LECTURE 09

State and **Memento**

Software Development Principle 115~135

State(状态模式)

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

Also known as
Objects for States

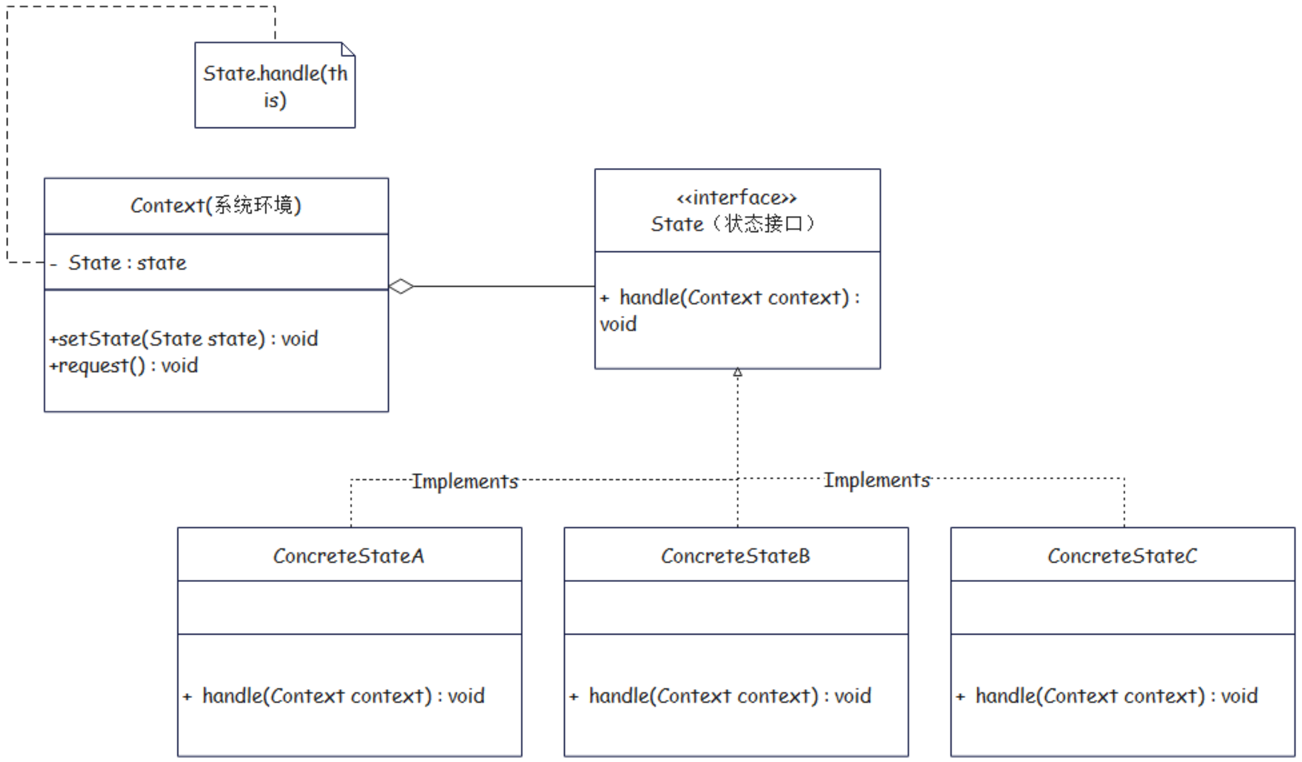
Motivation
The State class declares an interface common to all classes that represent different operational states.

Applicability

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants.

Class Diagram


State.handle(this)
Context(系统环境)
- State : state
+setState(State state) : void
+request() : void
<<interface>>
State（状态接口）
+ handle(Context context) : void
Implements
Implements
ConcreteStateA
+ handle(Context context) : void
ConcreteStateB
+ handle(Context context) : void
ConcreteStateC
+ handle(Context context) : void

Participants

Context

- ➢ defines the interface of interest to clients.
- ➢ maintains an instance of a ConcreteState subclass that defines the current state.

State

- ➢ defines an interface for encapsulating the behavior associated with a particular state of the Context.

ConcreteState subclasses

- ➢ each subclass implements a behavior associated with a state of the Context.

