Course: Soft Design

LECTURE 07

Template and **Iterator**

Software Development Principle 78~99

Template (模板模式)

Intent

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.

Motivation

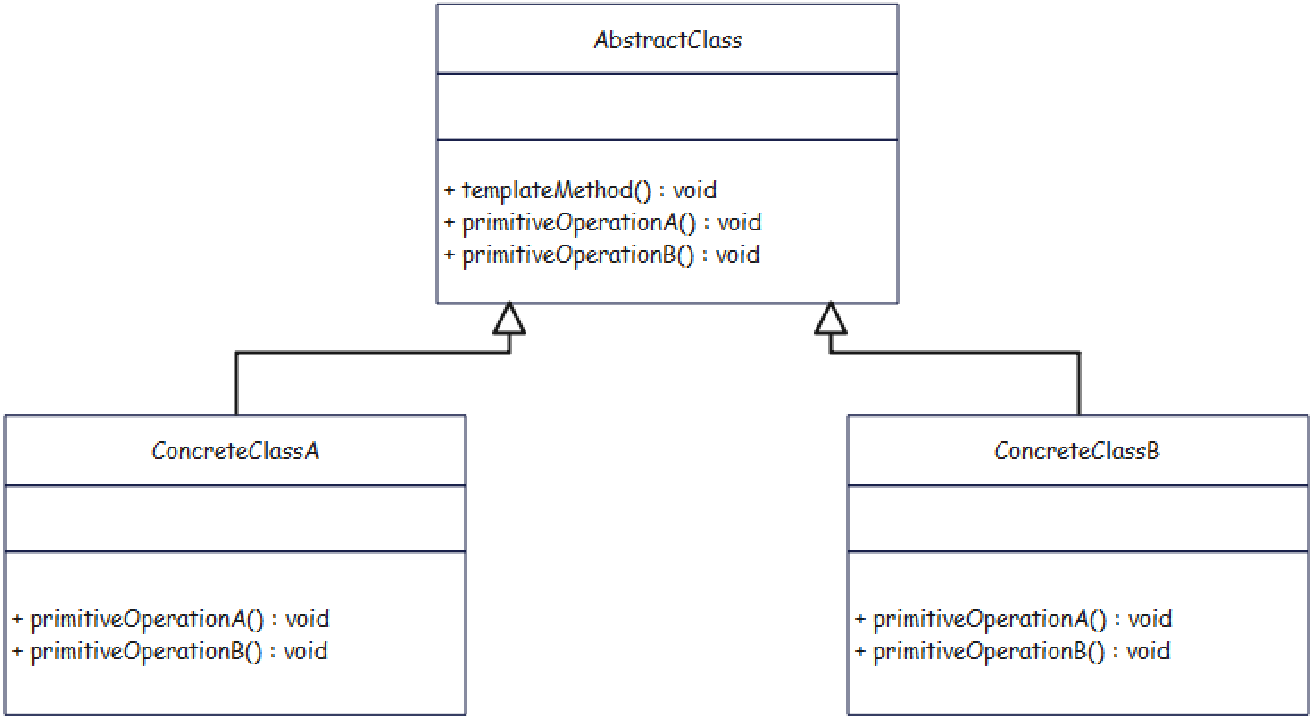
Applications built with the framework can subclass Application and Document to suit specific needs.

Applicability

The Template Method pattern should be used:

- to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
- to control subclasses extensions. You can define a template method that calls "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points.

* Class Diagram


AbstractClass
+ templateMethod() : void
+ primitiveOperationA() : void
+ primitiveOperationB() : void
ConcreteClassA
+ primitiveOperationA() : void
+ primitiveOperationB() : void
ConcreteClassB
+ primitiveOperationA() : void
+ primitiveOperationB() : void

Participants

AbstractClass (Application)

- defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
- implements a template method defining the skeleton of an algorithm.

ConcreteClass (MyApplication)

- implements the primitive operations to carry out subclass-specific steps of the algorithm.

Collaborations

ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Consequences

Template methods are a fundamental technique for code reuse. They are particularly important in class libraries, because they are the means for factoring out common behavior in library classes.

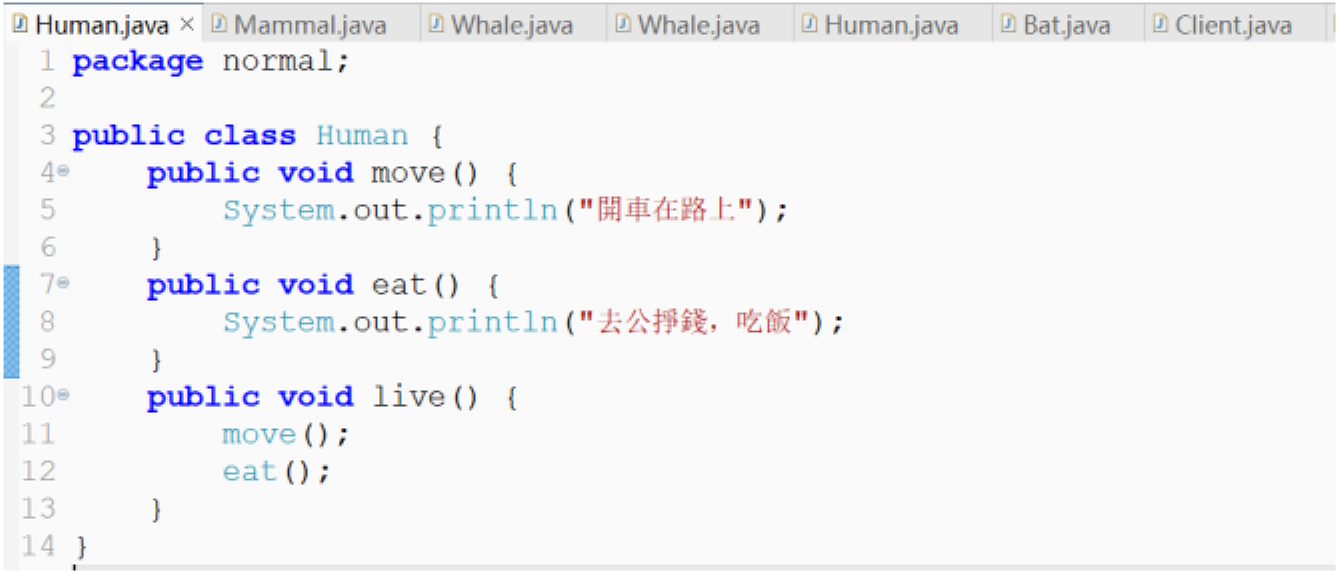
Template methods call the following kinds of operations:

- concrete operations
- concrete AbstractClass operations
- primitive operations
- factory methods (see Factory Method (121));
- hook operations, which provide default behavior that subclasses can extend if necessary.

Implementation

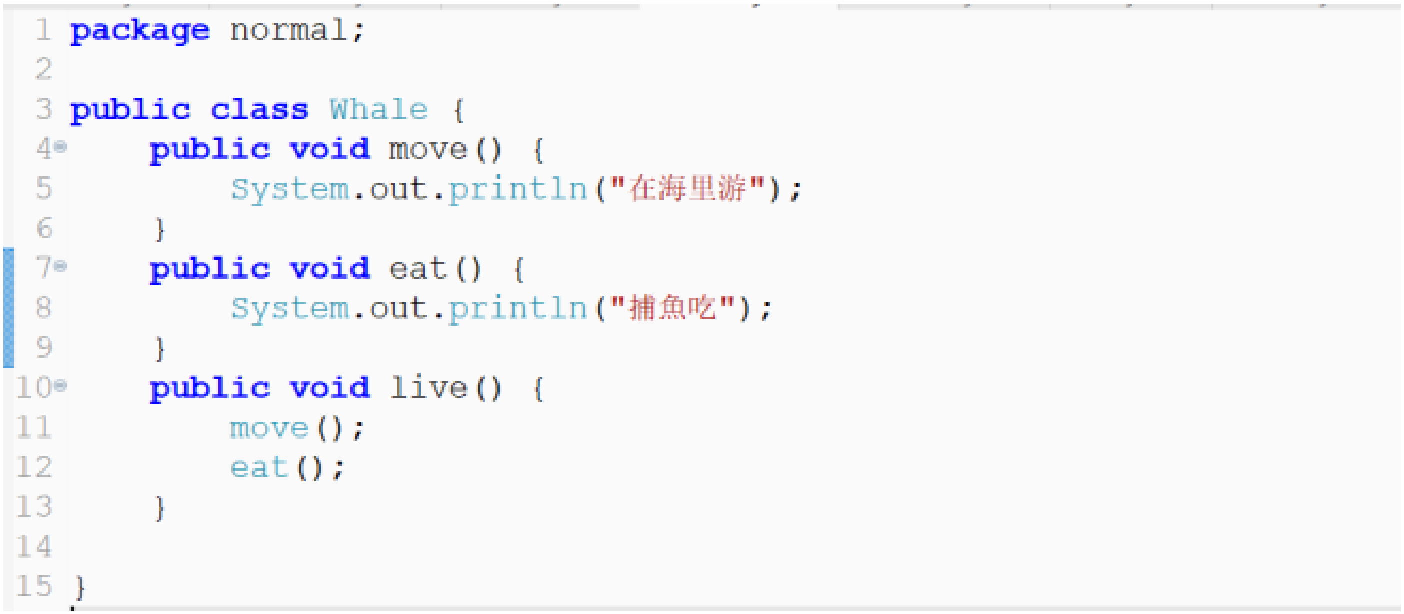
- Using C++ access control.
- Minimizing primitive operations.
- Naming conventions.

