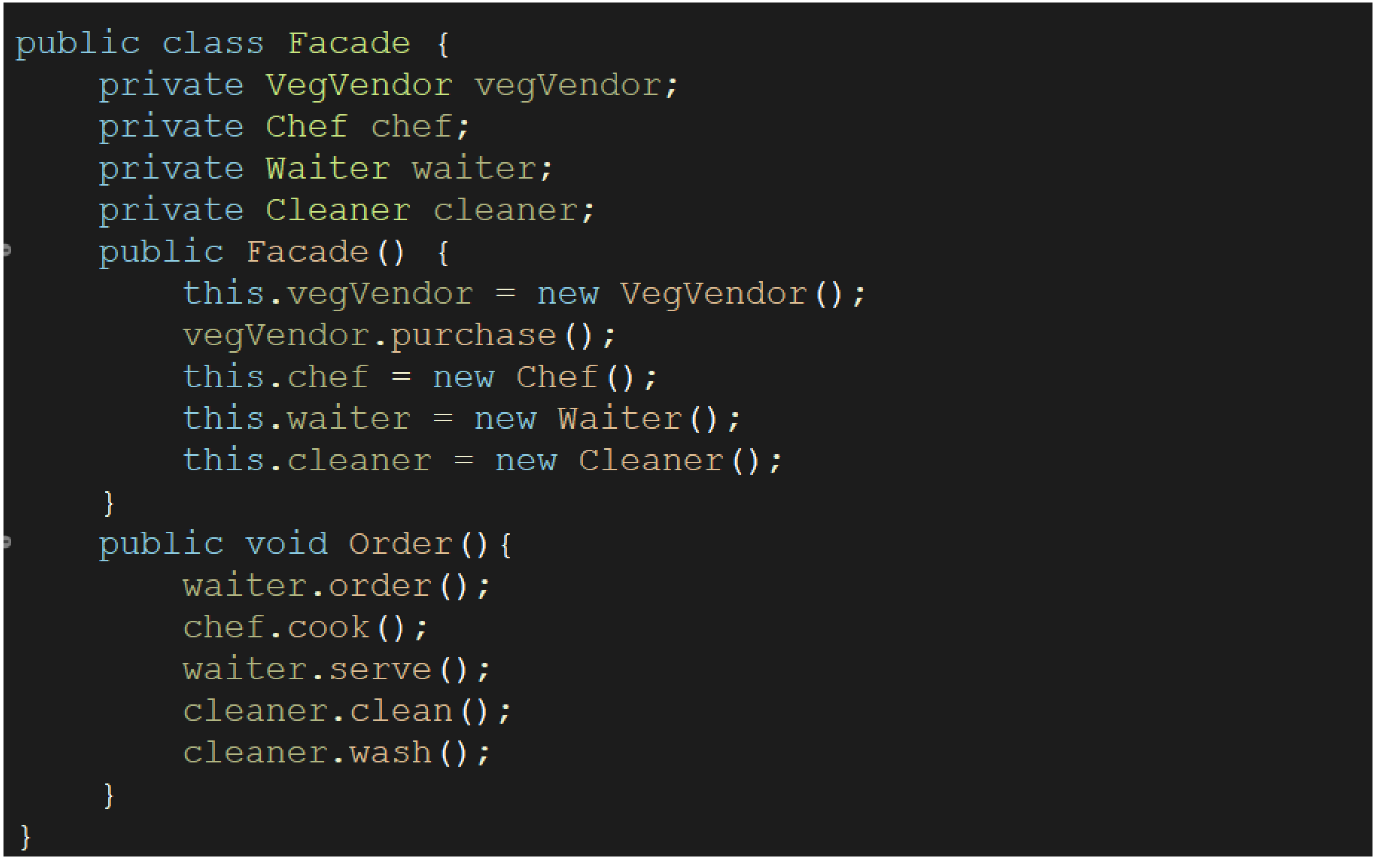
```
public class VegVendor {
    public void purchase() {
        System.out.println("供應商準備菜品......");
    }
}
```



```
public class Chef {
    public void cook() {
        System.out.println("厨師開始做飯.....");
    }
}
```

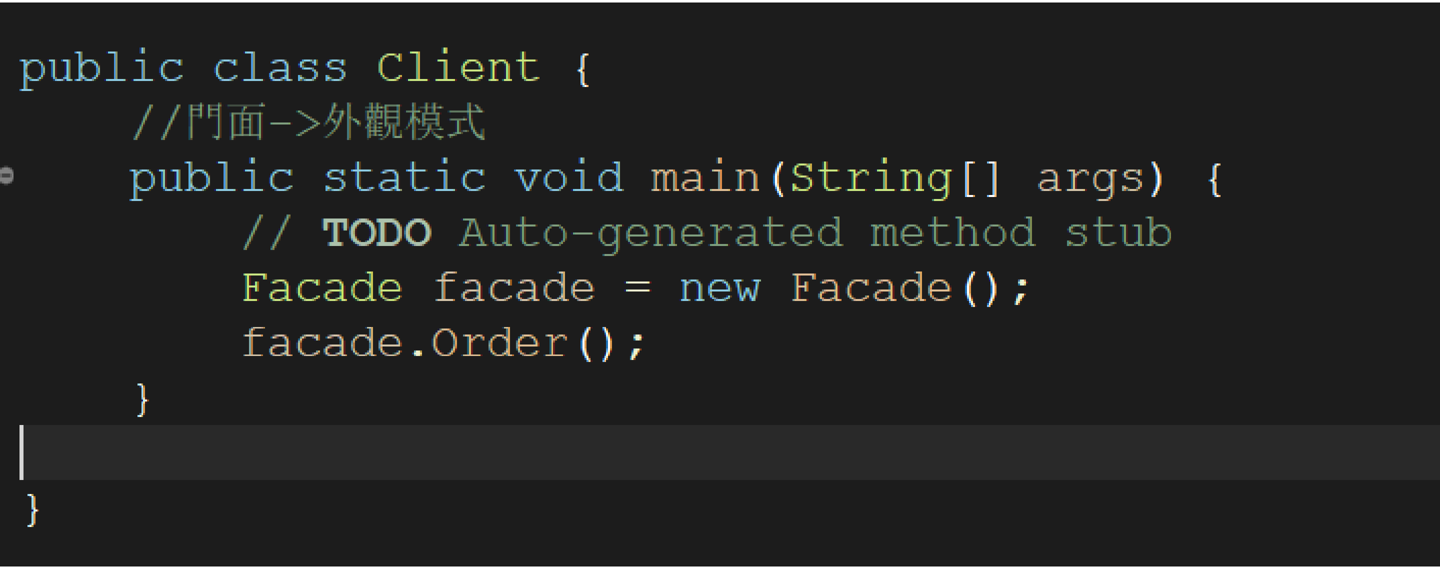


```
public class Cleaner {
    public void clean() {
        System.out.println("清潔工開始收拾桌子.....");
    }
    public void wash() {
        System.out.println("清潔工開始洗碗....."); }
}
```



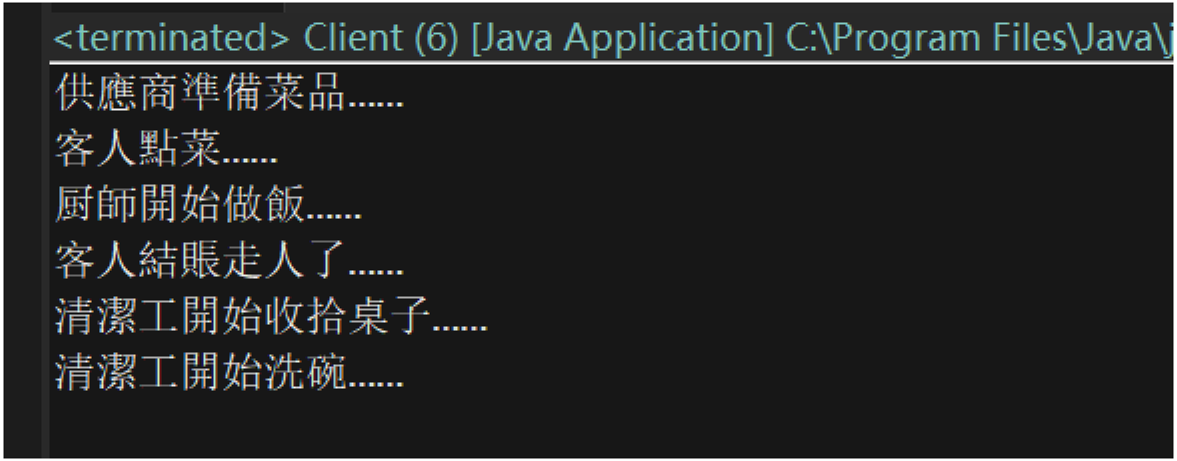
```
public class Facade {
    private VegVendor vegVendor;
    private Chef chef;
    private Waiter waiter;
    private Cleaner cleaner;
    public Facade() {
        this.vegVendor = new VegVendor();
        vegVendor.purchase();
        this.chef = new Chef();
        this.waiter = new Waiter();
        this.cleaner = new Cleaner();
    }
    public void Order(){
        waiter.order();
        chef.cook();
        waiter.serve();
        cleaner.clean();
        cleaner.wash();
    }
}
```

