LECTURE 05

Adapter and **Flyweight**

Software Development Principle 58~76

Adapter (適配器模式)

Intent

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

Also known as

Wrapper

Motivation

Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

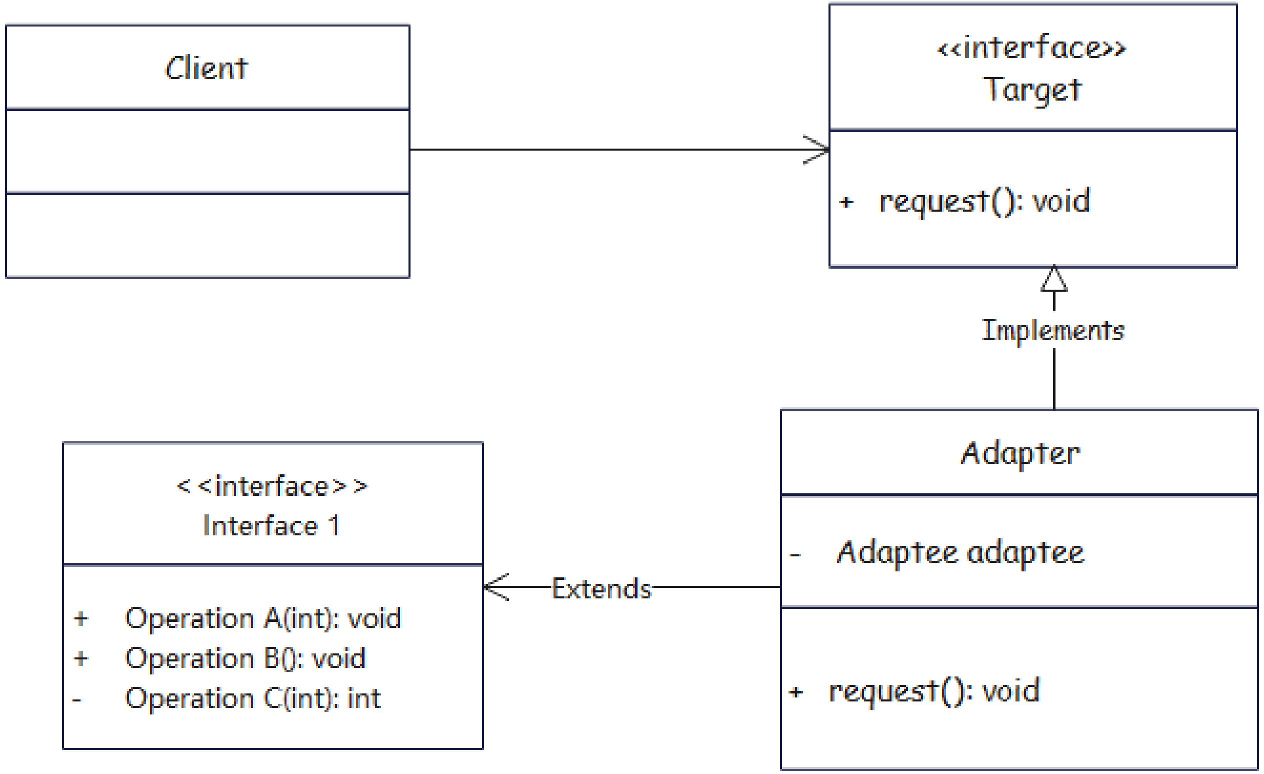
Course: Soft Design

Applicability

Use the Adapter pattern when

- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

* Class Diagram



Participants

Target (Shape)
defines the domain-specific interface that Client uses.

Client (DrawingEditor)
collaborates with objects conforming to the Target interface.

Adaptee (TextView)
defines an existing interface that needs adapting.

Adapter (TextShape)
adapts the interface of Adaptee to the Target interface.

Collaborations

Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

Consequences

Class and object adapters have different trade-offs.

A class adapter

1. adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
2. lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
3. introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

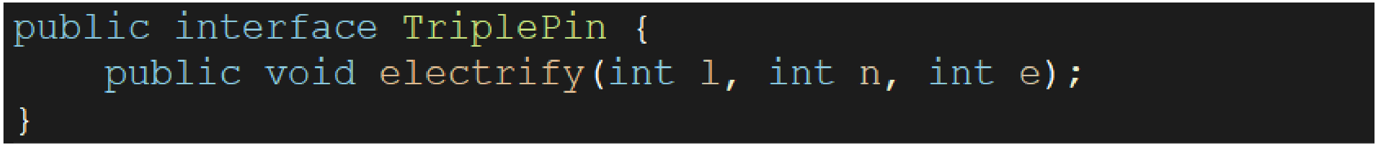
1. lets a single Adapter work with many Adaptees — that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
2. makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Implementation

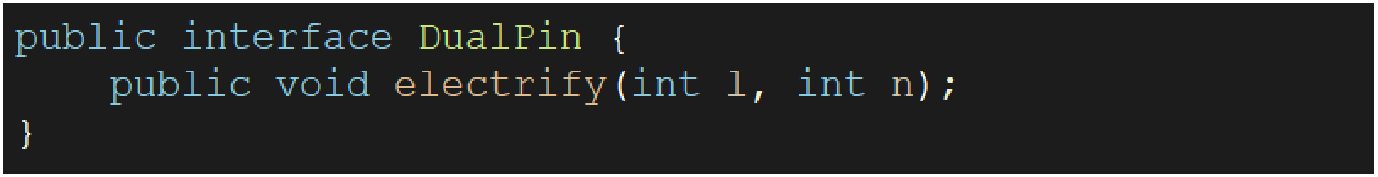
- *How much adapting does Adapter do?*
- *Pluggable adapters.*
- *Using two-way adapters to provide transparency.*

Course: Soft Design

Course: Soft Design

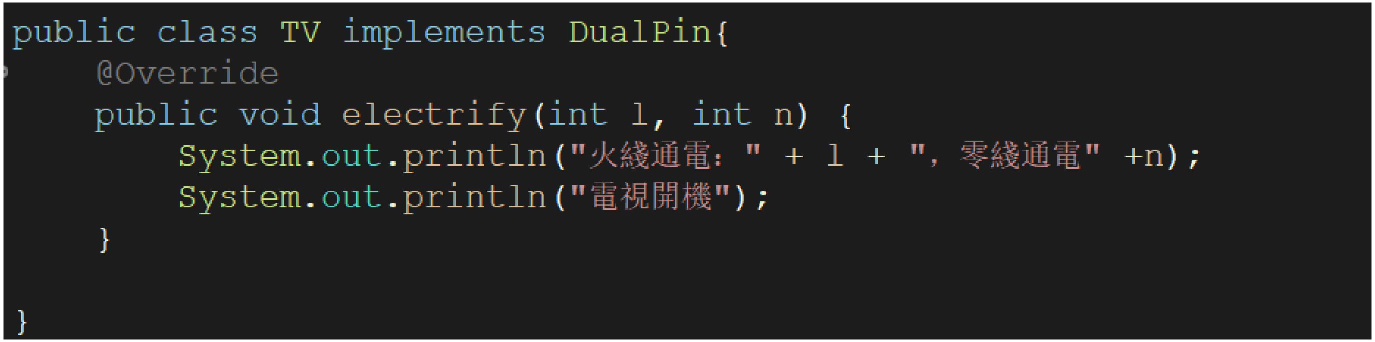


```
public interface TriplePin {
    public void electrify(int l, int n, int e);
}
```