Course: Soft Design


Human.java × | Mammal.java | Whale.java | Whale.java | Human.java | Bat.java | Client.java

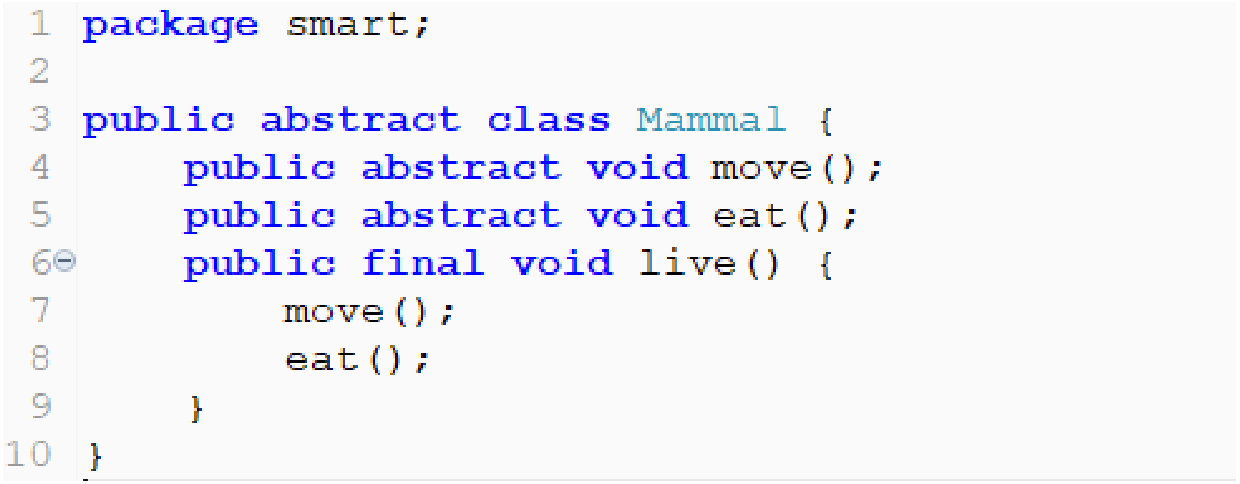
```
package normal;

public class Human {
    public void move() {
        System.out.println("開車在路上");
    }
    public void eat() {
        System.out.println("去公掙錢, 吃飯");
    }
    public void live() {
        move();
        eat();
    }
}
```

Course: Soft Design



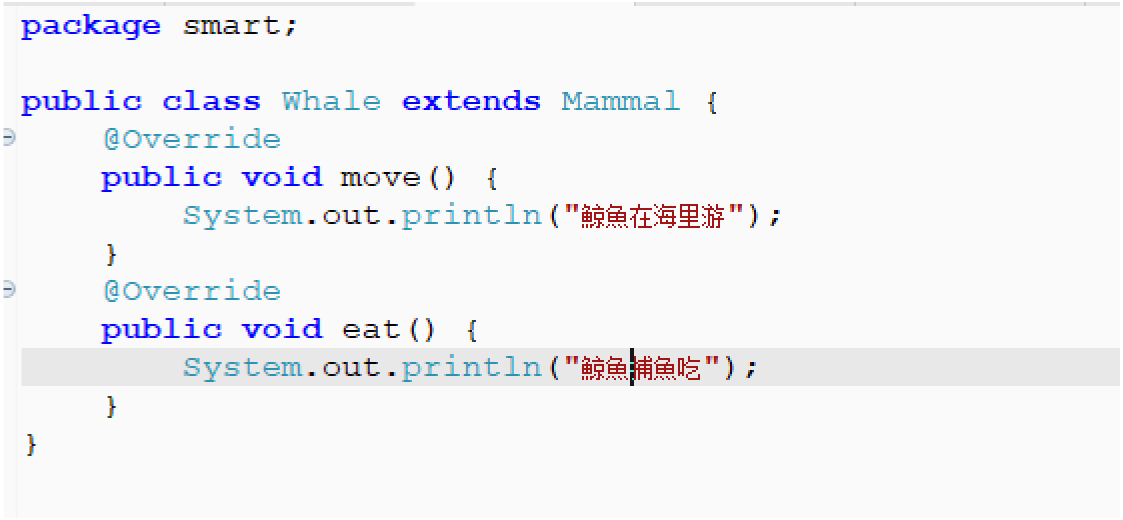
```java
package normal;

public class Whale {
    public void move() {
        System.out.println("在海里游");
    }
    public void eat() {
        System.out.println("捕魚吃");
    }
    public void live() {
        move();
        eat();
    }

}
```

Course: Soft Design



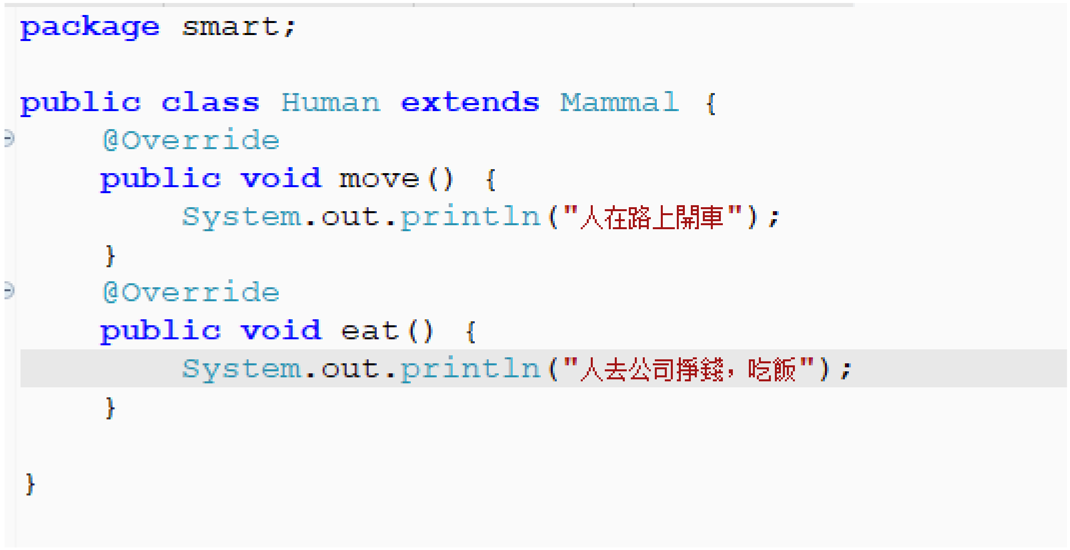
```java
package smart;

public abstract class Mammal {
	public abstract void move();
	public abstract void eat();
	public final void live() {
		move();
		eat();
	}
}
```

Course: Soft Design



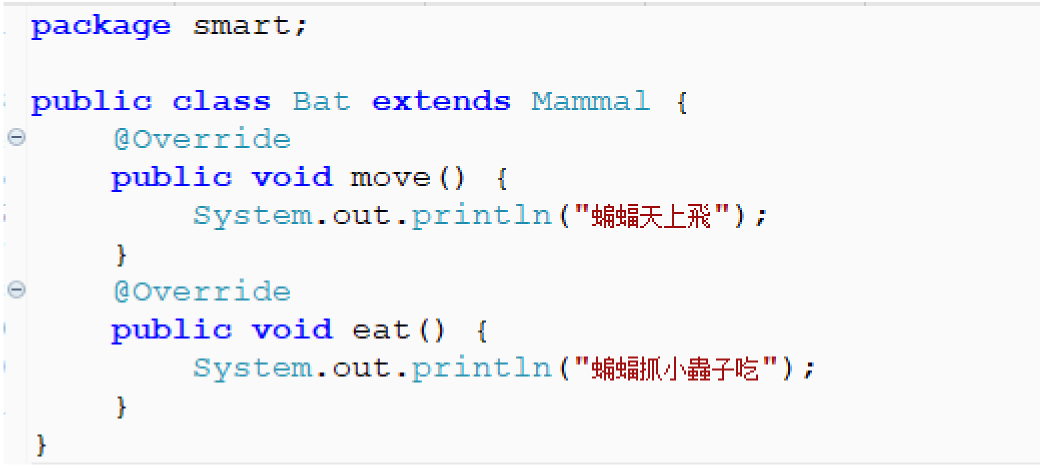
```
package smart;

public class Whale extends Mammal {
	@Override
	public void move() {
		System.out.println("鲸鱼在海里游");
	}
	@Override
	public void eat() {
		System.out.println("鲸鱼捕鱼吃");
	}
}
```