Course: Soft Design



```java
public class Client {
    //門面->外觀模式
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Facade facade = new Facade();
        facade.Order();
    }

}
```

Course: Soft Design



```
<terminated> Client (6) [Java Application] C:\Program Files\Java\j
供應商準備菜品......
客人點菜......
廚師開始做飯......
客人結賬走人了......
清潔工開始收拾桌子......
清潔工開始洗碗......
```

Composite(組合模式)

Intent

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.

Motivation

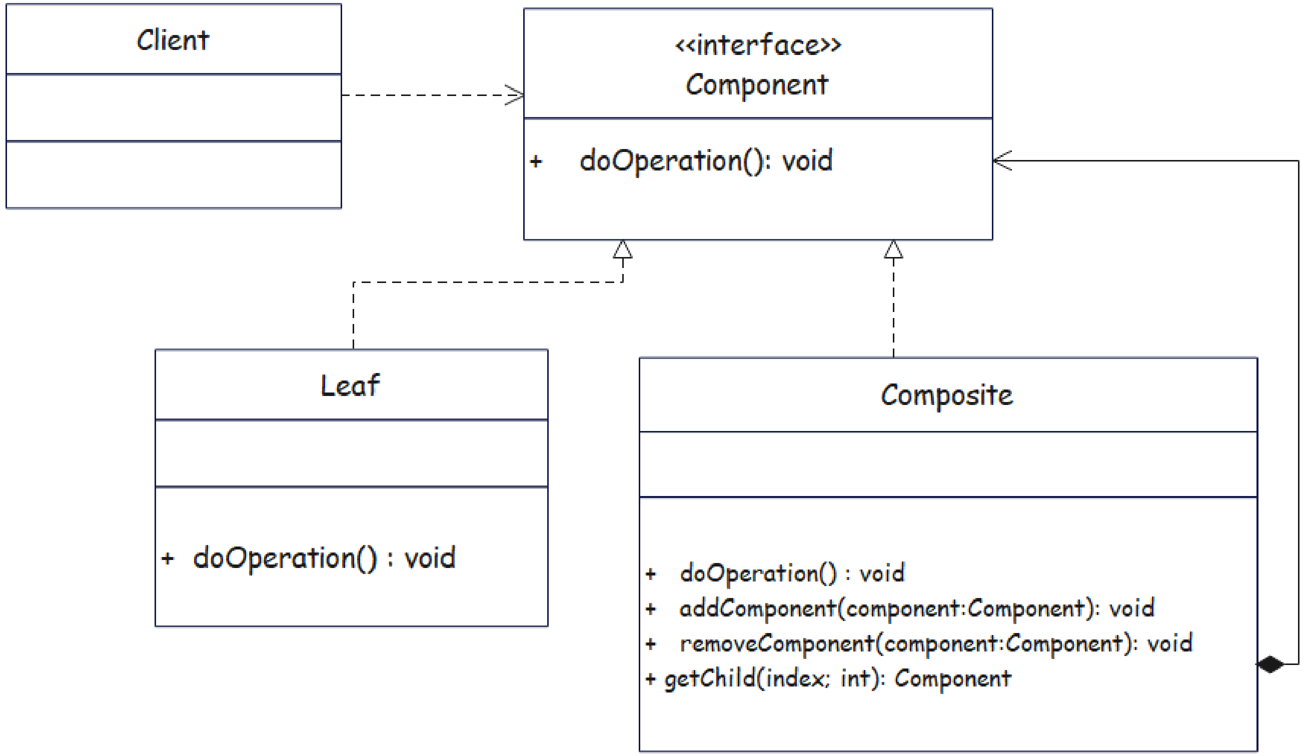
Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a **facade** object that provides a single, simplified interface to the more general facilities of a subsystem.

Applicability

Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem.
- there are many dependencies between clients and the implementation classes of an abstraction.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

* Class Diagram


Client
<<interface>>
Component
+ doOperation(): void
Leaf
+ doOperation() : void
Composite
+ doOperation() : void
+ addComponent(component:Component): void
+ removeComponent(component:Component): void
+ getChild(index: int): Component

Participants

Component (Root)

- declares the interface for objects in the composition.
- implements default behavior for the interface common to all classes, as appropriate.
- declares an interface for accessing and managing its child components.
- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

Leaf

- represents leaf objects in the composition. A leaf has no children.
- defines behavior for primitive objects in the composition.

Composite (Branch)

- defines behavior for components having children.
- stores child components.
- implements child-related operations in the Component interface.

Client

- manipulates objects in the composition through the Component interface.

Collaborations

Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences

- defines class hierarchies consisting of primitive objects and composite objects.
- makes the client simple.
- makes it easier to add new kinds of components.
- can make your design overly general.

Implementation

There are many issues to consider when implementing the Composite pattern:

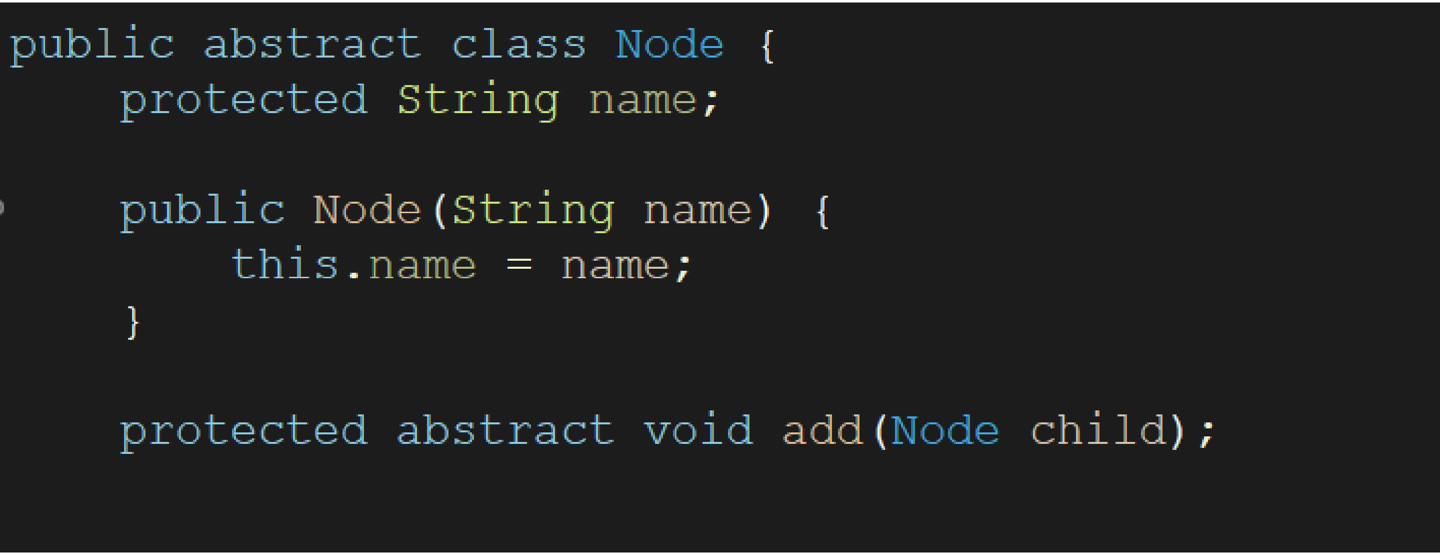
a) *Explicit parent references.*

b) *Sharing components.*

c) *Maximizing the Component interface.*

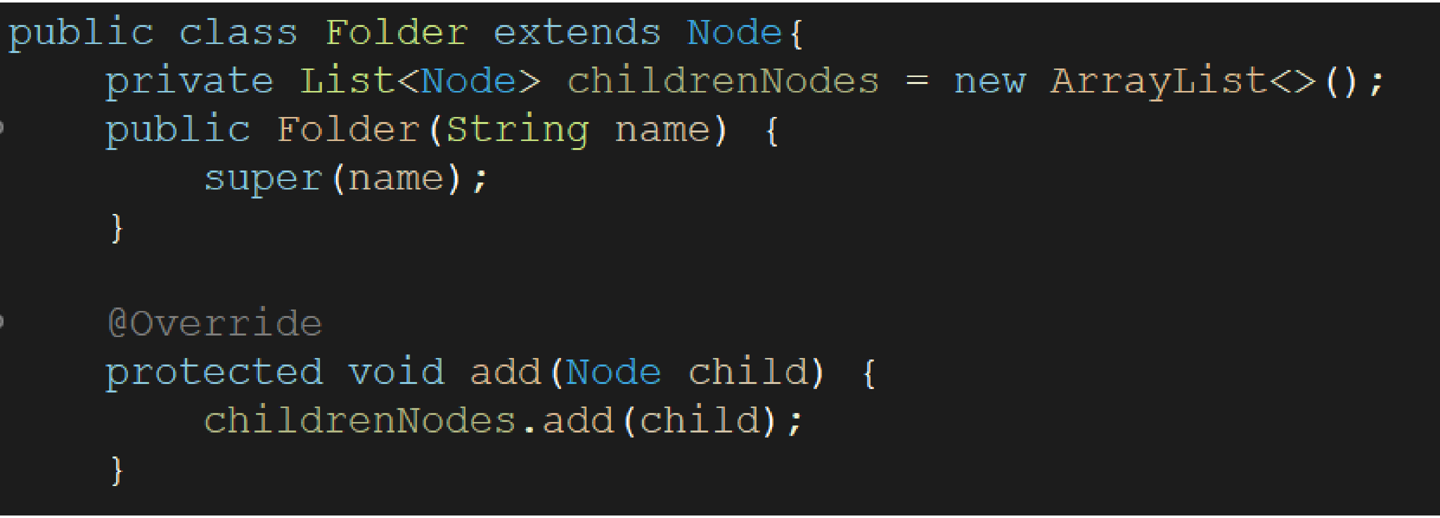
d) *Declaring the child management operations.*

Course: Soft Design

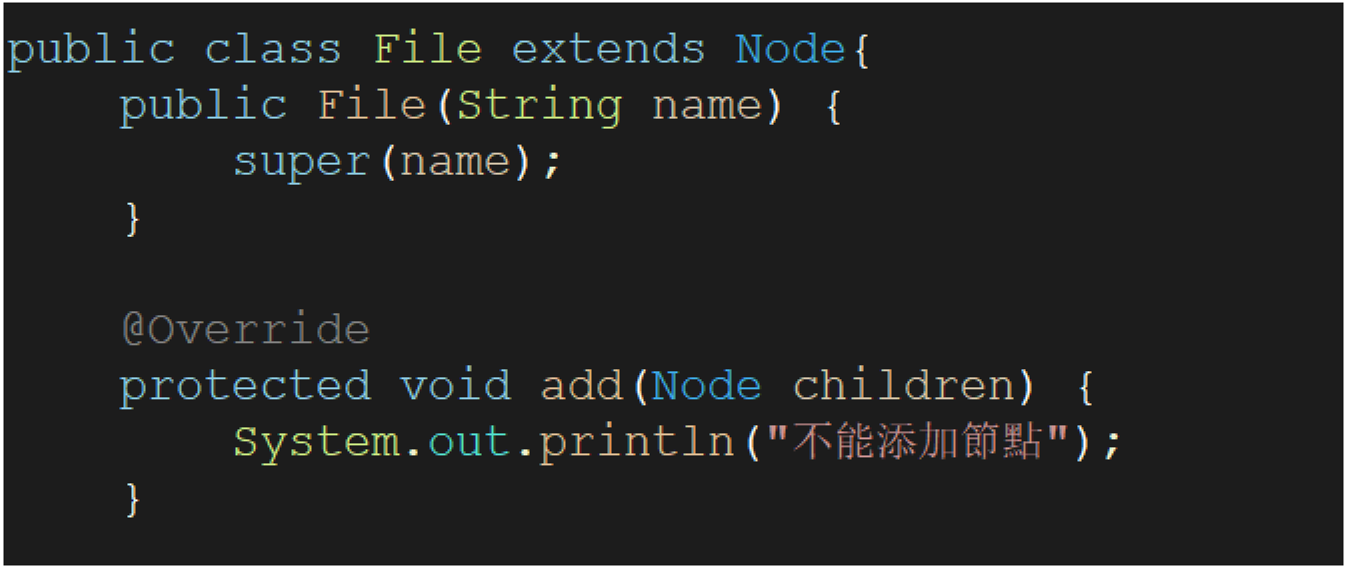


```java
public abstract class Node {
    protected String name;

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

    protected abstract void add(Node child);
```

Course: Soft Design

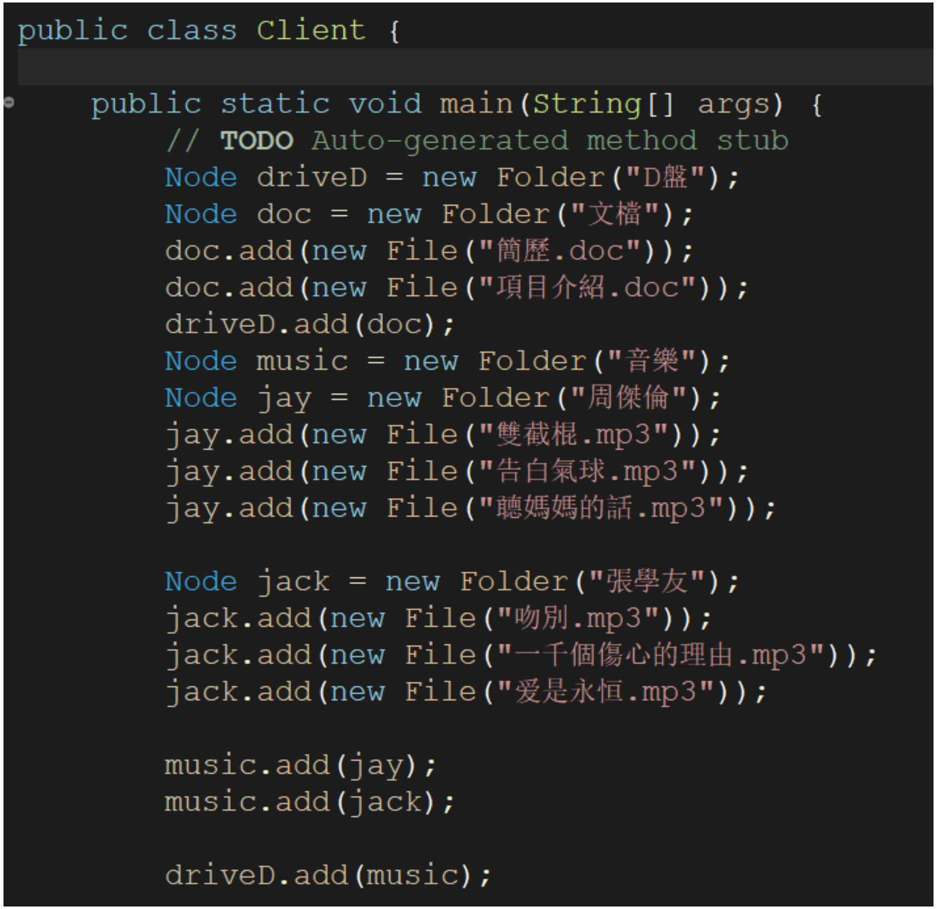


```java
public class Folder extends Node{
    private List<Node> childrenNodes = new ArrayList<>();
    public Folder(String name) {
        super(name);
    }

    @Override
    protected void add(Node child) {
        childrenNodes.add(child);
    }
```