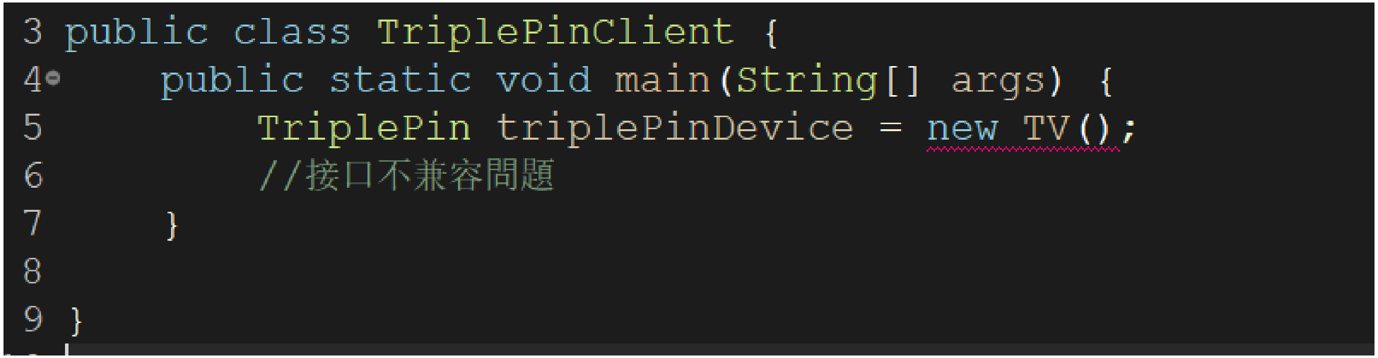


```
public interface DualPin {
    public void electrify(int l, int n);
}
```

Course: Soft Design

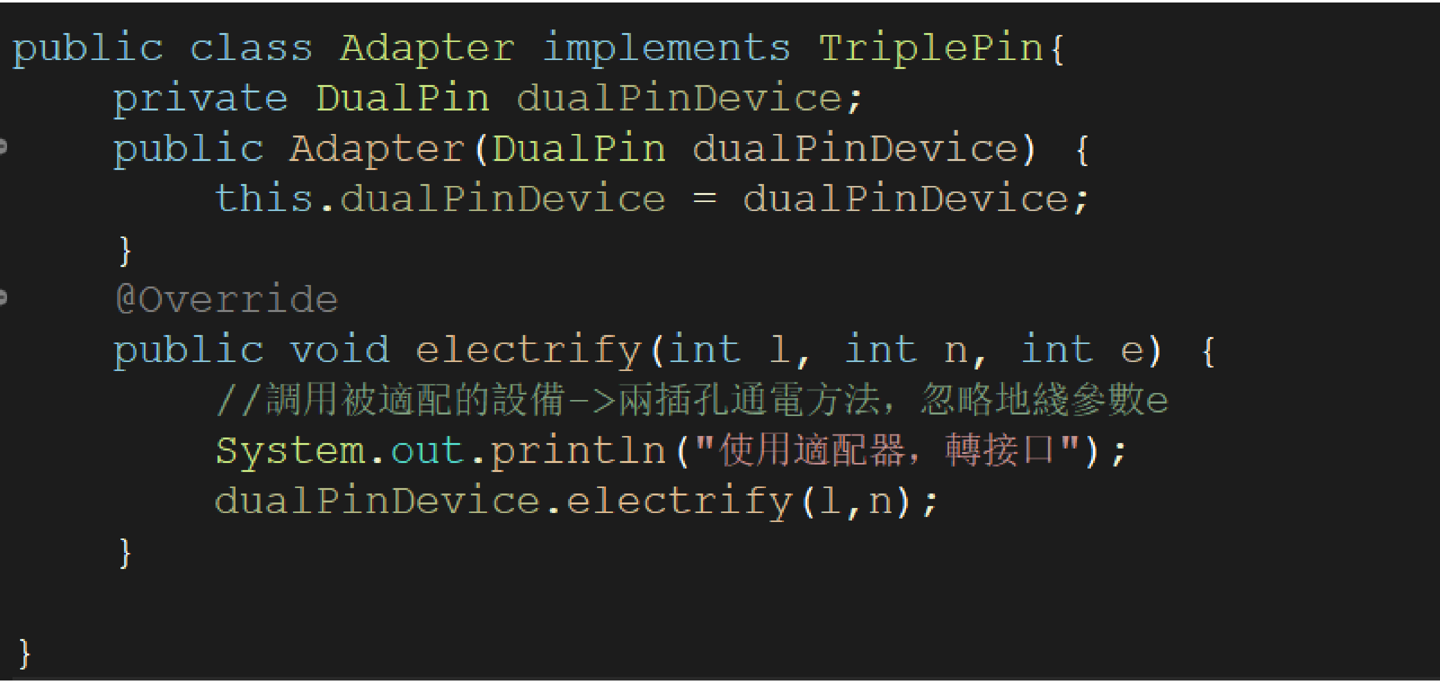


```java
public class TV implements DualPin{
    @Override
    public void electrify(int l, int n) {
        System.out.println("火綫通電: " + l + ", 零綫通電" +n);
        System.out.println("電視開機");
    }

}
```



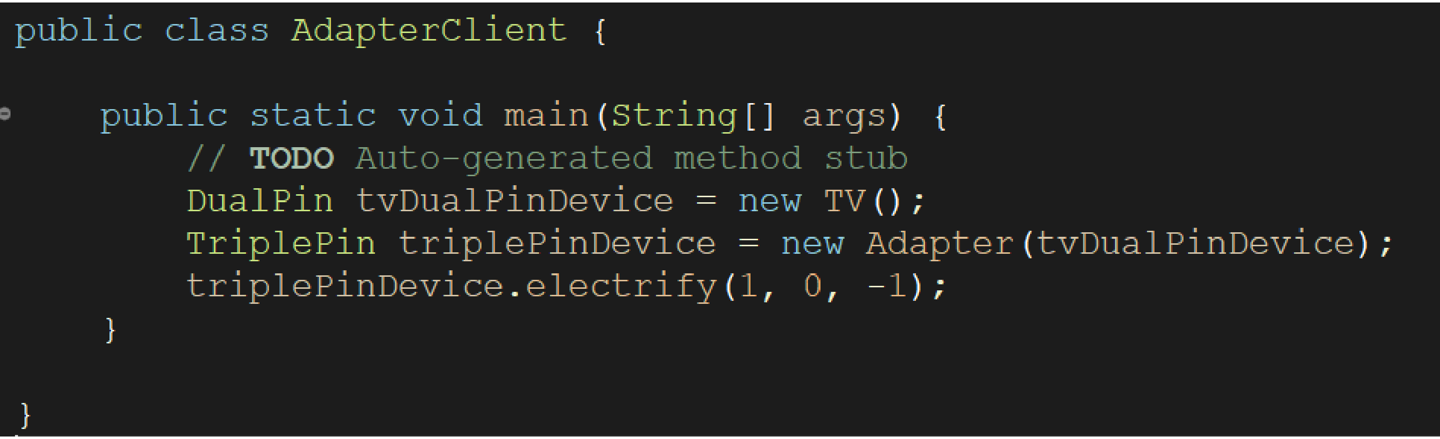
```java
public class TriplePinClient {
    public static void main(String[] args) {
        TriplePin triplePinDevice = new TV();
        //接口不兼容問題
    }

}
```