Course: Soft Design

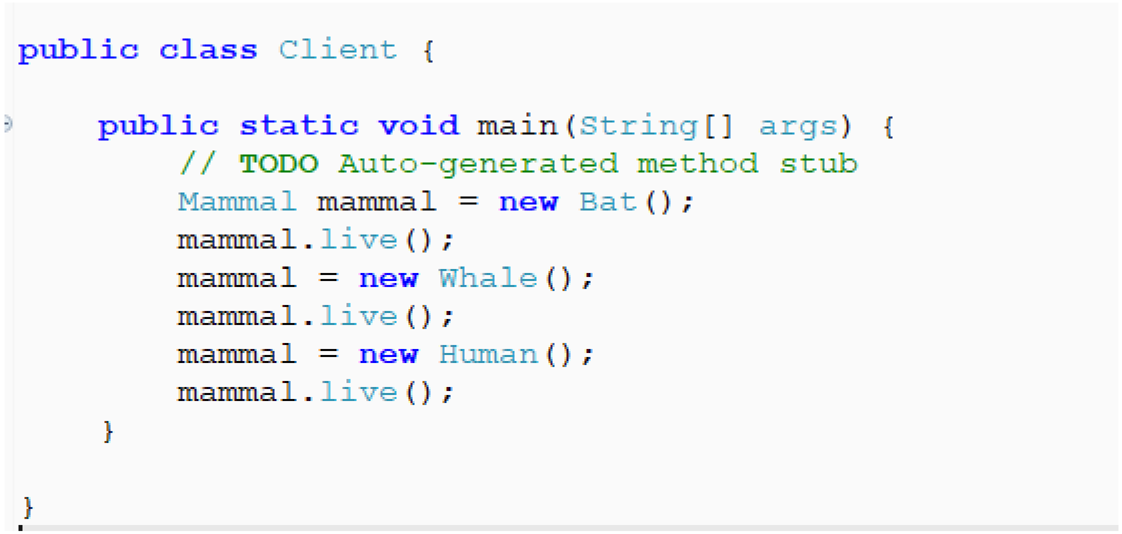


```
package smart;

public class Human extends Mammal {
    @Override
    public void move() {
        System.out.println("人在路上開車");
    }
    @Override
    public void eat() {
        System.out.println("人去公司掙錢，吃飯");
    }

}
```

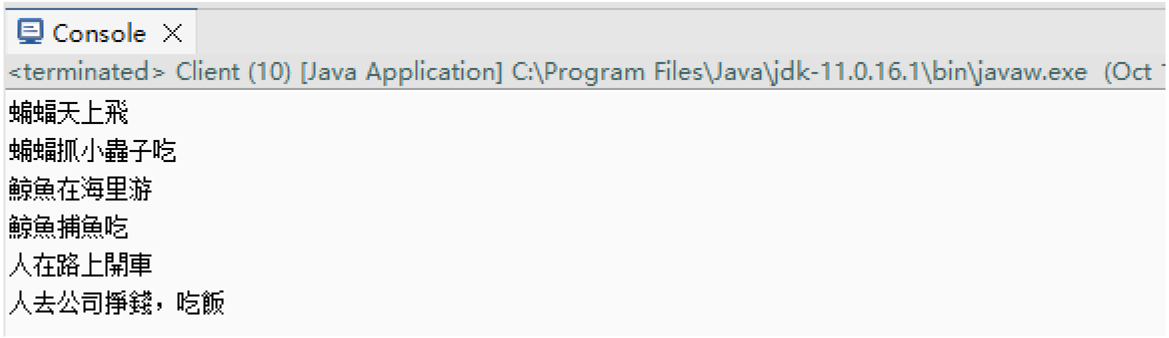
Course: Soft Design



```
package smart;

public class Bat extends Mammal {
    @Override
    public void move() {
        System.out.println("蝙蝠天上飛");
    }
    @Override
    public void eat() {
        System.out.println("蝙蝠抓小蟲子吃");
    }
}
```



```java
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Mammal mammal = new Bat();
        mammal.live();
        mammal = new Whale();
        mammal.live();
        mammal = new Human();
        mammal.live();
    }

}
```

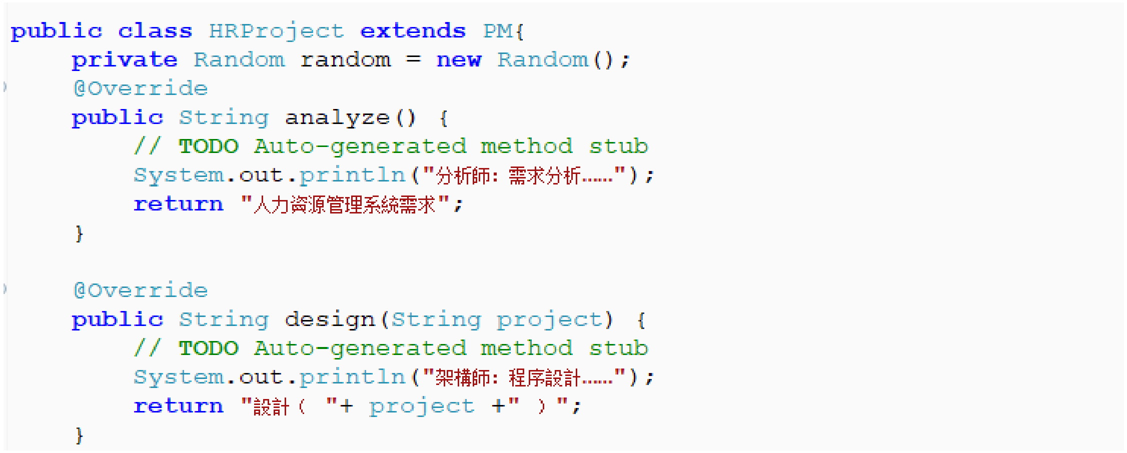


Console

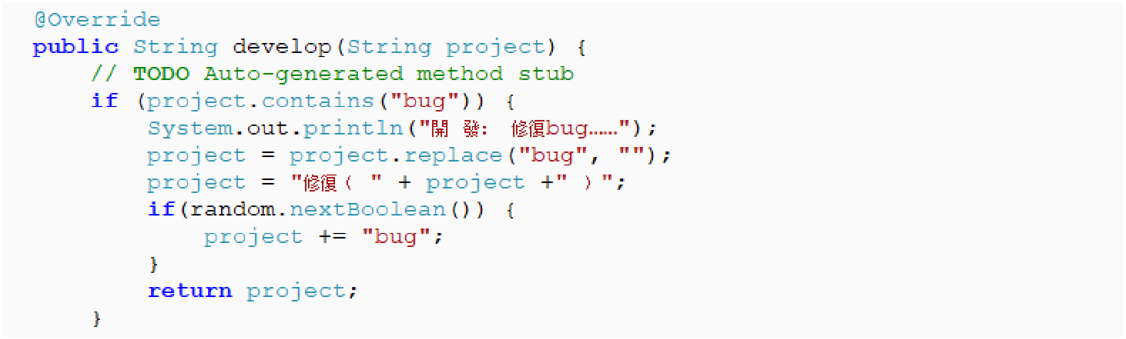
```
<terminated> Client (10) [Java Application] C:\Program Files\Java\jdk-11.0.16.1\bin\javaw.exe  (Oct
蝙蝠天上飛
蝙蝠抓小蟲子吃
鯨魚在海里游
鯨魚捕魚吃
人在路上開車
人去公司掙錢，吃飯
```

Course: Soft Design

```java
public abstract class PM {
    public abstract String analyze();//需求分析
    public abstract String design(String project);//軟件設計
    public abstract String develop(String project);//代碼開發
    public abstract boolean test(String project);//質量測試
    public abstract void release(String project);//上綫發佈

    protected final void kickoff() {
        String requirement = analyze();
        String designCode = design(requirement);
        do {
            designCode = develop(designCode);
        }while(!test(designCode));
        release(designCode);
    }
}
```