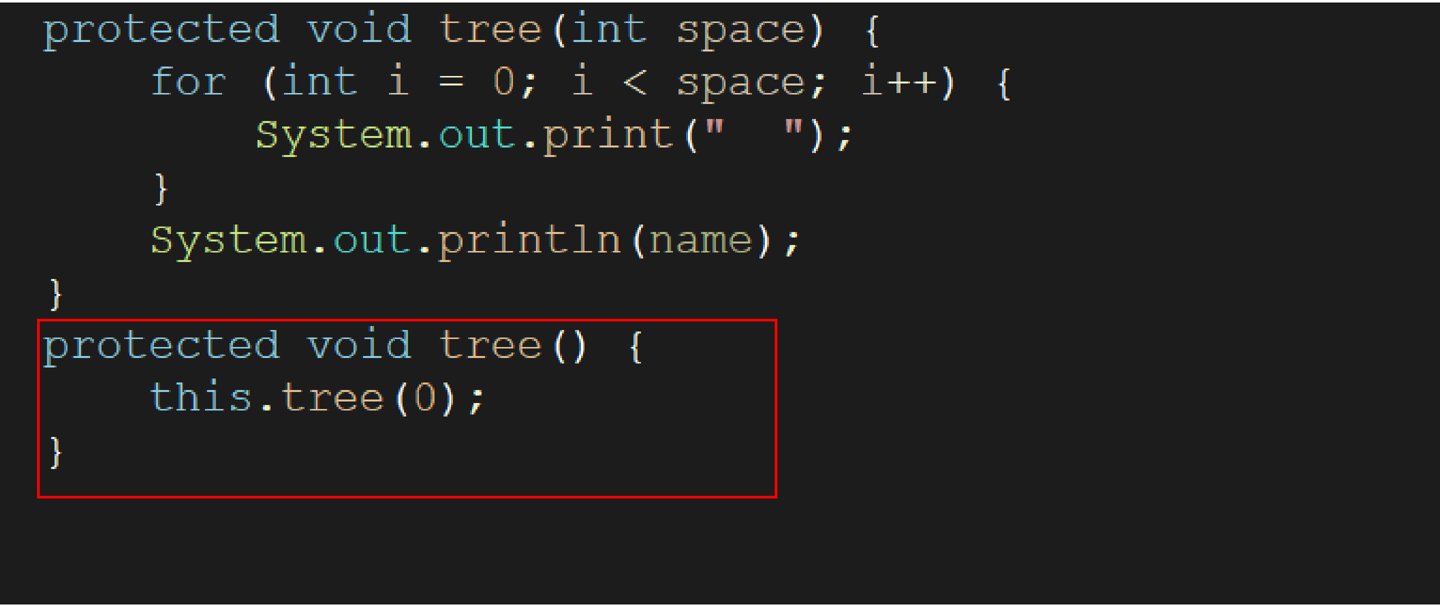


```
public class File extends Node{
    public File(String name) {
        super(name);
    }

    @Override
    protected void add(Node children) {
        System.out.println("不能添加節點");
    }
```

Course: Soft Design

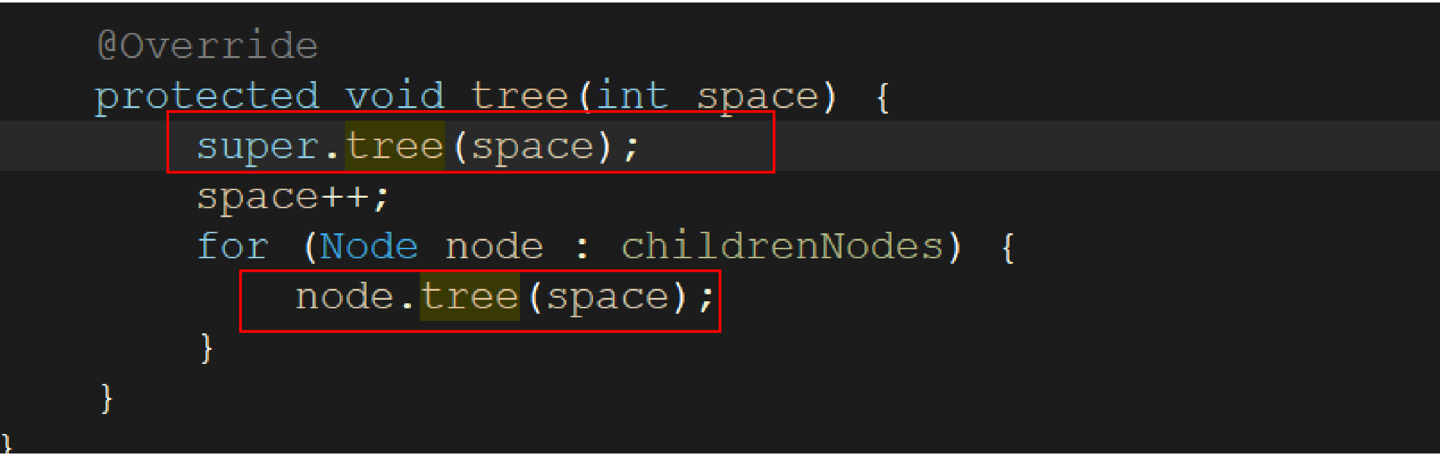


```
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Node driveD = new Folder("D盤");
        Node doc = new Folder("文檔");
        doc.add(new File("簡歷.doc"));
        doc.add(new File("項目介紹.doc"));
        driveD.add(doc);
        Node music = new Folder("音樂");
        Node jay = new Folder("周傑倫");
        jay.add(new File("雙截棍.mp3"));
        jay.add(new File("告白氣球.mp3"));
        jay.add(new File("聽媽媽的話.mp3"));

        Node jack = new Folder("張學友");
        jack.add(new File("吻別.mp3"));
        jack.add(new File("一千個傷心的理由.mp3"));
        jack.add(new File("爱是永恒.mp3"));

        music.add(jay);
        music.add(jack);

        driveD.add(music);
```

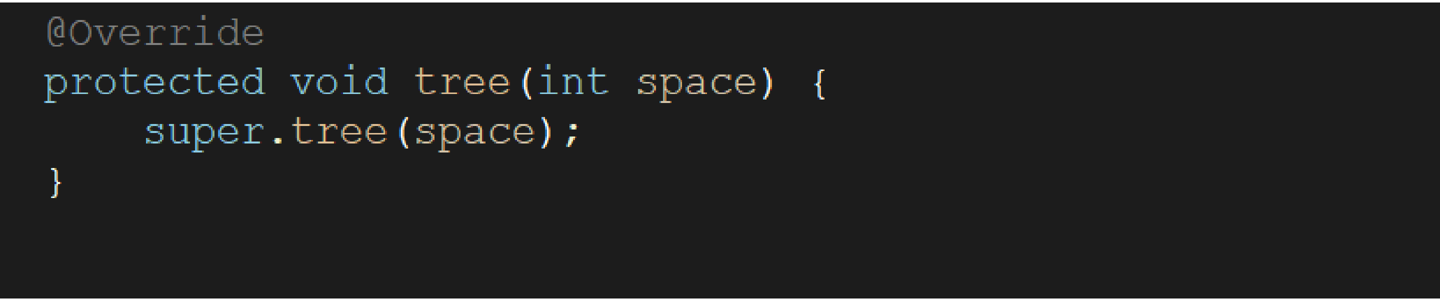


```
protected void tree(int space) {
    for (int i = 0; i < space; i++) {
        System.out.print("   ");
    }
    System.out.println(name);
}
protected void tree() {
    this.tree(0);
}
```