Course: Soft Design

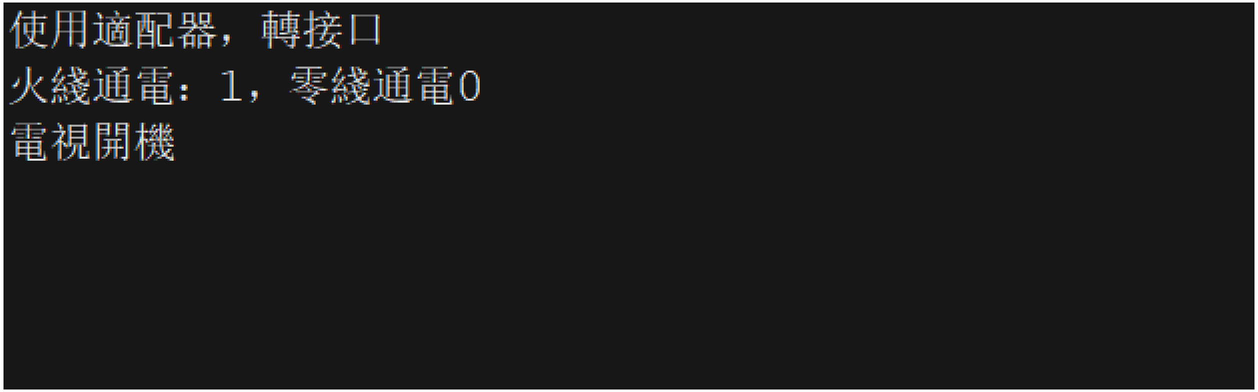


```
public class Adapter implements TriplePin{
    private DualPin dualPinDevice;
    public Adapter(DualPin dualPinDevice) {
        this.dualPinDevice = dualPinDevice;
    }
    @Override
    public void electrify(int l, int n, int e) {
        //調用被適配的設備->兩插孔通電方法，忽略地綫參數e
        System.out.println("使用適配器，轉接口");
        dualPinDevice.electrify(l,n);
    }

}
```

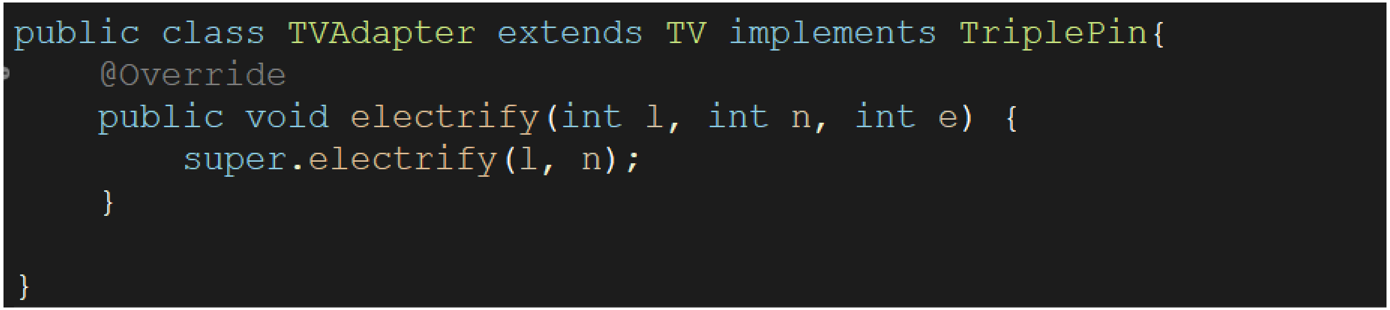
Course: Soft Design



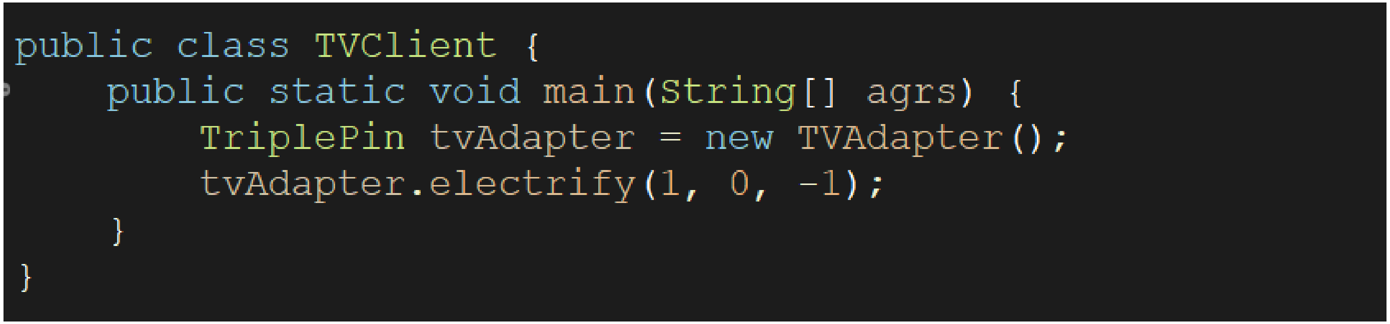
```java
public class AdapterClient {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        DualPin tvDualPinDevice = new TV();
        TriplePin triplePinDevice = new Adapter(tvDualPinDevice);
        triplePinDevice.electrify(1, 0, -1);
    }

}
```



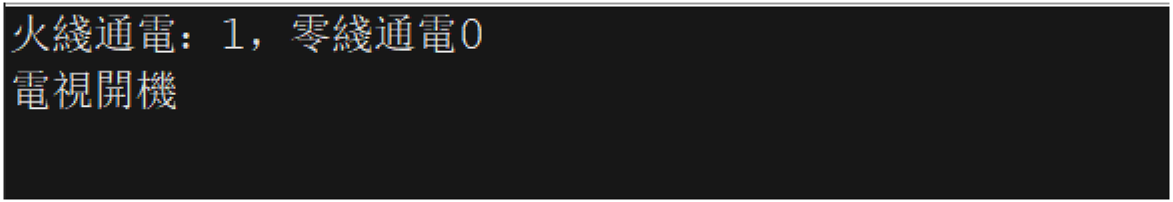
```
使用適配器，轉接口
火綫通電：1，零綫通電0
電視開機
```



```java
public class TVAdapter extends TV implements TriplePin{
    @Override
    public void electrify(int l, int n, int e) {
        super.electrify(l, n);
    }

}
```



```java
public class TVClient {
    public static void main(String[] agrs) {
        TriplePin tvAdapter = new TVAdapter();
        tvAdapter.electrify(1, 0, -1);
    }
}
```



```
火綫通電: 1, 零綫通電0
電視開機
```

Flyweight (享元模式)

Intent

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

Motivation

Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive.