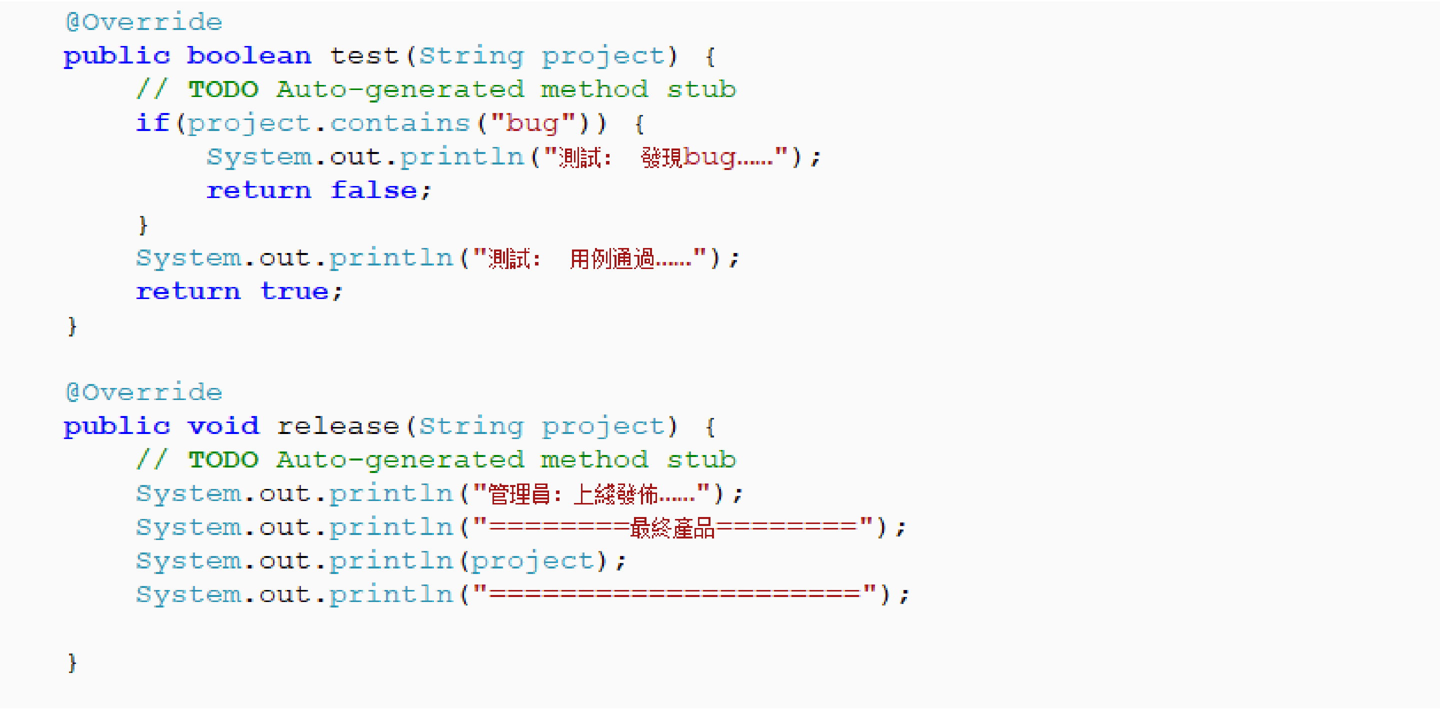


```java
public class HRProject extends PM{
    private Random random = new Random();
    @Override
    public String analyze() {
        // TODO Auto-generated method stub
        System.out.println("分析師: 需求分析......");
        return "人力資源管理系統需求";
    }

    @Override
    public String design(String project) {
        // TODO Auto-generated method stub
        System.out.println("架構師: 程序設計......");
        return "設計（ "+ project +" ）";
    }
```

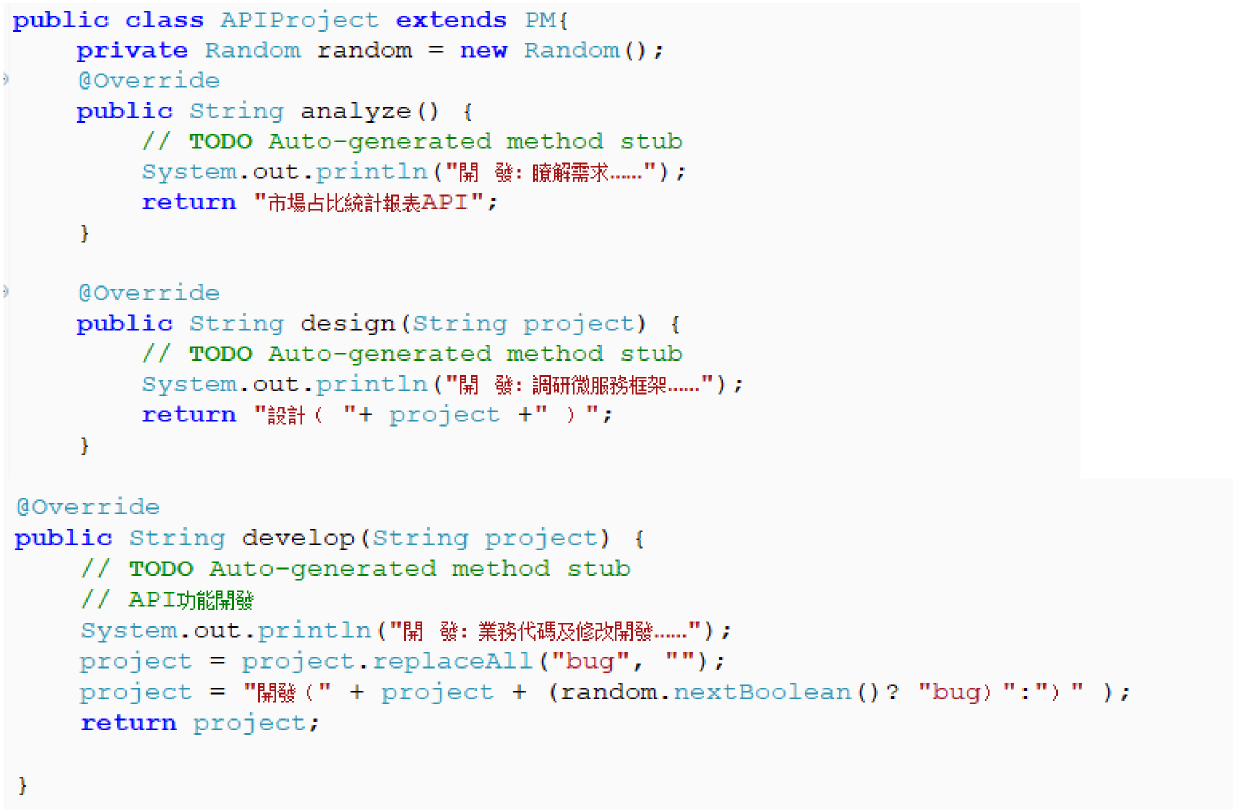


```java
  @Override
  public String develop(String project) {
      // TODO Auto-generated method stub
      if (project.contains("bug")) {
        System.out.println("開 發:  修復bug......");
        project = project.replace("bug", "");
        project = "修復（ " + project +" ）";
        if(random.nextBoolean()) {
            project += "bug";
        }
        return project;
    }
```

Course: Soft Design

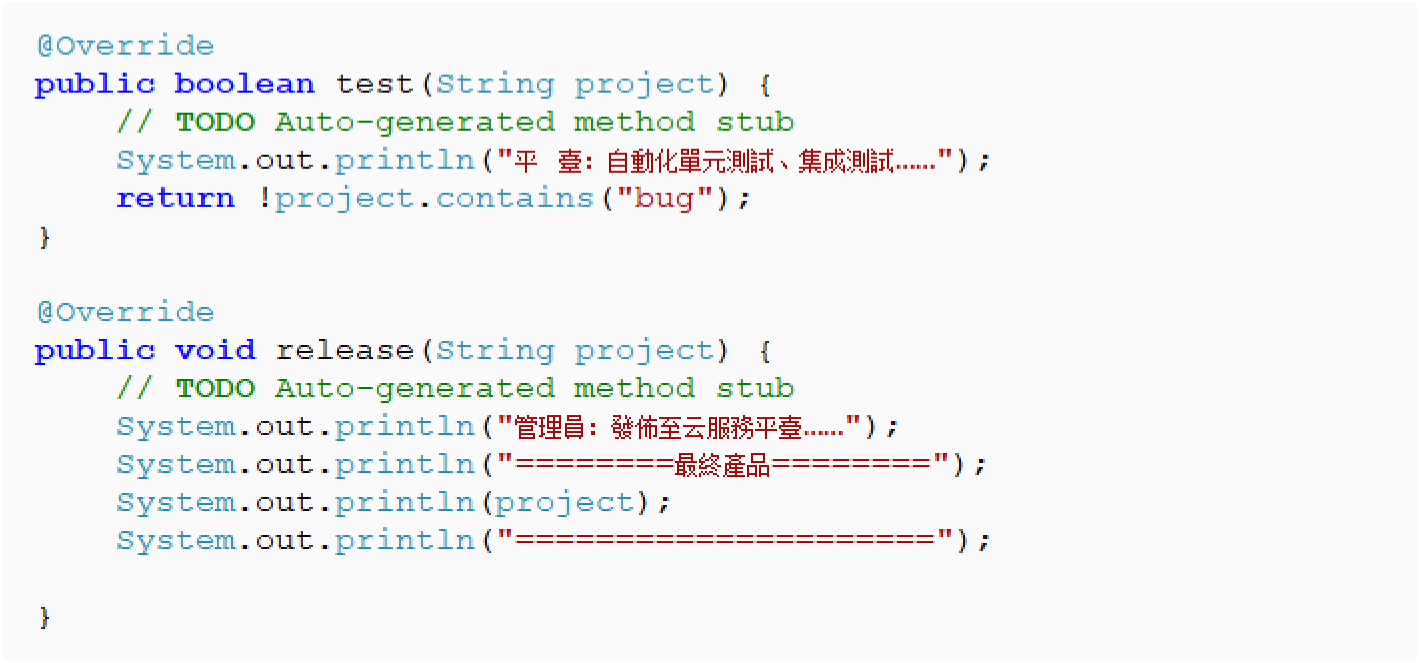


```
@Override
public boolean test(String project) {
    // TODO Auto-generated method stub
    if(project.contains("bug")) {
        System.out.println("測試:  發現bug……");
        return false;
    }
    System.out.println("測試:  用例通過……");
    return true;
}

@Override
public void release(String project) {
    // TODO Auto-generated method stub
    System.out.println("管理員: 上綫發佈……");
    System.out.println("========最終產品========");
    System.out.println(project);
    System.out.println("=====================");

}
```