Collaborations

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

Consequences

The State pattern has the following consequences:

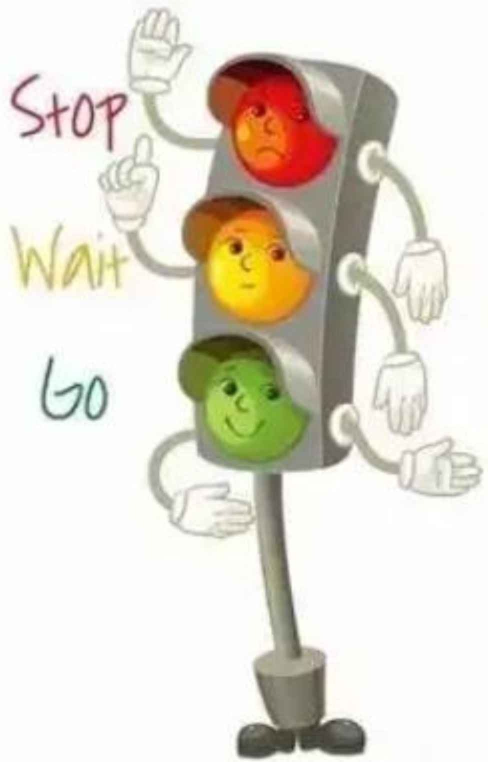
- It localizes state-specific behavior and partitions behavior for different states.
- It makes state transitions explicit.
- State objects can be shared.

Implementation

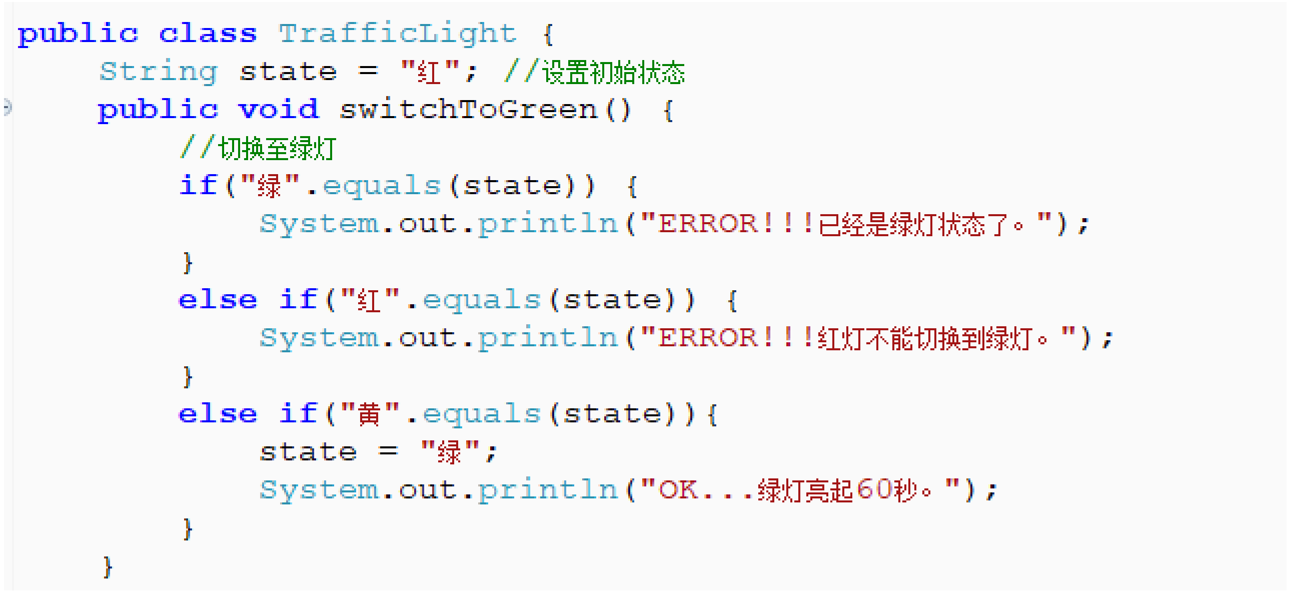
The State pattern raises a variety of implementation issues:

- Who defines the state transitions?
- A table-based alternative.
- Creating and destroying State objects.
- Using dynamic inheritance.