Course: Soft Design

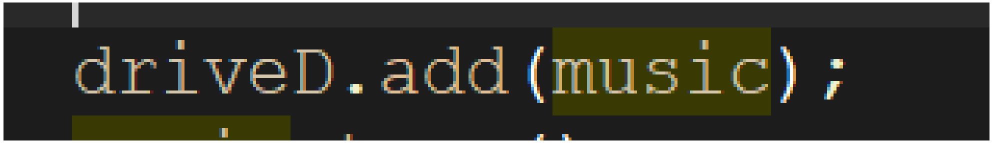


```
    @Override
    protected void tree(int space) {
        super.tree(space);
        space++;
        for (Node node : childrenNodes) {
            node.tree(space);
        }
    }
}
```

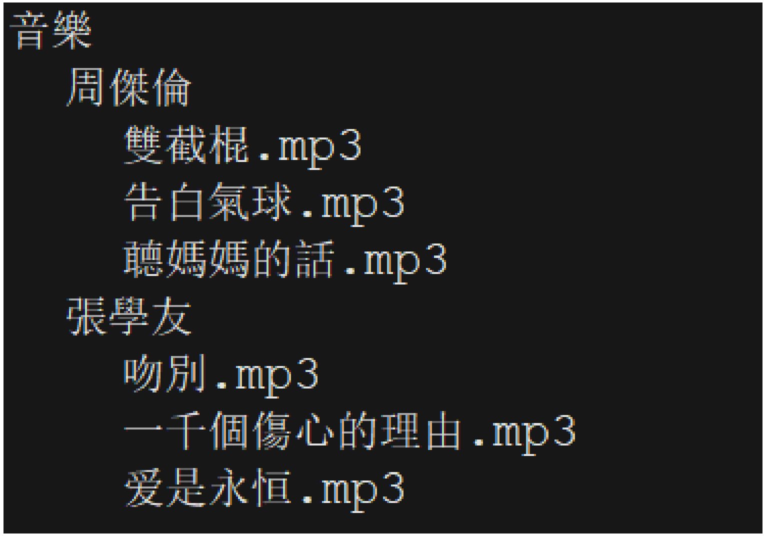


```
@Override
protected void tree(int space) {
    super.tree(space);
}
```

Course: Soft Design



```
driveD.add(music);
```



```
音樂
    周傑倫
        雙截棍.mp3
        告白氣球.mp3
        聽媽媽的話.mp3
    張學友
        吻別.mp3
        一千個傷心的理由.mp3
        爱是永恒.mp3
```

Decorator (裝飾模式)

Intent

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

Also known as

Wrapper

Motivation

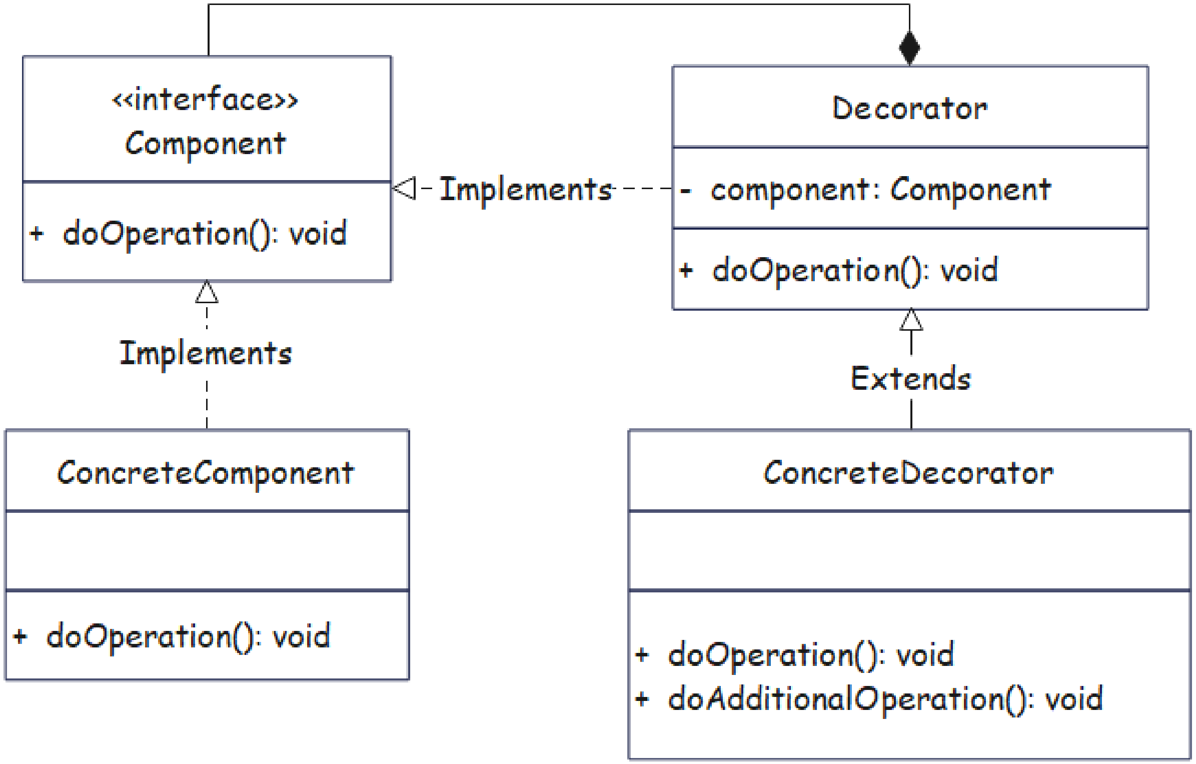
Sometimes we want to add responsibilities to individual objects, not to an entire class.

Applicability

Use the Decorator pattern when

- to add responsibilities to individual objects dynamically and transparently,
- that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by subclassing is impractical.

* Class Diagram



Participants

Component (VisualComponent)
defines the interface for objects that can have responsibilities added to them dynamically.

ConcreteComponent (TextView)
defines an object to which additional responsibilities can be attached.

Decorator
maintains a reference to a Component object and defines an interface that conforms to Component's interface.

ConcreteDecorator (Border/Scroll)
adds responsibilities to the component.

Collaborations

Decorator forwards requests to its Component object. It may optionally perform additional operations before and after forwarding the request.

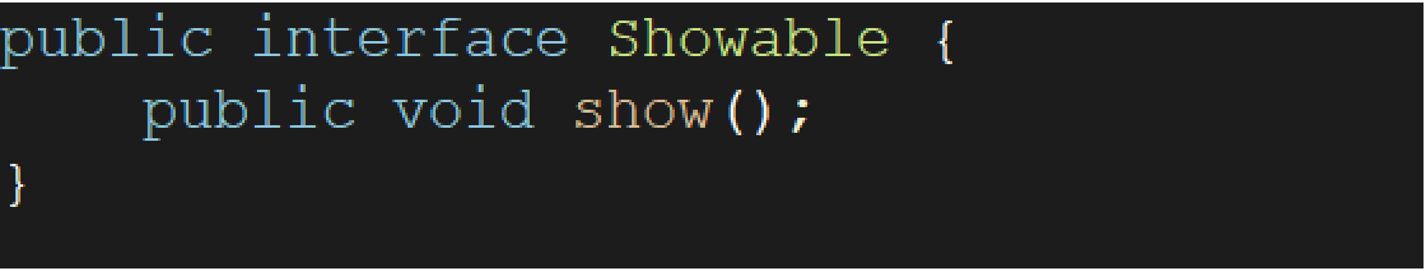
Consequences

1. More flexibility than static inheritance.
2. Avoids feature-laden classes high up in the hierarchy.
3. A decorator and its component aren't identical.
4. Lots of little objects.