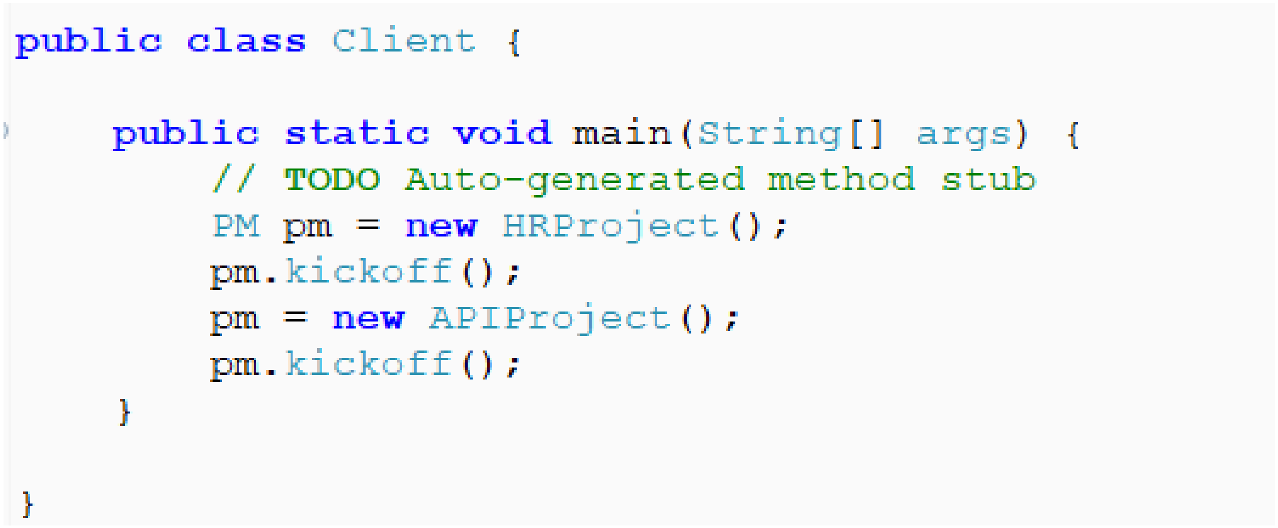
```
public class APIProject extends PM{
    private Random random = new Random();
    @Override
    public String analyze() {
        // TODO Auto-generated method stub
        System.out.println("開 發: 瞭解需求......");
        return "市場占比統計報表API";
    }

    @Override
    public String design(String project) {
        // TODO Auto-generated method stub
        System.out.println("開 發: 調研微服務框架......");
        return "設計（ "+ project +" ）";
    }
@Override
public String develop(String project) {
    // TODO Auto-generated method stub
    // API功能開發
    System.out.println("開 發: 業務代碼及修改開發......");
    project = project.replaceAll("bug", "");
    project = "開發（" + project + (random.nextBoolean()? "bug）":"）" );
    return project;

}
```

Course: Soft Design



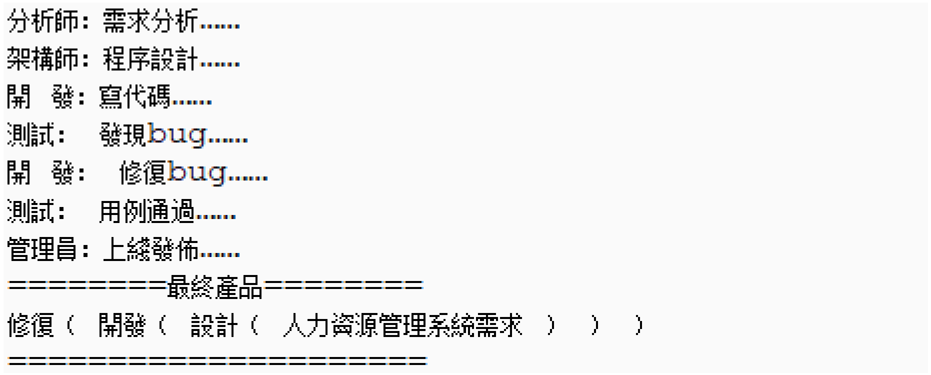
```java
@Override
public boolean test(String project) {
    // TODO Auto-generated method stub
    System.out.println("平 臺: 自動化單元測試、集成測試......");
    return !project.contains("bug");
}

@Override
public void release(String project) {
    // TODO Auto-generated method stub
    System.out.println("管理員: 發佈至云服務平臺......");
    System.out.println("========最終產品========");
    System.out.println(project);
    System.out.println("=====================");

}
```

Course: Soft Design

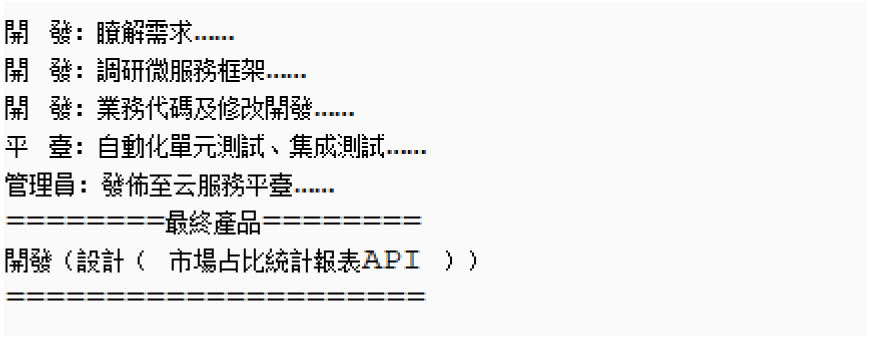


```
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        PM pm = new HRProject();
        pm.kickoff();
        pm = new APIProject();
        pm.kickoff();
    }

}
```

Course: Soft Design



```
分析師：需求分析......
架構師：程序設計......
開　發：寫代碼......
測試：　發現bug......
開　發：　修復bug......
測試：　用例通過......
管理員：上綫發佈......
=========最終產品=========
修復（　開發（　設計（　人力資源管理系統需求　）　）　）
========================
```



```
開　發：瞭解需求......
開　發：調研微服務框架......
開　發：業務代碼及修改開發......
平　臺：自動化單元測試、集成測試......
管理員：發佈至云服務平臺......
=========最終產品=========
開發（設計（　市場占比統計報表API　））
========================
```

Iterator (迭代模式)

Intent

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

Also known as

Cursor

Motivation

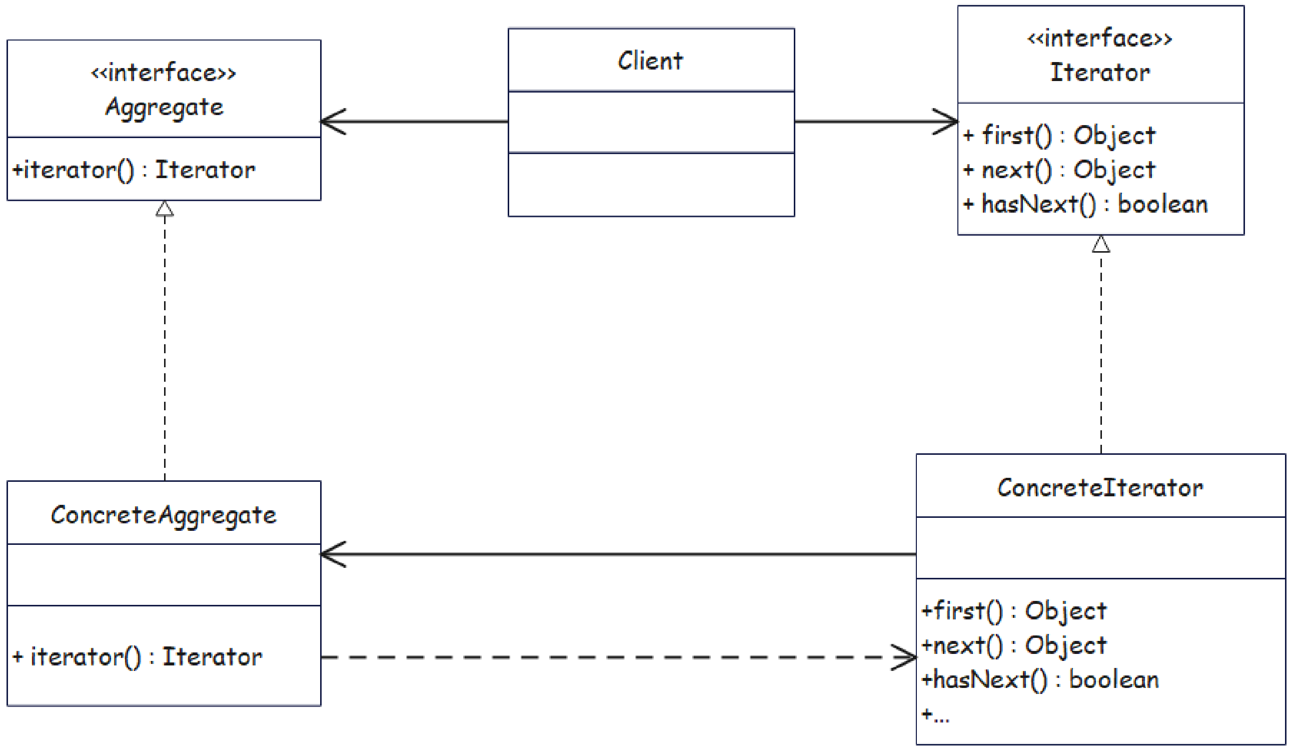
The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object.