Course: Soft Design

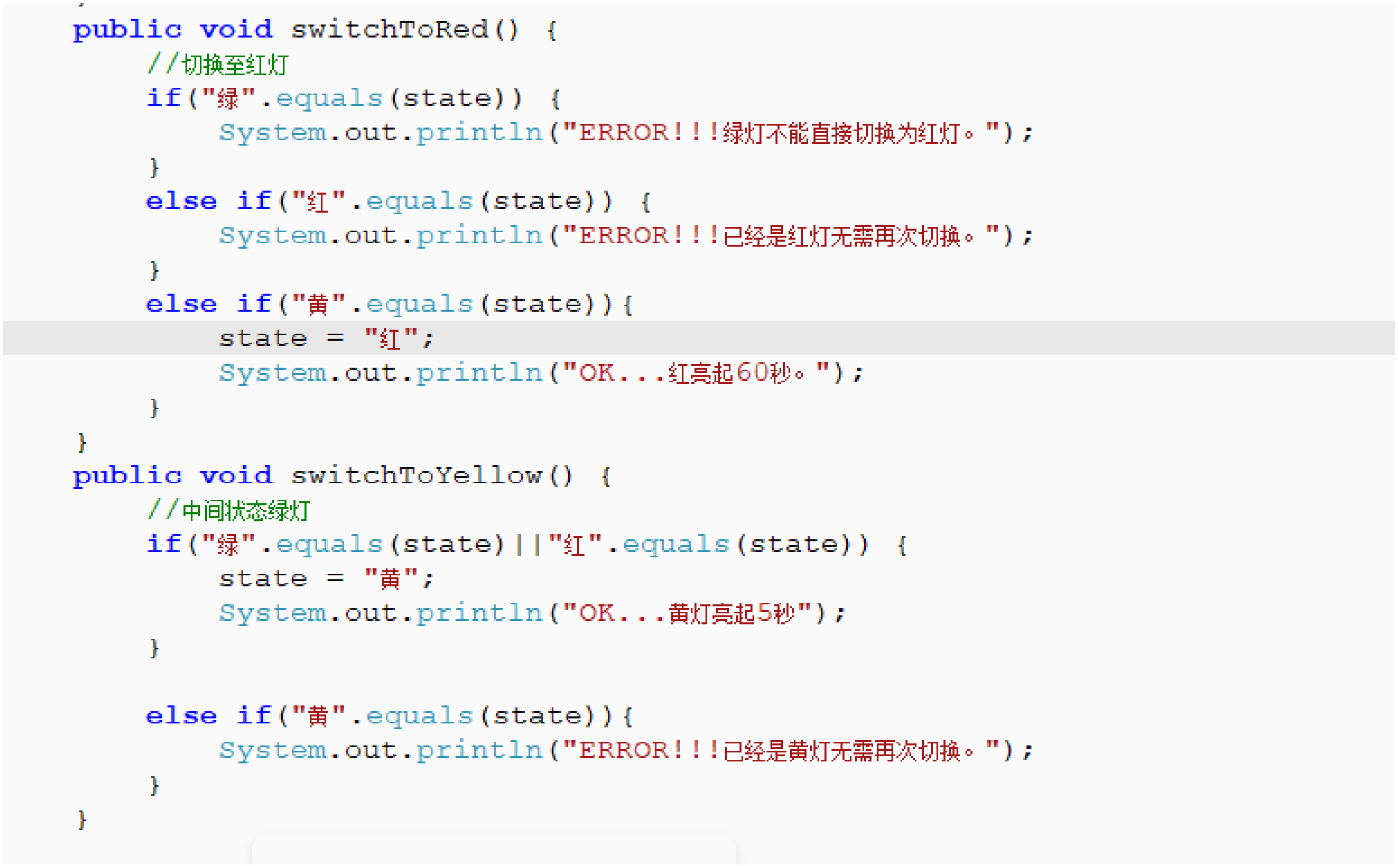


Course: Soft Design



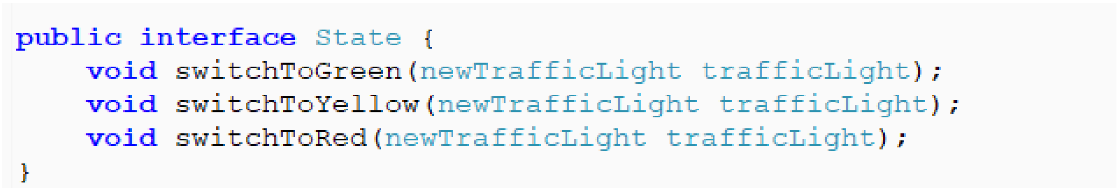
```java
public class TrafficLight {
    String state = "红"; //设置初始状态
    public void switchToGreen() {
        //切换至绿灯
        if("绿".equals(state)) {
            System.out.println("ERROR!!!已经是绿灯状态了。");
        }
        else if("红".equals(state)) {
            System.out.println("ERROR!!!红灯不能切换到绿灯。");
        }
        else if("黄".equals(state)){
            state = "绿";
            System.out.println("OK...绿灯亮起60秒。");
        }
    }
```

Course: Soft Design

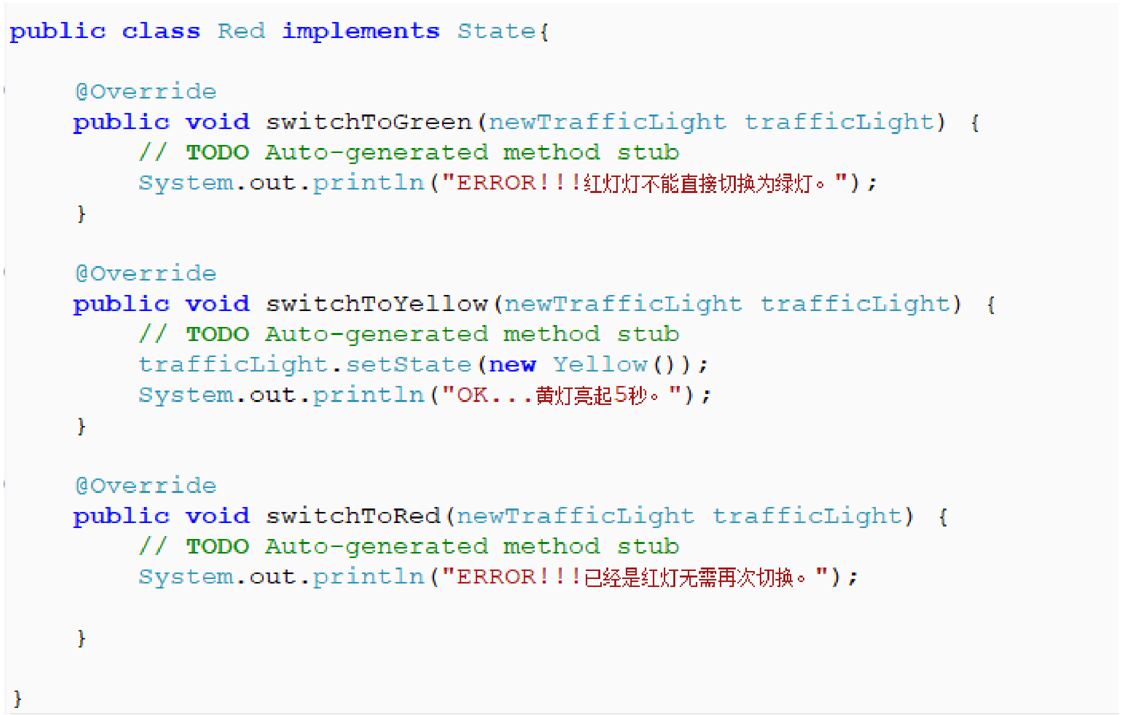


```java
    public void switchToRed() {
        //切换至红灯
        if("绿".equals(state)) {
            System.out.println("ERROR!!!绿灯不能直接切换为红灯。");
        }
        else if("红".equals(state)) {
            System.out.println("ERROR!!!已经是红灯无需再次切换。");
        }
        else if("黄".equals(state)){
            state = "红";
            System.out.println("OK...红亮起60秒。");
        }
    }
    public void switchToYellow() {
        //中间状态绿灯
        if("绿".equals(state)||"红".equals(state)) {
            state = "黄";
            System.out.println("OK...黄灯亮起5秒");
        }

        else if("黄".equals(state)){
            System.out.println("ERROR!!!已经是黄灯无需再次切换。");
        }
    }
```