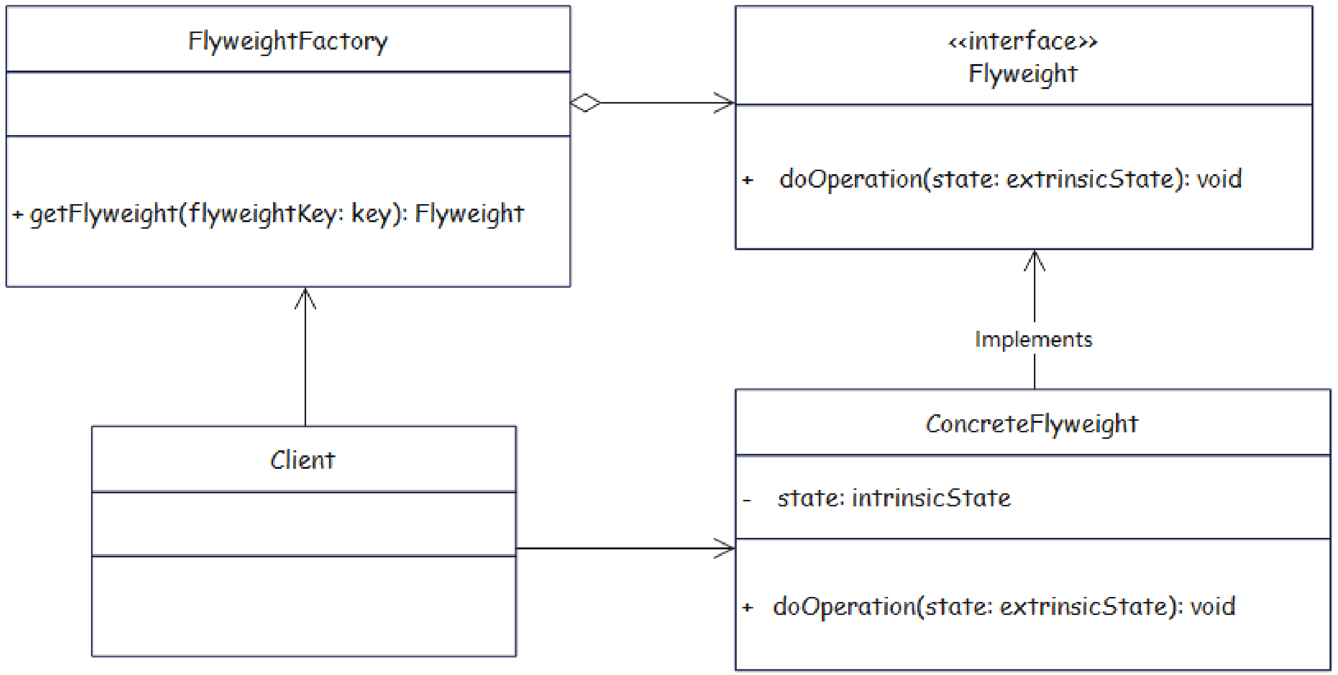
Course: Soft Design

Applicability

Apply the Flyweight pattern when *all* of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.

* Class Diagram


FlyweightFactory
+ getFlyweight(flyweightKey: key): Flyweight
<<interface>>
Flyweight
+ doOperation(state: extrinsicState): void
Implements
Client
ConcreteFlyweight
- state: intrinsicState
+ doOperation(state: extrinsicState): void

Participants

Flyweight(Glyph)
declares an interface through which flyweights can receive and act on extrinsic state.

ConcreteFlyweight(Character)
implements the Flyweight interface and adds storage for intrinsic state, if any.

UnsharedConcreteFlyweight(Row, Column)
not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing; it doesn't enforce it.

FlyweightFactory

creates and manages flyweight objects.
ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.

Client

maintains a reference to flyweight(s).
computes or stores the extrinsic state of flyweight(s).

Collaborations

- State that a flyweight needs to function must be characterized as either intrinsic or extrinsic.
- Clients should not instantiate ConcreteFlyweights directly. Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

Consequences

Storage savings are a function of several factors:

- the reduction in the total number of instances

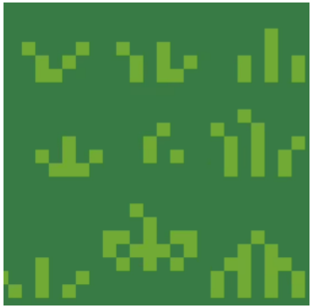
  that comes from sharing
- the amount of intrinsic state per object
- whether extrinsic state is computed or stored.