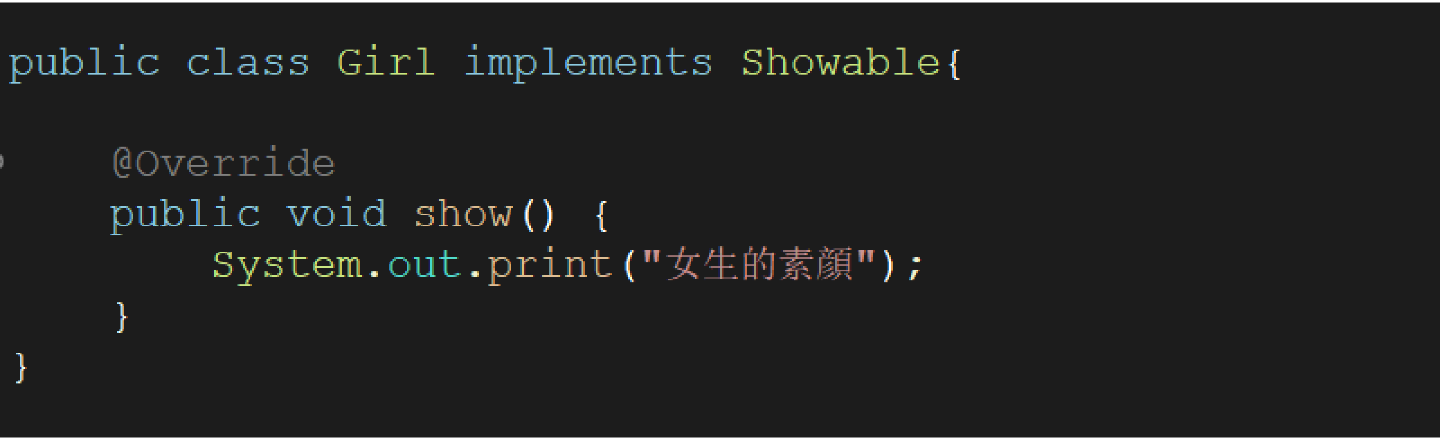
Implementation

- *Interface conformance.*
- *Omitting the abstract Decorator class. T*
- *Keeping Component classes lightweight.*
- *Changing the skin of an object versus changing its guts.*

Course: Soft Design

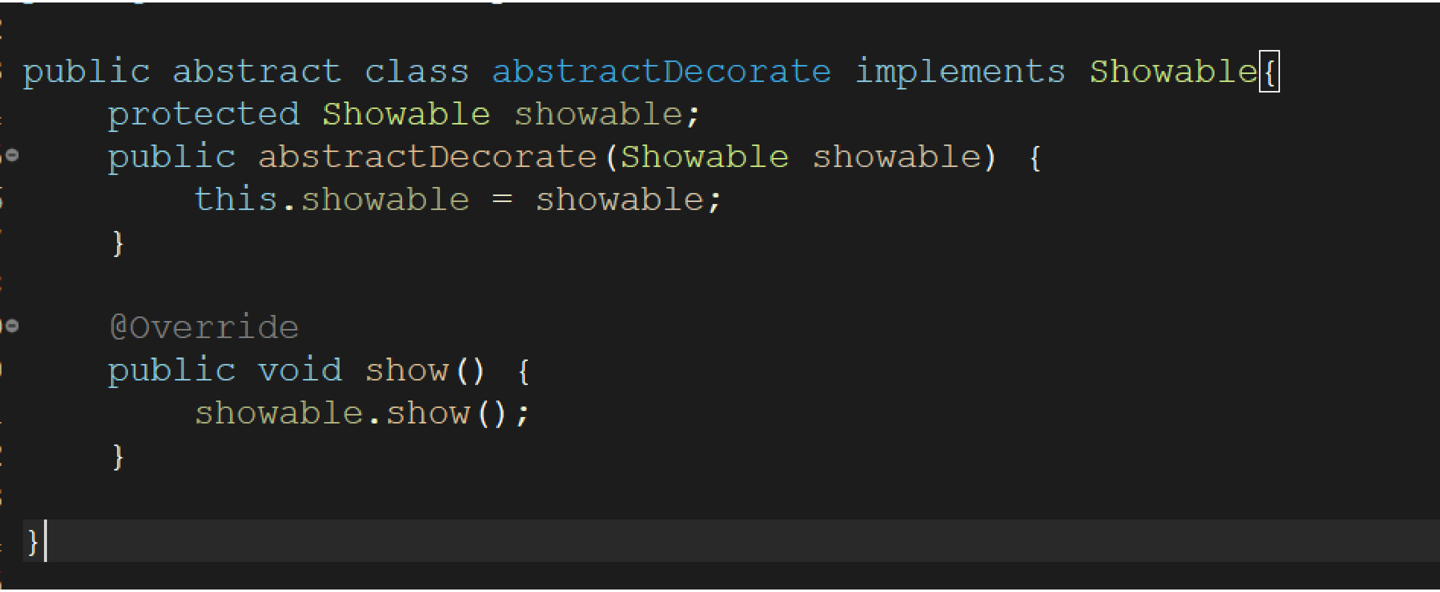


```java
public interface Showable {
    public void show();
}
```



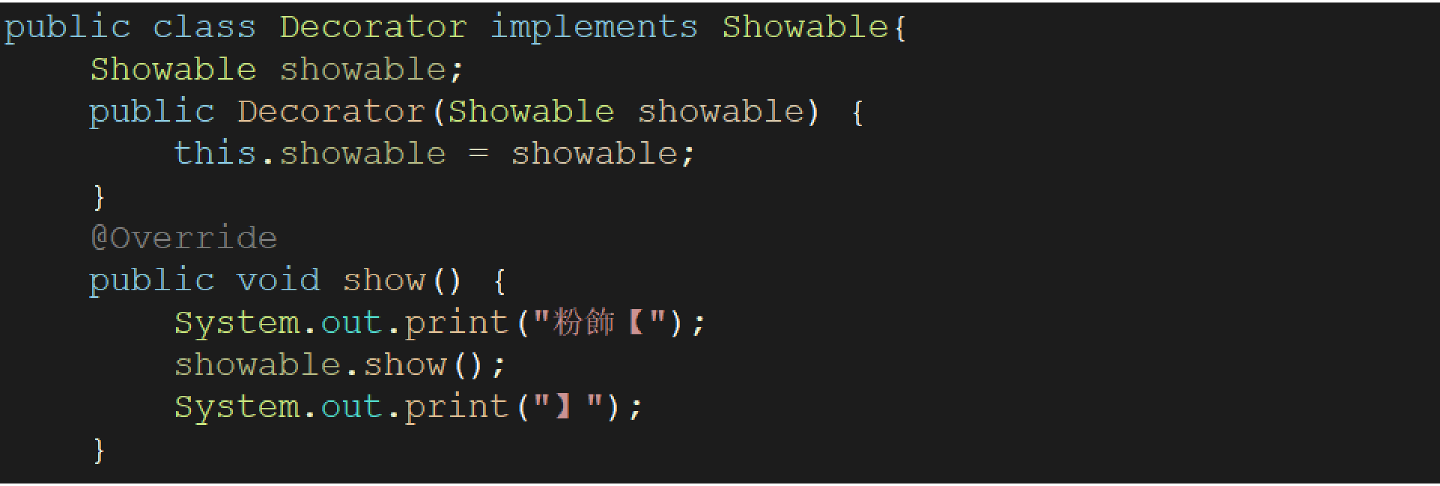
```java
public class Girl implements Showable{

    @Override
    public void show() {
        System.out.print("女生的素顏");
    }
}
```

Course: Soft Design

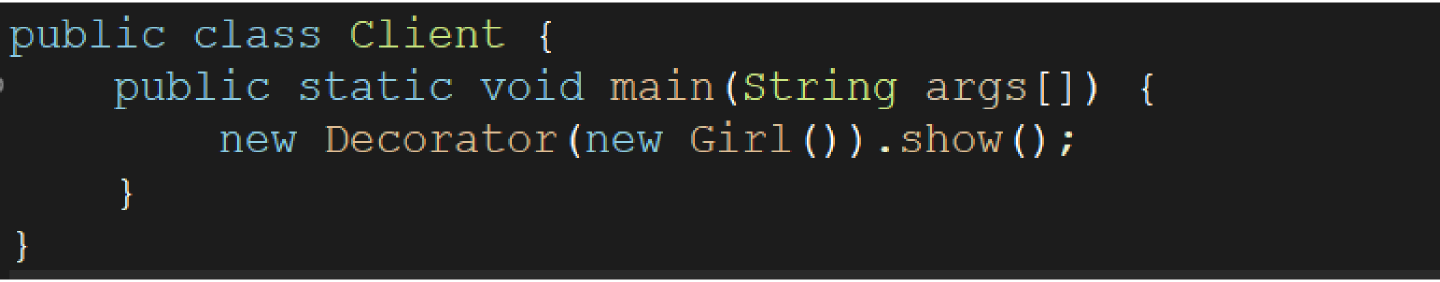


```
public abstract class abstractDecorate implements Showable{
    protected Showable showable;
    public abstractDecorate(Showable showable) {
        this.showable = showable;
    }

    @Override
    public void show() {
        showable.show();
    }

}
```