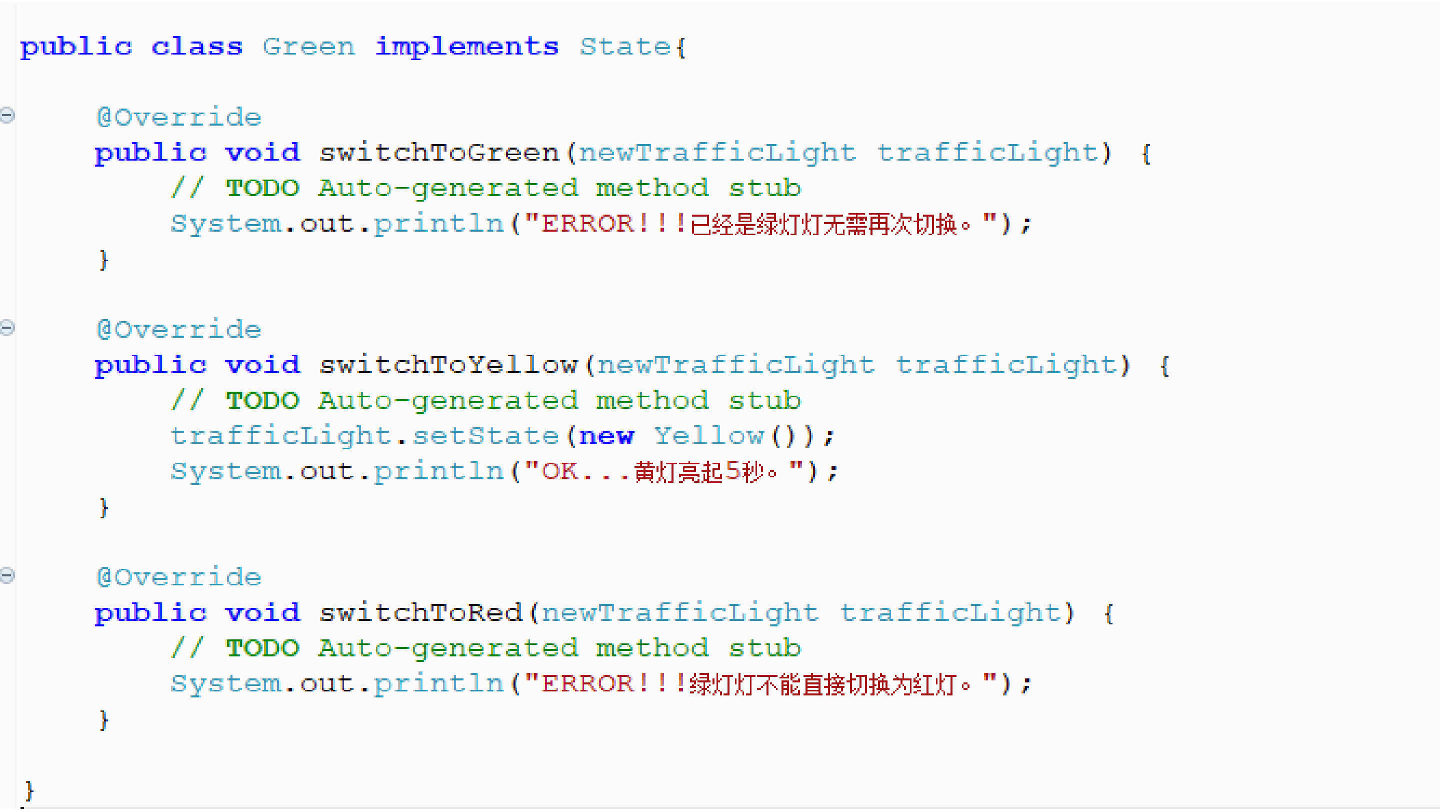
Course: Soft Design



```java
public interface State {
    void switchToGreen(newTrafficLight trafficLight);
    void switchToYellow(newTrafficLight trafficLight);
    void switchToRed(newTrafficLight trafficLight);
}
```