Applicability

Use the Iterator pattern:

- to access an aggregate object's contents without exposing its internal representation.both the abstractions and their implementations should be extensible by subclassing.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures .

* Class Diagram



Participants

Iterator

- defines an interface for accessing and traversing elements.

ConcreteIterator

- implements the Iterator interface.
- keeps track of the current position in the traversal of the aggregate.

Aggregate

- defines an interface for creating an Iterator object.

ConcreteAggregate

- implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Collaborations

- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences

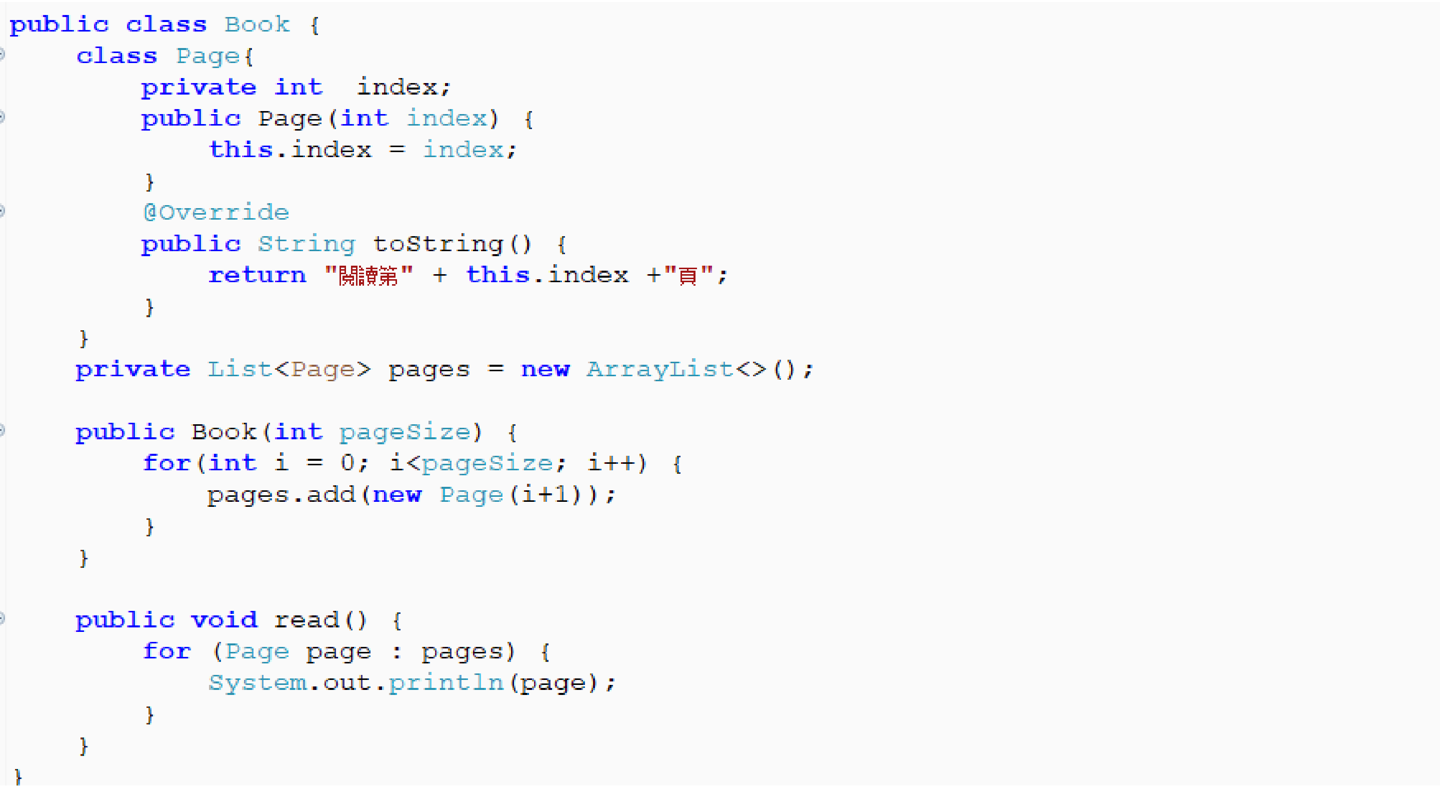
The Iterator pattern has three important consequences:

- It supports variations in the traversal of an aggregate.
- Iterators simplify the Aggregate interface.
- More than one traversal can be pending on an aggregate.

Implementation

- *Who controls the iteration?*
- *Who defines the traversal algorithm?*
- *How robust is the iterator?*
- *Additional Iterator operations.*
- *Iterators may have privileged access.*
- *Using polymorphic iterators in C++.*
- *Null iterators.*
- *Iterators for composites.*

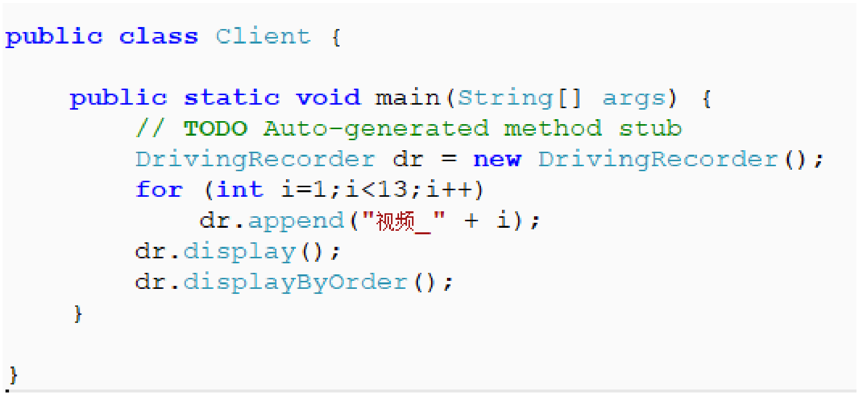
Course: Soft Design



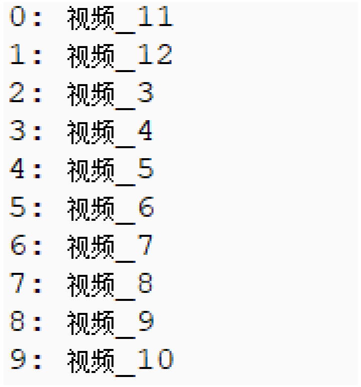
```java
public class Book {
    class Page{
        private int  index;
        public Page(int index) {
            this.index = index;
        }
        @Override
        public String toString() {
            return "閱讀第" + this.index +"頁";
        }
    }
    private List<Page> pages = new ArrayList<>();

    public Book(int pageSize) {
        for(int i = 0; i<pageSize; i++) {
            pages.add(new Page(i+1));
        }
    }

    public void read() {
        for (Page page : pages) {
            System.out.println(page);
        }
    }
}
```

Course: Soft Design

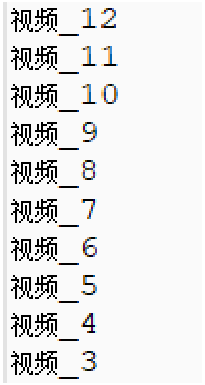
```java
public class DrivingRecorder {
    private int index = -1;
    private String[] records = new String[10];

    public void append(String record) {
        if(index == 9)
            index = 0;
        else
            index++;
        records[index] = record;
    }
    public void display(){
        for (int i = 0; i<10; i++) {
            System.out.println(i + ": " + records[i]);
        }
    }
    public void displayByOrder() {
        for(int i = index, loopCount = 0; loopCount < 10; i = i==0?i=9:i-1, loopCount++) {
            System.out.println(records[i]);

        }
    }


}
```



```
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        DrivingRecorder dr = new DrivingRecorder();
        for (int i=1;i<13;i++)
            dr.append("视频_" + i);
        dr.display();
        dr.displayByOrder();
    }

}
```