Course: Soft Design

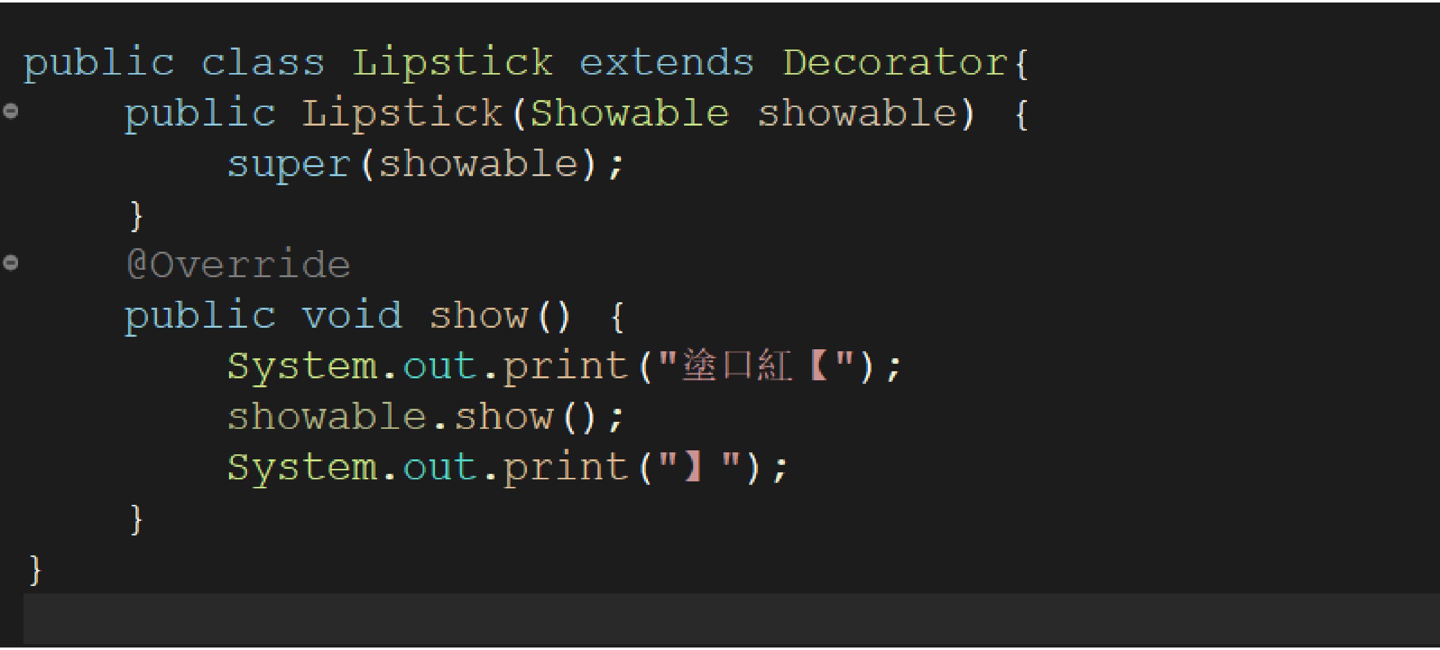


```java
public class Decorator implements Showable{
    Showable showable;
    public Decorator(Showable showable) {
        this.showable = showable;
    }
    @Override
    public void show() {
        System.out.print("粉飾【");
        showable.show();
        System.out.print("】");
    }
```



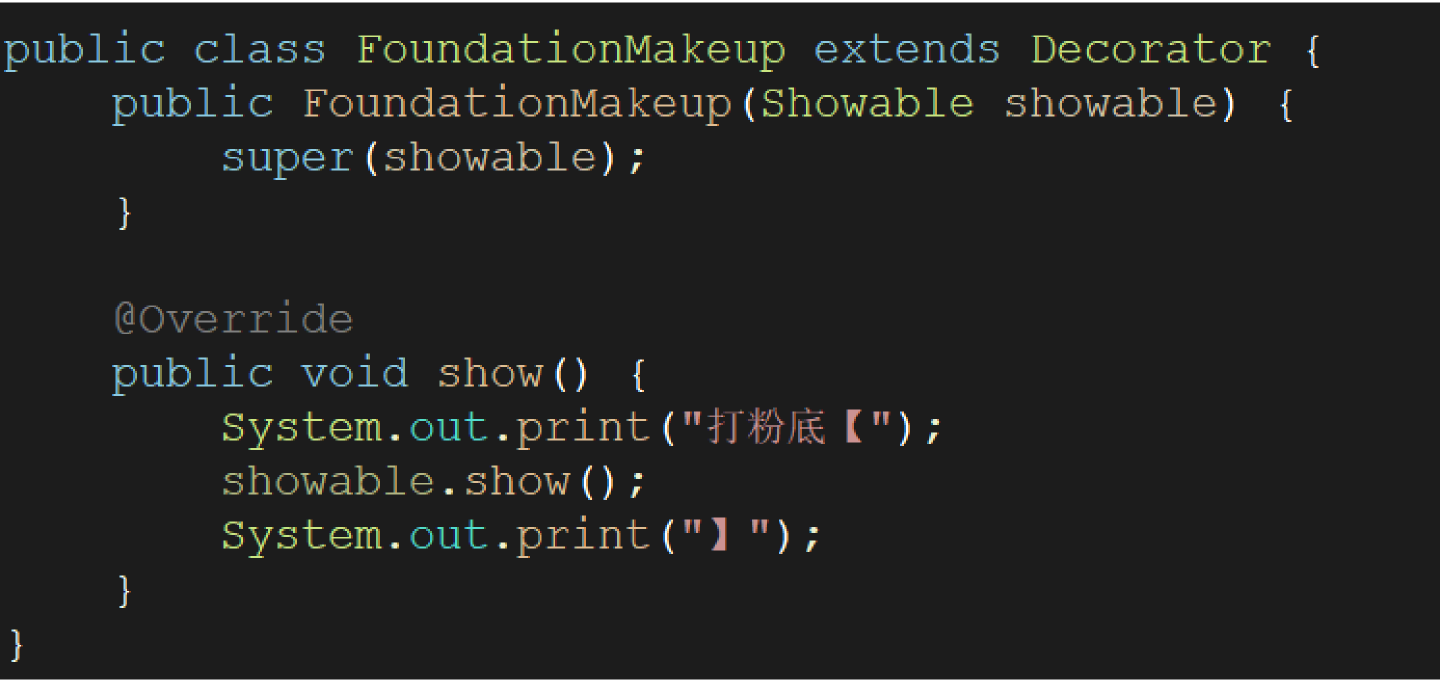
```java
public class Client {
    public static void main(String args[]) {
        new Decorator(new Girl()).show();
    }
}
```

Course: Soft Design



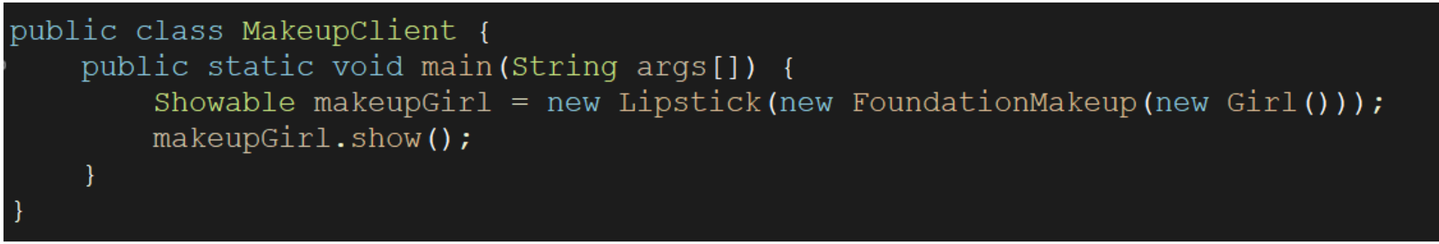
```java
public class Lipstick extends Decorator{
    public Lipstick(Showable showable) {
        super(showable);
    }
    @Override
    public void show() {
        System.out.print("塗口紅【");
        showable.show();
        System.out.print("】");
    }
}
```