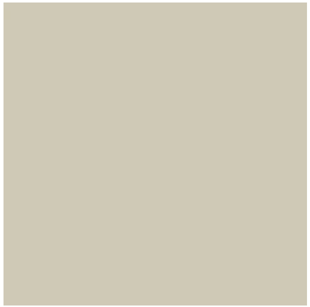
Implementation

- *Removing extrinsic state.*
- *Managing shared objects.*

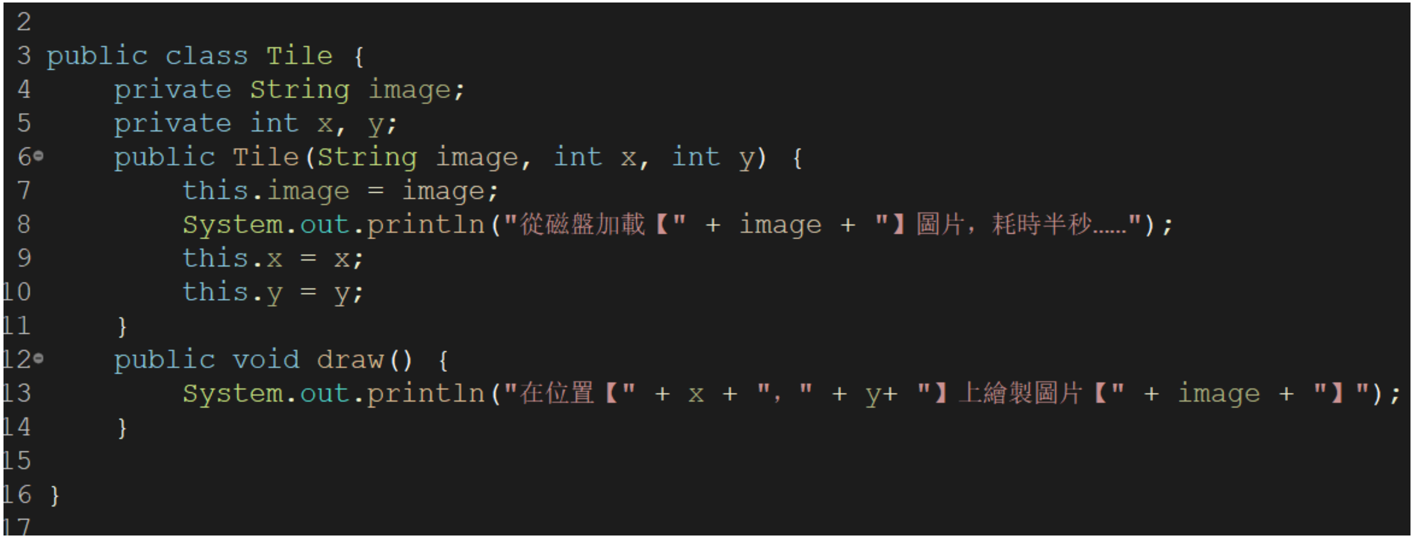
Course: Soft Design

River

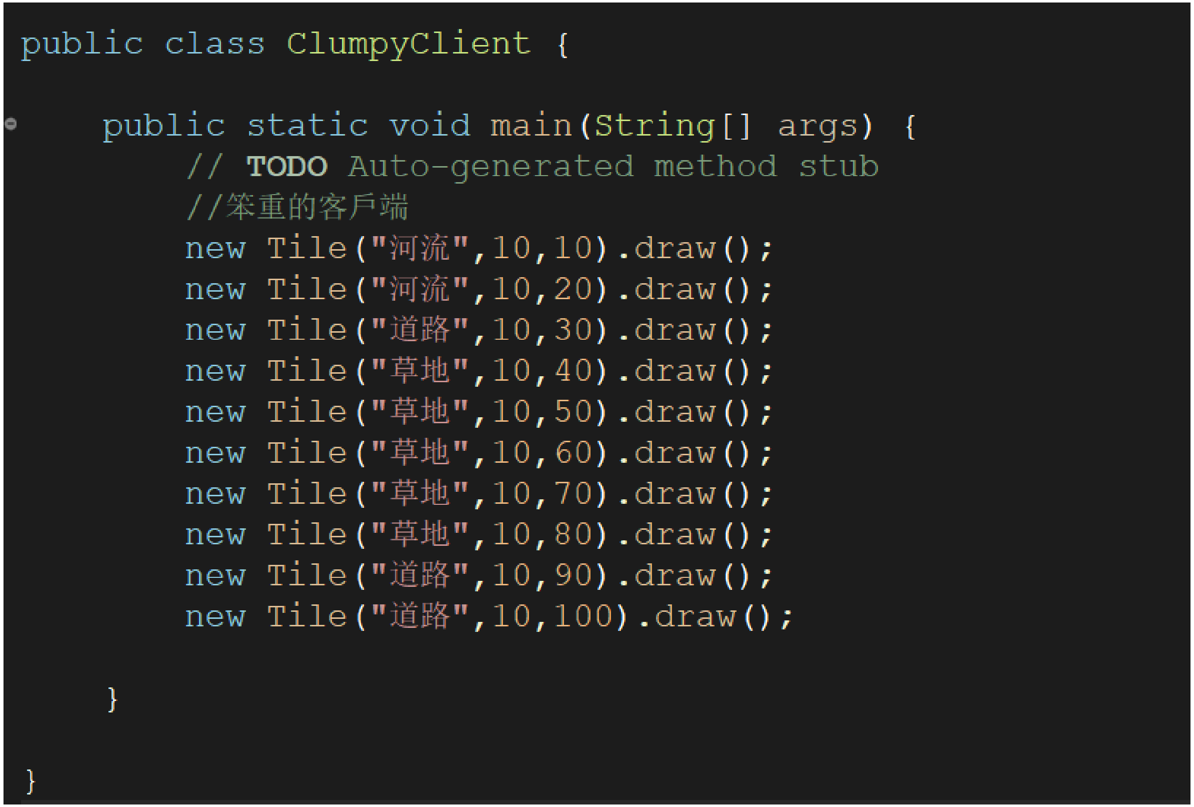

Grass


Road

House

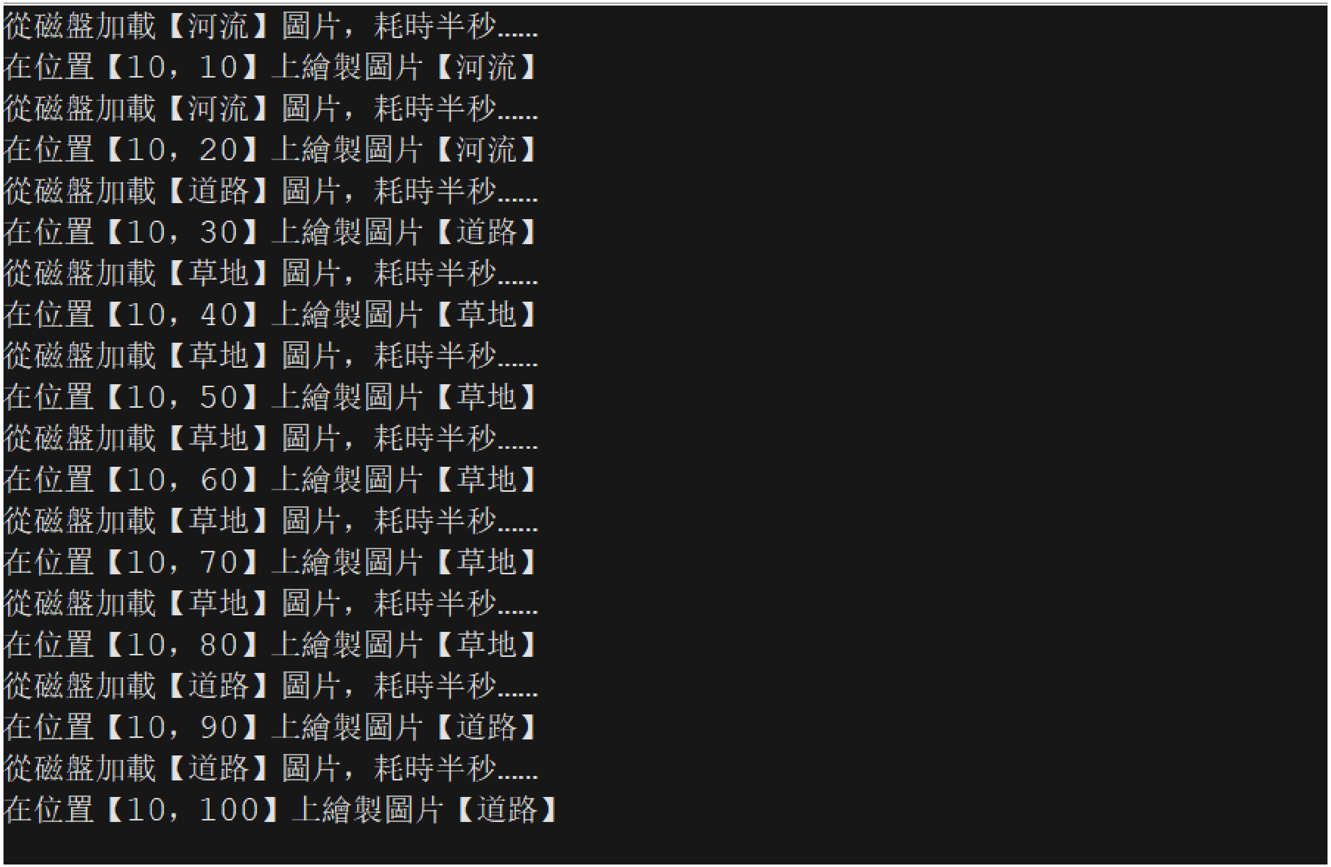


```
public class Tile {
    private String image;
    private int x, y;
    public Tile(String image, int x, int y) {
        this.image = image;
        System.out.println("從磁盤加載【" + image + "】圖片，耗時半秒……");
        this.x = x;
        this.y = y;
    }
    public void draw() {
        System.out.println("在位置【" + x + ", " + y+ "】上繪製圖片【" + image + "】");
    }

}
```



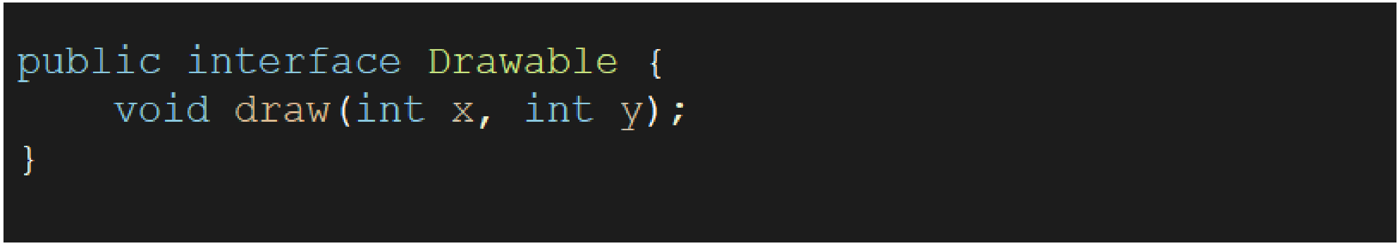
```
public class ClumpyClient {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //笨重的客戶端
        new Tile("河流",10,10).draw();
        new Tile("河流",10,20).draw();
        new Tile("道路",10,30).draw();
        new Tile("草地",10,40).draw();
        new Tile("草地",10,50).draw();
        new Tile("草地",10,60).draw();
        new Tile("草地",10,70).draw();
        new Tile("草地",10,80).draw();
        new Tile("道路",10,90).draw();
        new Tile("道路",10,100).draw();

    }

}
```