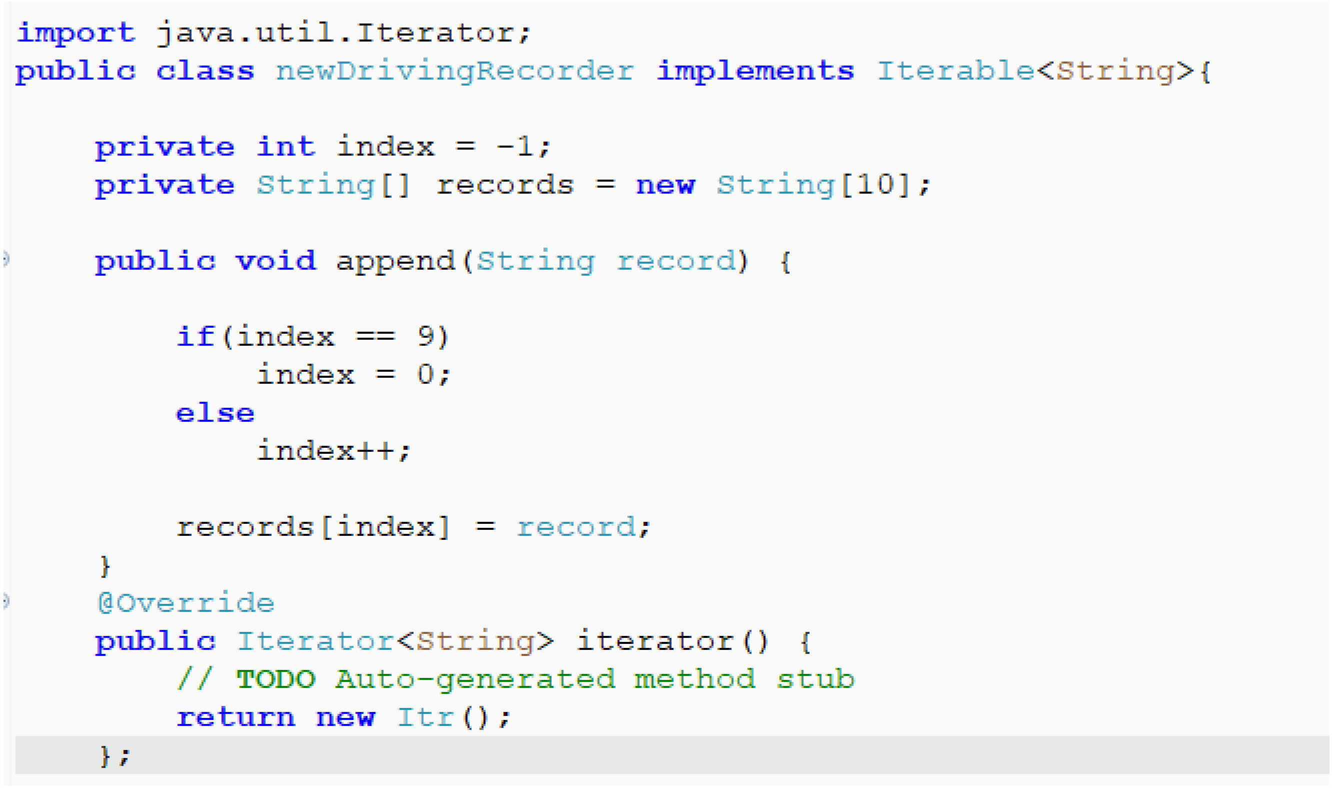


```
0: 视频_11
1: 视频_12
2: 视频_3
3: 视频_4
4: 视频_5
5: 视频_6
6: 视频_7
7: 视频_8
8: 视频_9
9: 视频_10
```



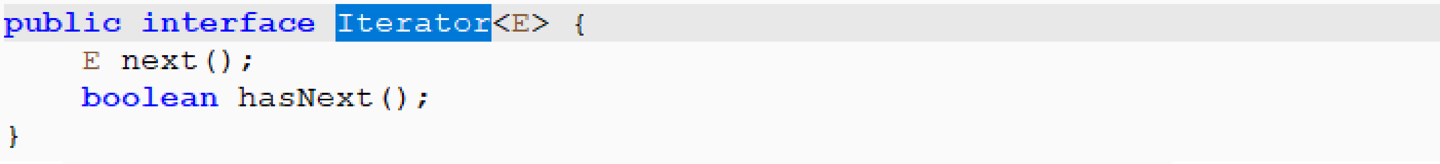
```
视频_12
视频_11
视频_10
视频_9
视频_8
视频_7
视频_6
视频_5
视频_4
视频_3
```

Course: Soft Design

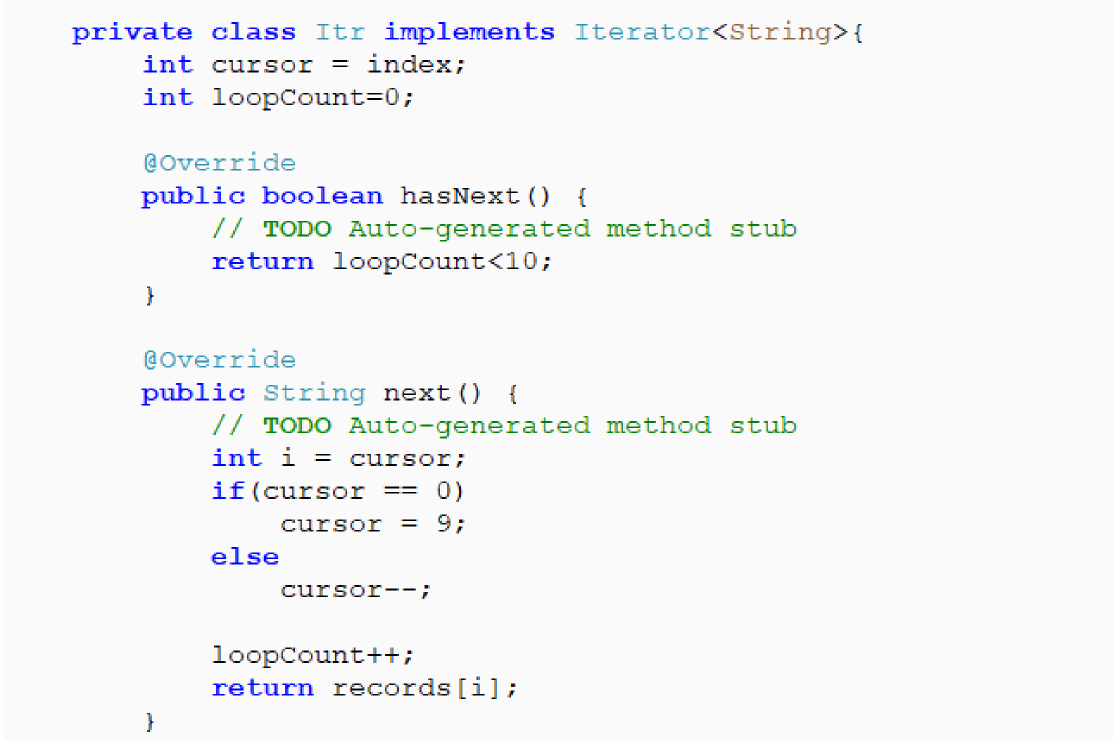


```java
import java.util.Iterator;
public class newDrivingRecorder implements Iterable<String>{

    private int index = -1;
    private String[] records = new String[10];

    public void append(String record) {

        if(index == 9)
            index = 0;
        else
            index++;

        records[index] = record;
    }
    @Override
    public Iterator<String> iterator() {
        // TODO Auto-generated method stub
        return new Itr();
    };
```