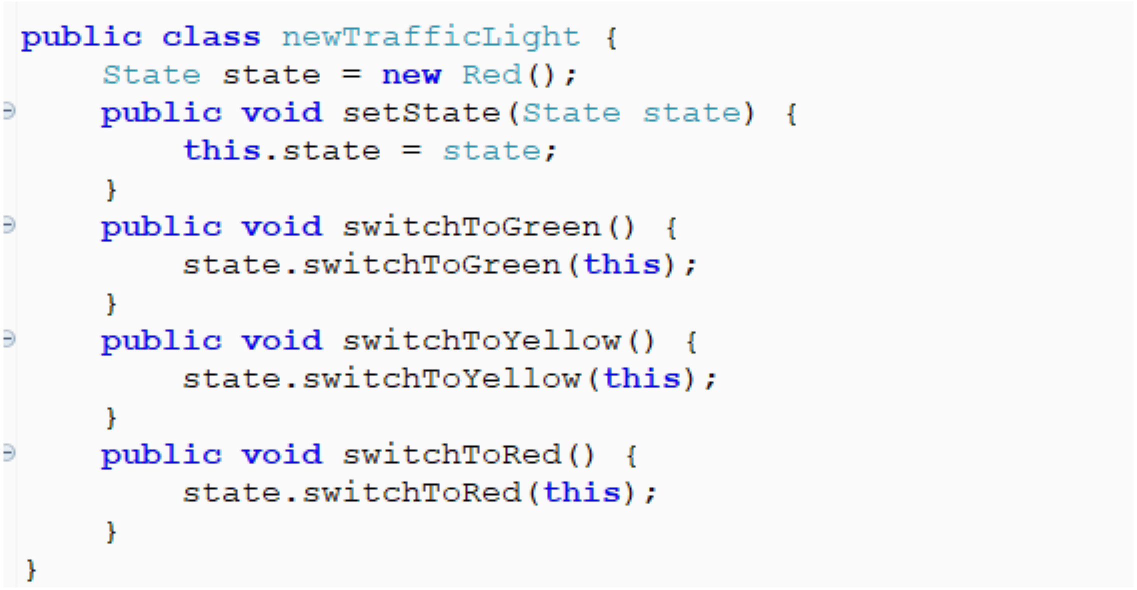


```java
public class Red implements State{

    @Override
    public void switchToGreen(newTrafficLight trafficLight) {
        // TODO Auto-generated method stub
        System.out.println("ERROR!!!红灯灯不能直接切换为绿灯。");
    }

    @Override
    public void switchToYellow(newTrafficLight trafficLight) {
        // TODO Auto-generated method stub
        trafficLight.setState(new Yellow());
        System.out.println("OK...黄灯亮起5秒。");
    }

    @Override
    public void switchToRed(newTrafficLight trafficLight) {
        // TODO Auto-generated method stub
        System.out.println("ERROR!!!已经是红灯无需再次切换。");

    }

}
```