Course: Soft Design

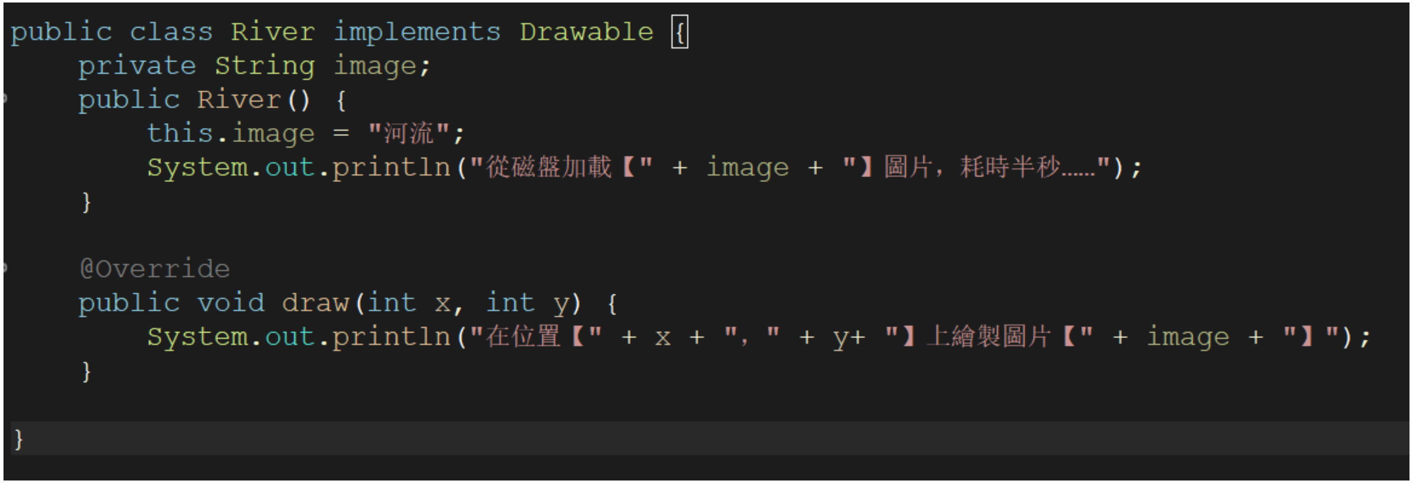


```
從磁盤加載【河流】圖片，耗時半秒......
在位置【10，10】上繪製圖片【河流】
從磁盤加載【河流】圖片，耗時半秒......
在位置【10，20】上繪製圖片【河流】
從磁盤加載【道路】圖片，耗時半秒......
在位置【10，30】上繪製圖片【道路】
從磁盤加載【草地】圖片，耗時半秒......
在位置【10，40】上繪製圖片【草地】
從磁盤加載【草地】圖片，耗時半秒......
在位置【10，50】上繪製圖片【草地】
從磁盤加載【草地】圖片，耗時半秒......
在位置【10，60】上繪製圖片【草地】
從磁盤加載【草地】圖片，耗時半秒......
在位置【10，70】上繪製圖片【草地】
從磁盤加載【草地】圖片，耗時半秒......
在位置【10，80】上繪製圖片【草地】
從磁盤加載【道路】圖片，耗時半秒......
在位置【10，90】上繪製圖片【道路】
從磁盤加載【道路】圖片，耗時半秒......
在位置【10，100】上繪製圖片【道路】
```

Course: Soft Design

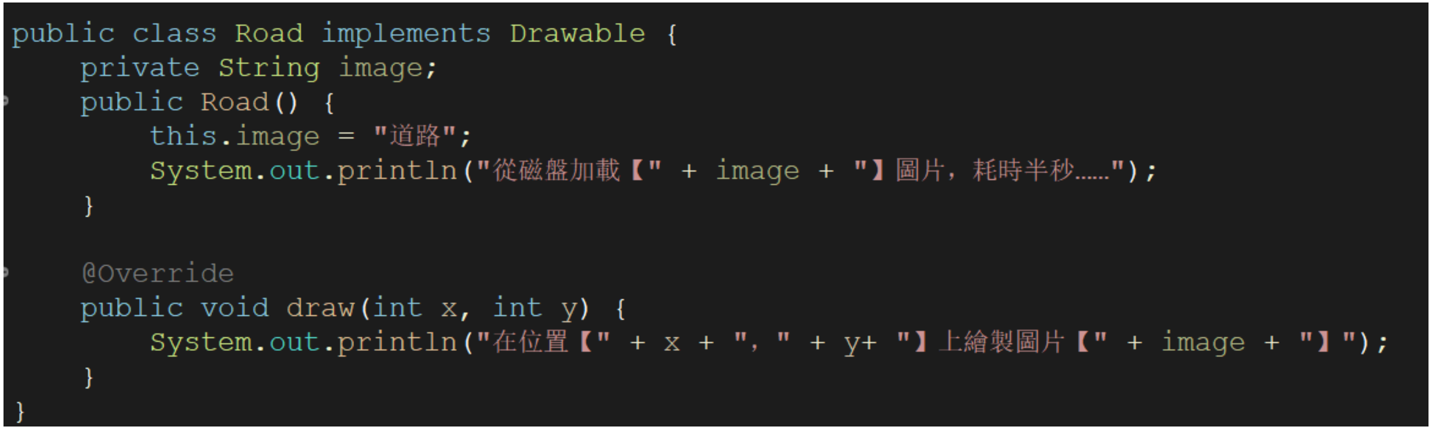


```java
public interface Drawable {
    void draw(int x, int y);
}
```

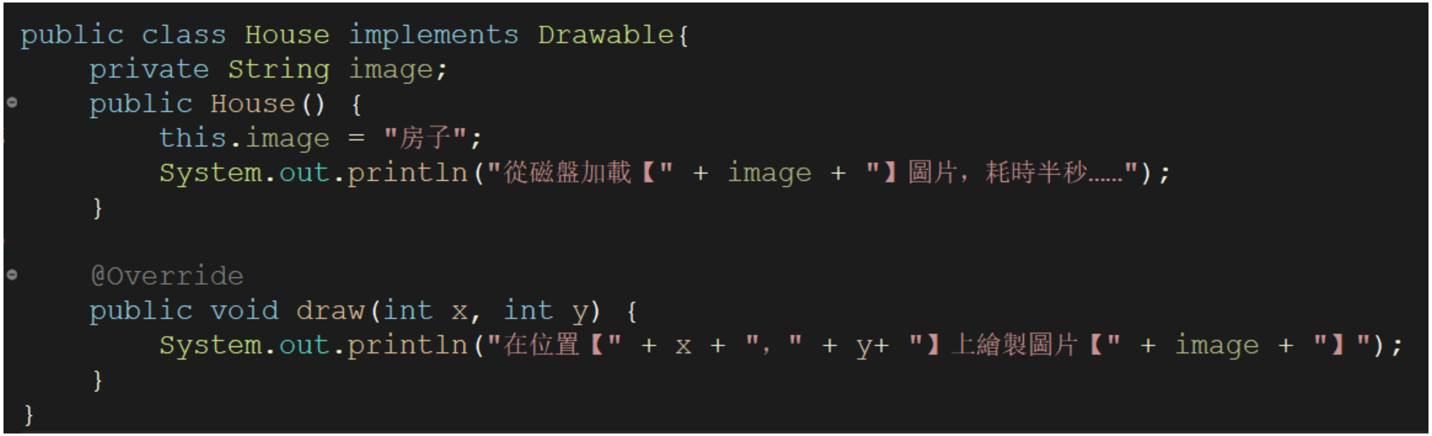


```java
public class River implements Drawable {
    private String image;
    public River() {
        this.image = "河流";
        System.out.println("從磁盤加載【" + image + "】圖片，耗時半秒......");
    }

    @Override
    public void draw(int x, int y) {
        System.out.println("在位置【" + x + ", " + y+ "】上繪製圖片【" + image + "】");
    }

}
```