Course: Soft Design

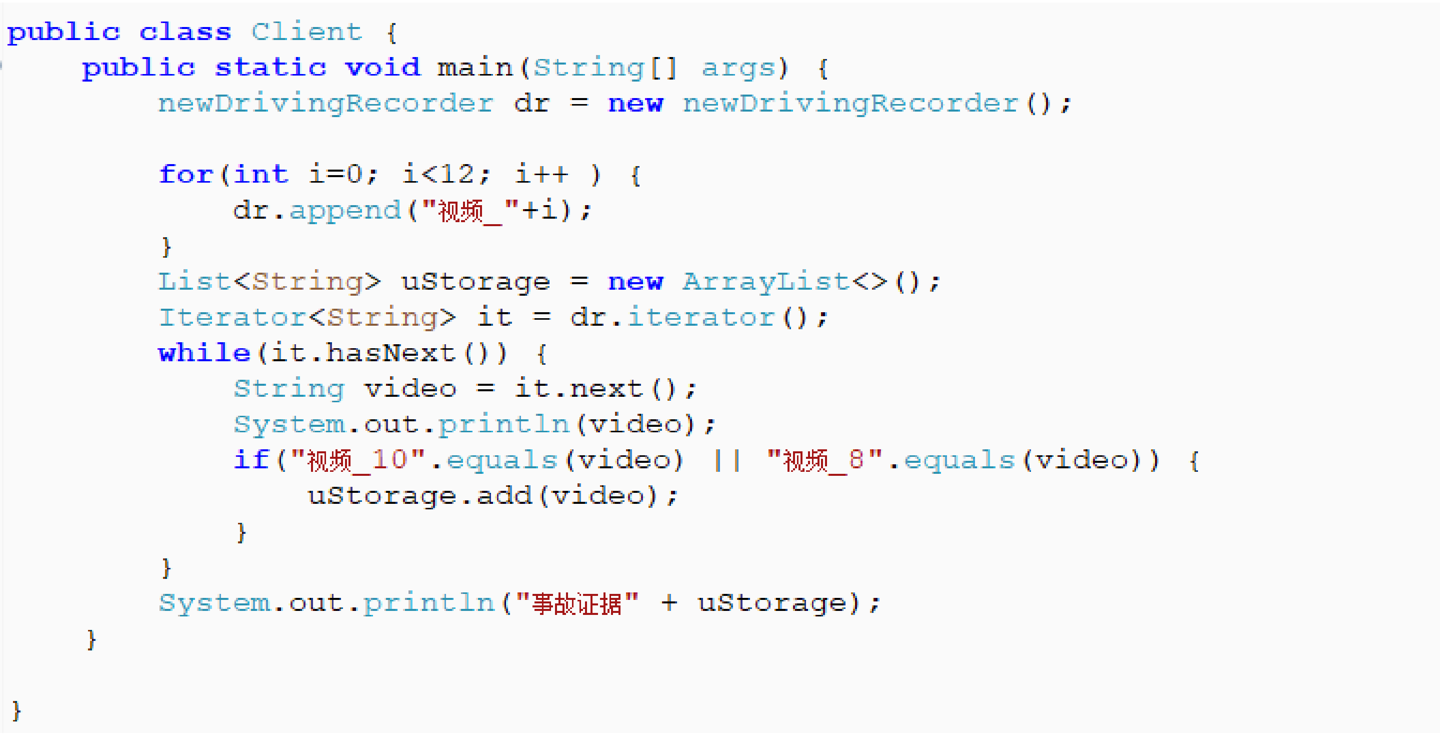


```
public interface Iterator<E> {
     E next();
     boolean hasNext();
}
```



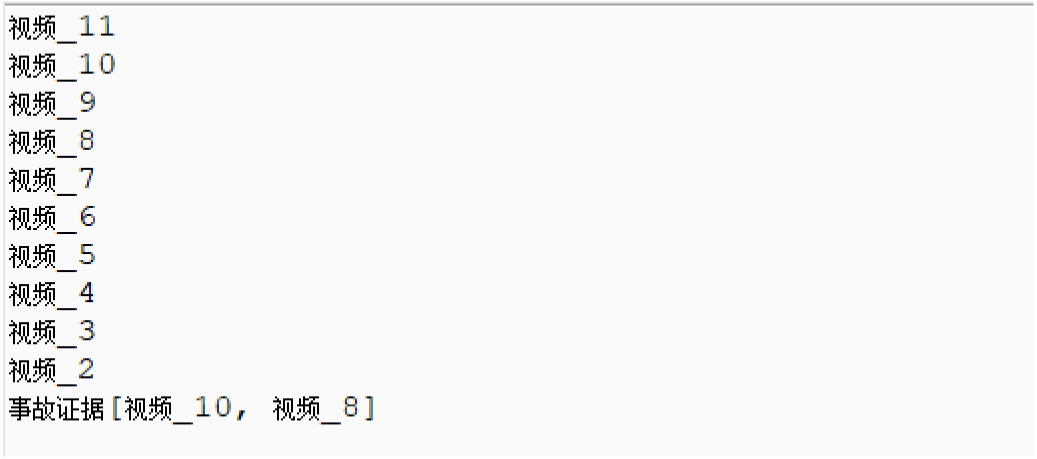
```
    private class Itr implements Iterator<String>{
        int cursor = index;
        int loopCount=0;

        @Override
        public boolean hasNext() {
            // TODO Auto-generated method stub
            return loopCount<10;
        }

        @Override
        public String next() {
            // TODO Auto-generated method stub
            int i = cursor;
            if(cursor == 0)
                cursor = 9;
            else
                cursor--;

            loopCount++;
            return records[i];
        }
```

Course: Soft Design



```java
public class Client {
    public static void main(String[] args) {
        newDrivingRecorder dr = new newDrivingRecorder();

        for(int i=0; i<12; i++ ) {
            dr.append("视频_"+i);
        }
        List<String> uStorage = new ArrayList<>();
        Iterator<String> it = dr.iterator();
        while(it.hasNext()) {
            String video = it.next();
            System.out.println(video);
            if("视频_10".equals(video) || "视频_8".equals(video)) {
                uStorage.add(video);
            }
        }
        System.out.println("事故证据" + uStorage);
    }

}
```

Course: Soft Design



```
视频_11
视频_10
视频_9
视频_8
视频_7
视频_6
视频_5
视频_4
视频_3
视频_2
事故证据[视频_10, 视频_8]
```

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.

Chapter 3 Coding Principles

Coding is the set of activities including:

1. Translating the algorithms specified during design into programs writ-ten in a computer language.
2. 2. Translating, usually automatically, the programs into a language directly executable by a computer.

The primary output of coding is a documented program listing.

Principle 87

Avoid tricks

Many programmers love to create programs with tricks. These are constructs that perform a function correctly, but in a particularly obscure manner. There are many ways to explain why tricks are used so often:

1. Programmers are extremely intelligent and want to demonstrate that intelligence.
2. 2.Maintainers, when they finally figure out how the trick works, will not only recognize how smart the original programmer was, but also will realize how smart they themselves are.
3. 3. Job security. Bottom line: Show the world how smart you are by avoiding tricky code!

Principle 88

Avoid global variables

Global variables make it convenient to write programs; after all, if you need to access or change x, you just do it. Unfortunately, if x is ever accessed and found to have an inappropriate value (say,-16.3 ships), it is difficult to determine which software component is at fault."Global"implies that anybody could have altered its value incorrectly.

Principle 89

Write to read top-down

People generally read a program from top (i.e,first line) to bottom (i.e,last line). Write a program to help the reader understand it. Among the implications of this principle are:
1.Include a detailed external specification up front to clearly define the program purpose and use.
2. Specify externally accessed routines,local variables, and algorithms up front.
3. Use the so-called"structured" programming constructs, which are inherently easier to follow.

Principle 90

Avoid side-effects

A side-effect of a procedure is something the procedure does that is not its main purpose and that is visible (or whose results are perceivable)from outside the procedure. Side-effects are the sources of many subtle errors in software, that is, the ones that are the most latent and the ones that are most difficult to discover once their symptoms manifest themselves.

Principle 91

Use meaningful names

An even better argument is that overly shortened names actually decrease productivity. There are two reasons: (1) Testing and maintenance costs rise because people spend time trying to decode names, and (2)more time could be spent typing when using shortened names! The second argument is true because of the necessity to add comments.

Principle 92

Write programs for people first

The most valuable resource is now labor: labor to develop the software, labor to maintain the software, and labor to enhance capability. With few application exceptions, programmers should think first of the people who will later attempt to understand and adapt the software. Anything that can be done to assist them should be done.

Principle 93

Use optimal data structures

The structure of data and the structure of programs manipulating that data are intimately interrelated. If you select the right data structures, your algorithms (and thus your code) become easy to write, and easy to read, and therefore easy to maintain. Read any book on algorithms or on data structures(they're one and the same!).

Principle 94

Get it right before you make it faster

It is far easier to adapt a working program to make it run faster than to adapt a fast program to make it work. Don't worry about optimization when doing your initial coding. [On the other hand, don't use a ridiculously inefficient algorithm or set of data structures (Principles 79 and 93).]

Principle 95

Comment before you finalize your code

I've often heard programmers say, "Why should I bother commenting my code now? It'll only change!" We comment code to make the software easi-er to debug, test, and maintain. By commenting your code while coding (or beforehand, see Principle 96),it will be easier for you to debug the software.

Principle 96

Document before you start coding

96.This advice will seem strange to some readers, but it becomes natural after being practiced for a while. Principle 95 explained why you should document your code before finishing it. Principle 96 goes one step further: You should document your code before starting to code!

Principle 97

Hand-execute every component

It might take 30 minutes to execute a software component by hand with a few simple test cases. Do it! I am suggesting this in addition to, not in lieu of, the more thorough computer-based unit testing that is already being performed.

Principle 98

Inspect code

Inspection of software detailed design and code was first proposed by Michael Fagan in his paper entitled"Design and Code Inspections to Reduce Errors in Program Development"[IBM Systems Journal,15,3 (July 1976), pp.182-211]. It can account for as many as 82 percent of all errors found in software. Inspection is much better than testing for finding errors. Define criteria for completing an inspection. Keep track of the types of errors found through inspection. Fagan's inspections consume approximately 15 percent of development resources with a net reduction in total development cost of 25 to 30 percent.

Principle 99

You can use unstructured languages

Unstructured code violates Edsger Dijkstra's guidance to restrict control structures to IF-THEN-ELSE, DO-WHILE, DO-UNTIL, and CASE. Notice that it is possible to write structured code in languages without these structures, such as in assembly languages, by documenting the code with the structured control statements and restricting the use of coro's to implementing these structures only.