Course: Soft Design

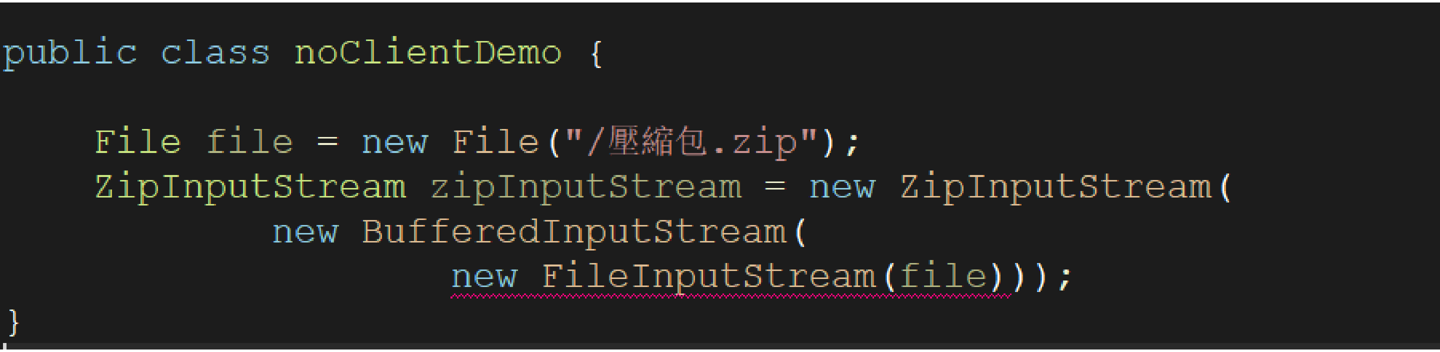


```java
public class FoundationMakeup extends Decorator {
    public FoundationMakeup(Showable showable) {
        super(showable);
    }

    @Override
    public void show() {
        System.out.print("打粉底【");
        showable.show();
        System.out.print("】");
    }
}
```

Course: Soft Design



```java
public class MakeupClient {
    public static void main(String args[]) {
        Showable makeupGirl = new Lipstick(new FoundationMakeup(new Girl()));
        makeupGirl.show();
    }
}
```



```java
public class noClientDemo {

    File file = new File("/壓縮包.zip");
    ZipInputStream zipInputStream = new ZipInputStream(
            new BufferedInputStream(
                    new FileInputStream(file)));
}
```

Principle 58

Specify when environment violates "acceptable" behavior

Requirements specifications often define characteristics of the system's environment. This information is used in making intelligent design decisions. It also often implies that the developer is contractually obligated to accommodate such characteristics. What happens after deployment when the environment exceeds the specified limits?

Principle 59

Self-destruct tbd's

It is often preached that a requirements specification should contain no TBDs(To Be Determined). Obviously, a specification with a TBD is not complete, but there may be very good reasons for approving and perhaps base lining the document with the TBD. This is particularly true for requirements whose precision are not critical to fundamental design decisions. When you create a TBD, be sure to footnote it with a "self-destruction note," that is, specify who will resolve the TBD and by when.

Principle 60

Store requirements in a database

Requirements are complex and highly volatile. For these reasons, storing them in electronic media, preferably a database, is a good idea. This will facilitate making changes, finding implications of changes, recording attributes of specific requirements,and so on.

DESIGN PRINCIPLES

Design is the set of activities including (1) defining an architecture for the software that satisfies the requirements and (2)specifying an algorithm for each software component in the architecture. The architecture includes a specification of all the building blocks of the software, how they interface with each other, how they are composed of one another, and how copies of components are instantiated (that is, copies made in memory of components and executed) and destroyed. The final product of design is a design specification.

Principle 61

Transition from requirements to design is not easy

Since design is difficult there are three possibilities:

1. No thought went into selecting an optimal design during requirements. In this case, you cannot afford to accept the design as the design.
2. 2. Alternative designs were enumerated and analyzed and best selected, all during requirements. Organizations cannot afford the effort to do a thorough design (typically 30 to 40 percent of total development costs) prior to base lining requirements,making a make/buy decision, and making a development cost estimate.
3. 3. The method assumes that some architecture is optimal for all applications. This is clearly not possible.

Principle 62

Trace design to requirements

When designing software, the designer must know which requirements are being satisfied by each component. All these needs can be satisfied by the creation of a large binary table with rows corresponding to all software components and columns corresponding to every requirement in the SRS. Al in any position indicates that this design component helps to satisfy this requirement.

Principle 63

Evaluate alternatives

A critical aspect of all engineering disciplines is the elaboration of multiple approaches, trade-off analyses among them, and the eventual adoption of one. After requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use an architecture simply because it was used in the requirements specification (Principle 46). After all, that architecture was selected to optimize understand ability of the system's external behavior. The architecture you want is the one that optimizes conformance with the requirements contained in the requirements specification.

Principle 64

Design without documentation is not design

I have often heard software engineers say, "I have finished the design. All that's left is its documentation." This makes no sense. Can you imagine a building architect saying, "I have completed the design of your new home. All that's left is to draw a picture of it," or a novelist saying" I have completed the novel. All that's left is to write it"? Design is the selection, abstraction, and recording of an appropriate architecture and algorithm onto paper or other medium.

Principle 65

Encapsulate

Information hiding is a simple, proven concept that results in software that is easier to test and easier still to maintain. Most software modules should hide some information from all other software. This information could be the structure of data, the contents of data, an algorithm, a design decision, or an interface to hardware, to a user, or to another piece of software.

Principle 66

Don't reinvent the wheel

When electrical engineers design new printed circuit boards, they go to a catalog of available integrated circuits to select the most appropriate components. When electrical engineers design new integrated circuits, they go to a catalog of standard cells. When architects design new homes, they go to catalogs of prefabricated doors, windows, moldings, and other components. All this is called "engineering." Software engineers usually reinvent components over and over again; they rarely salvage existing software components. It is interesting that the software industry calls this rare practice "reuse" rather than" engineering."

Principle 67

Keep it simple

A simple architecture or a simple algorithm goes a long way toward achieving high maintainability. Remember KISS. Also, as you decompose software into subcomponents, remember that a human has difficulty comprehending more than seven (plus or minus two) things at once.

Principle 68

Avoid numerous special cases

As you design your algorithms, you will undoubtedly realize that there are exceptional situations. Exceptional situations cause special cases to be added to your algorithm. Every special case makes it more difficult for you to debug and for others to modify, maintain, and enhance. If you find too many special cases, you probably have an inappropriate algorithm. Rethink and redesign the algorithm. See related Principle 67.

Principle 69

Minimize intellectual distance

Edsger Dijkstra defined intellectual distance as the distance between the real-world problem and the computerized solution to that problem. Richard Fairley argues that the smaller the intellectual distance, the easier it will be to maintain the software.

Principle 70

Keep design under intellectual control

A design is under intellectual control if it has been created and document-ed in a manner that enables its creators and maintainers to fully under-stand it.

Principle 71

Maintain conceptual integrity

Conceptual integrity is an attribute of a quality design. It implies that a limited number of design "forms" are used and that they are used uniformly. Ego satisfaction is not as important as conceptual integrity.

Principle 72

Conceptual errors are more significant than syntactic errors

When creating software, whether writing requirements specifications, design specifications, code, or tests, we spend considerable effort to remove syntactic errors. This is laudable. However, the real difficulty in constructing software arises from conceptual errors. Most developers spend more time looking for and correcting syntactic errors because, when found, these look like silly errors that in some way amuse the developer. In contrast to these, developers often feel in some way flawed, or incompetent, when they locate a conceptual error. No matter how good you are, you will make conceptual errors. Look for them.

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.