Course: Soft Design

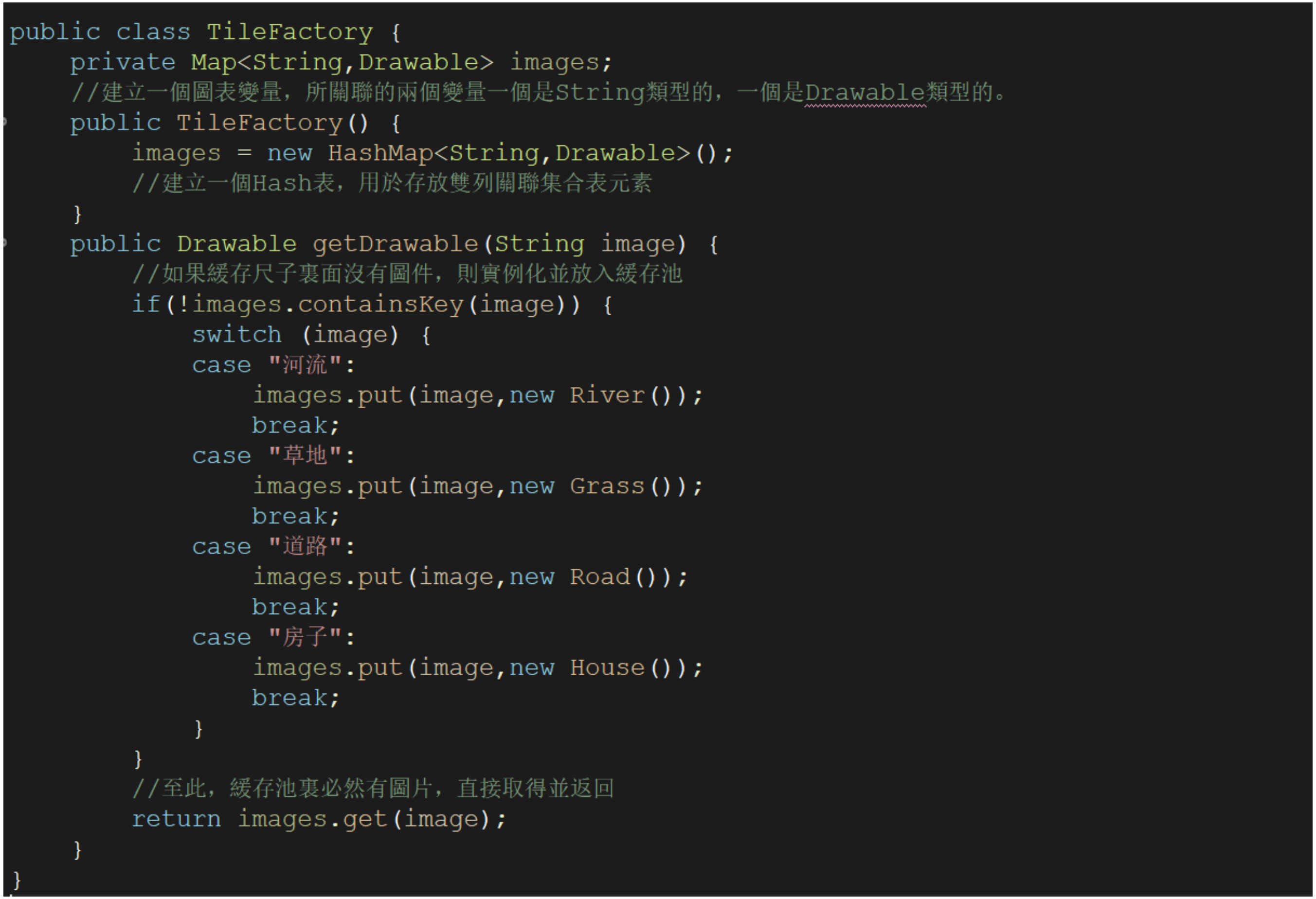


```
public class Road implements Drawable {
    private String image;
    public Road() {
        this.image = "道路";
        System.out.println("從磁盤加載【" + image + "】圖片，耗時半秒......");
    }

    @Override
    public void draw(int x, int y) {
        System.out.println("在位置【" + x + ", " + y+ "】上繪製圖片【" + image + "】");
    }
}
```



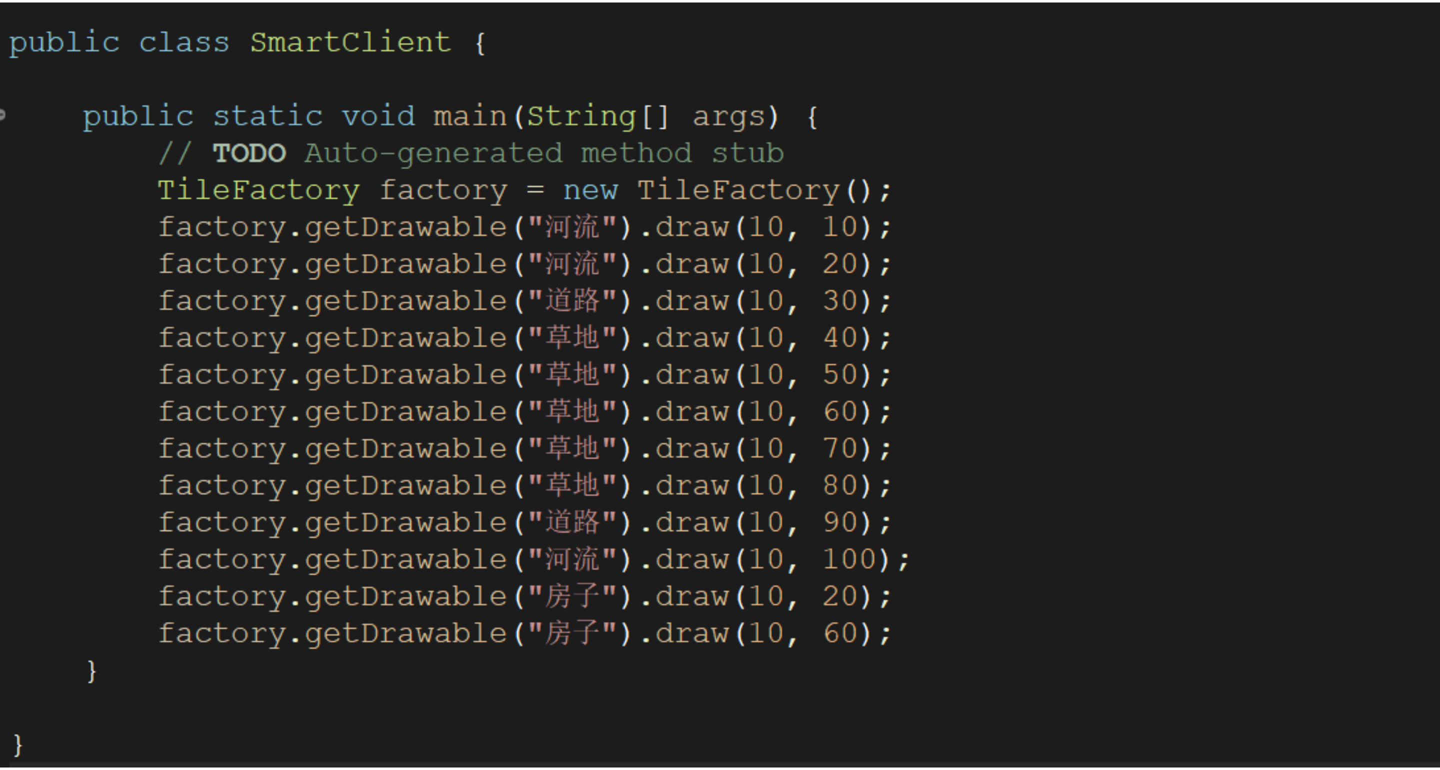
```
public class House implements Drawable{
    private String image;
    public House() {
        this.image = "房子";
        System.out.println("從磁盤加載【" + image + "】圖片，耗時半秒......");
    }

    @Override
    public void draw(int x, int y) {
        System.out.println("在位置【" + x + ", " + y+ "】上繪製圖片【" + image + "】");
    }
}
```

Course: Soft Design



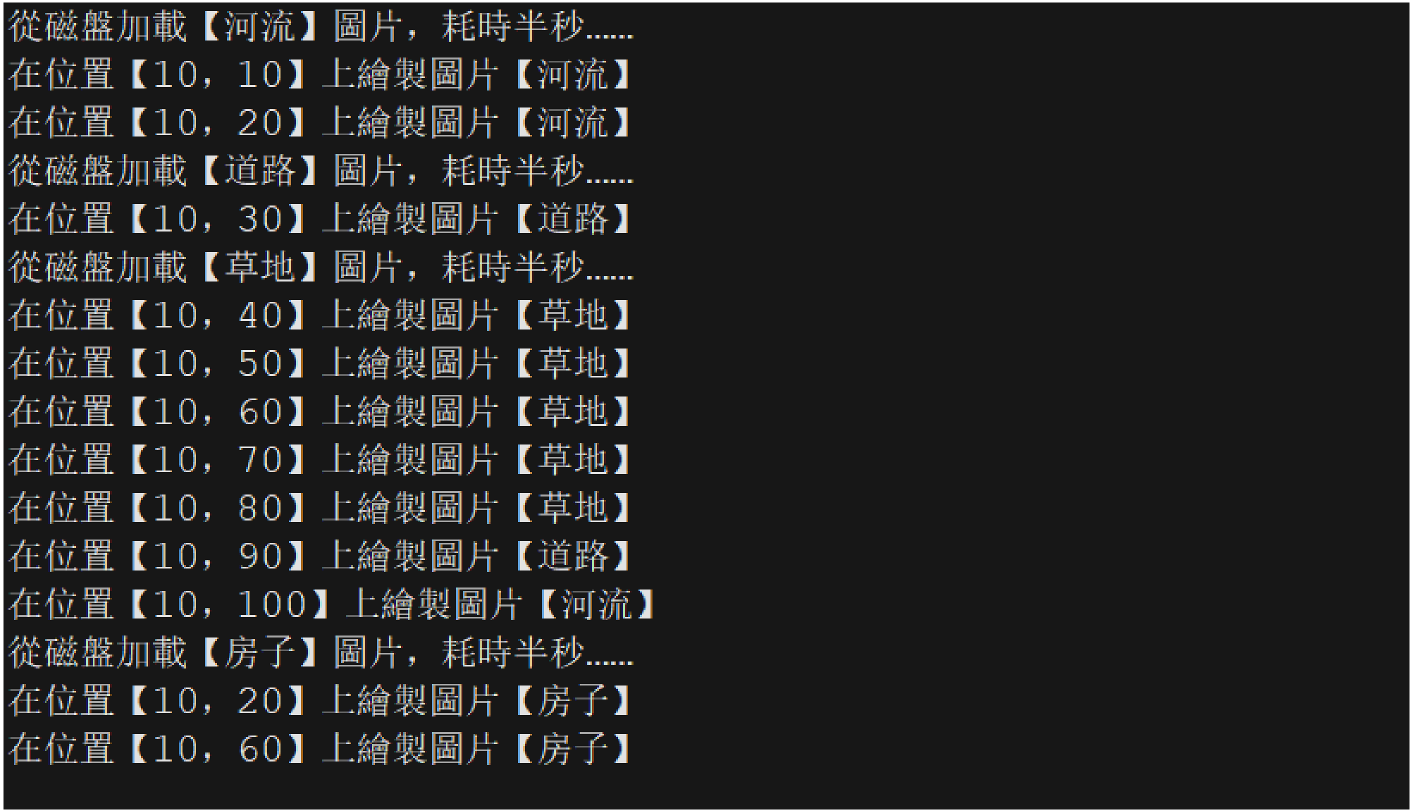
```
public class TileFactory {
    private Map<String,Drawable> images;
    //建立一個圖表變量，所關聯的兩個變量一個是String類型的，一個是Drawable類型的。
    public TileFactory() {
        images = new HashMap<String,Drawable>();
        //建立一個Hash表，用於存放雙列關聯集合表元素
    }
    public Drawable getDrawable(String image) {
        //如果緩存尺子裏面沒有圖件，則實例化並放入緩存池
        if(!images.containsKey(image)) {
            switch (image) {
            case "河流":
                images.put(image,new River());
                break;
            case "草地":
                images.put(image,new Grass());
                break;
            case "道路":
                images.put(image,new Road());
                break;
            case "房子":
                images.put(image,new House());
                break;
            }
        }
        //至此，緩存池裏必然有圖片，直接取得並返回
        return images.get(image);
    }
}
```

Course: Soft Design



```
public class SmartClient {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        TileFactory factory = new TileFactory();
        factory.getDrawable("河流").draw(10, 10);
        factory.getDrawable("河流").draw(10, 20);
        factory.getDrawable("道路").draw(10, 30);
        factory.getDrawable("草地").draw(10, 40);
        factory.getDrawable("草地").draw(10, 50);
        factory.getDrawable("草地").draw(10, 60);
        factory.getDrawable("草地").draw(10, 70);
        factory.getDrawable("草地").draw(10, 80);
        factory.getDrawable("道路").draw(10, 90);
        factory.getDrawable("河流").draw(10, 100);
        factory.getDrawable("房子").draw(10, 20);
        factory.getDrawable("房子").draw(10, 60);
    }

}
```

Course: Soft Design



```
從磁盤加載【河流】圖片，耗時半秒......
在位置【10，10】上繪製圖片【河流】
在位置【10，20】上繪製圖片【河流】
從磁盤加載【道路】圖片，耗時半秒......
在位置【10，30】上繪製圖片【道路】
從磁盤加載【草地】圖片，耗時半秒......
在位置【10，40】上繪製圖片【草地】
在位置【10，50】上繪製圖片【草地】
在位置【10，60】上繪製圖片【草地】
在位置【10，70】上繪製圖片【草地】
在位置【10，80】上繪製圖片【草地】
在位置【10，90】上繪製圖片【道路】
在位置【10，100】上繪製圖片【河流】
從磁盤加載【房子】圖片，耗時半秒......
在位置【10，20】上繪製圖片【房子】
在位置【10，60】上繪製圖片【房子】
```

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.

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.