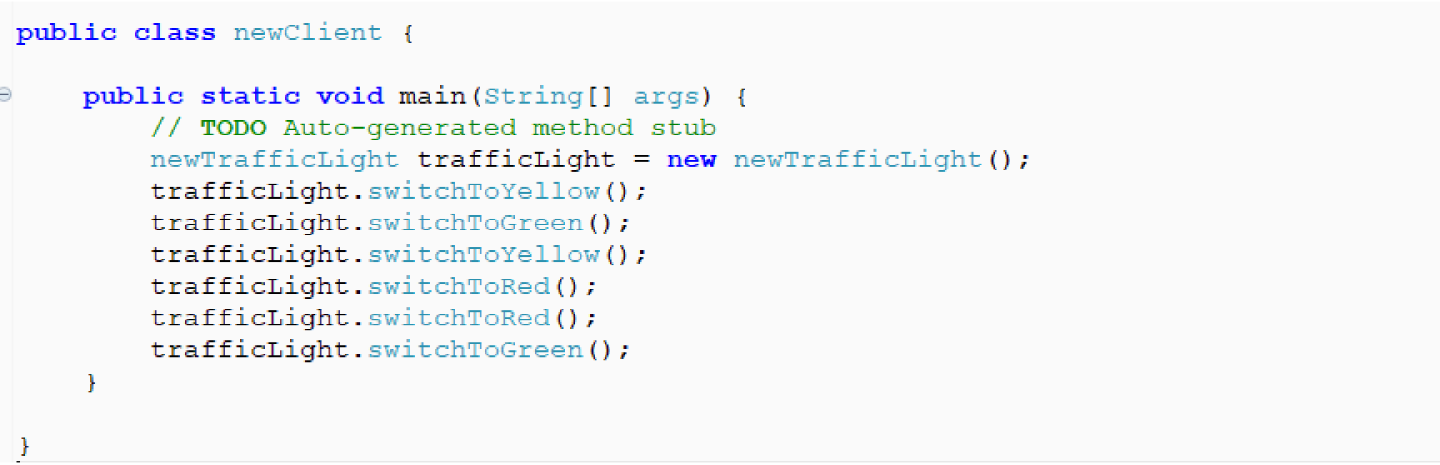
```java
public class Green implements State{

    @Override
    public void switchToGreen(newTrafficLight trafficLight) {
        // TODO Auto-generated method stub
        System.out.println("ERROR!!!已经是绿灯灯无需再次切换。");
    }

    @Override
    public void switchToYellow(newTrafficLight trafficLight) {
        // TODO Auto-generated method stub
        trafficLight.setState(new Yellow());
        System.out.println("OK...黄灯亮起5秒。");
    }

    @Override
    public void switchToRed(newTrafficLight trafficLight) {
        // TODO Auto-generated method stub
        System.out.println("ERROR!!!绿灯灯不能直接切换为红灯。");
    }

}
```

Course: Soft Design

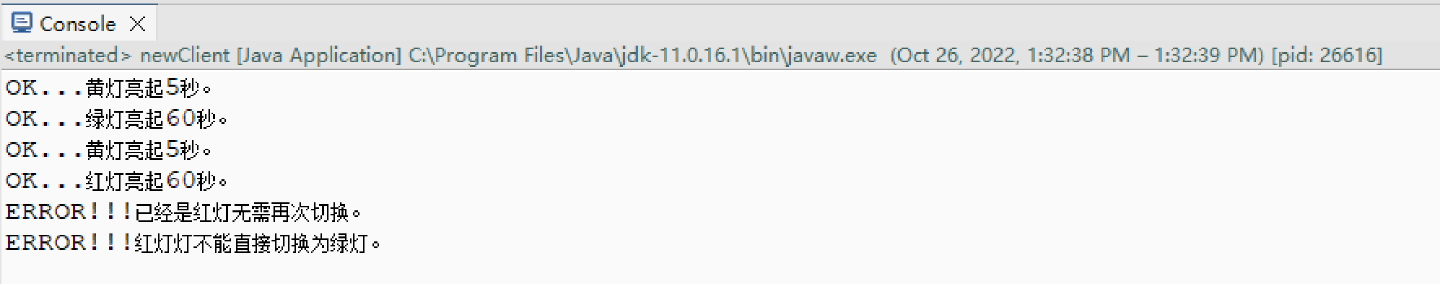


```java
public class newTrafficLight {
    State state = new Red();
    public void setState(State state) {
        this.state = state;
    }
    public void switchToGreen() {
        state.switchToGreen(this);
    }
    public void switchToYellow() {
        state.switchToYellow(this);
    }
    public void switchToRed() {
        state.switchToRed(this);
    }
}
```

Course: Soft Design



```java
public class newClient {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        newTrafficLight trafficLight = new newTrafficLight();
        trafficLight.switchToYellow();
        trafficLight.switchToGreen();
        trafficLight.switchToYellow();
        trafficLight.switchToRed();
        trafficLight.switchToRed();
        trafficLight.switchToGreen();
    }

}
```



Console

```
<terminated> newClient [Java Application] C:\Program Files\Java\jdk-11.0.16.1\bin\javaw.exe  (Oct 26, 2022, 1:32:38 PM – 1:32:39 PM) [pid: 26616]
OK...黄灯亮起5秒。
OK...绿灯亮起60秒。
OK...黄灯亮起5秒。
OK...红灯亮起60秒。
ERROR!!!已经是红灯无需再次切换。
ERROR!!!红灯灯不能直接切换为绿灯。
```

Memento (备忘录模式)

Intent

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

Also known as

Token

Motivation

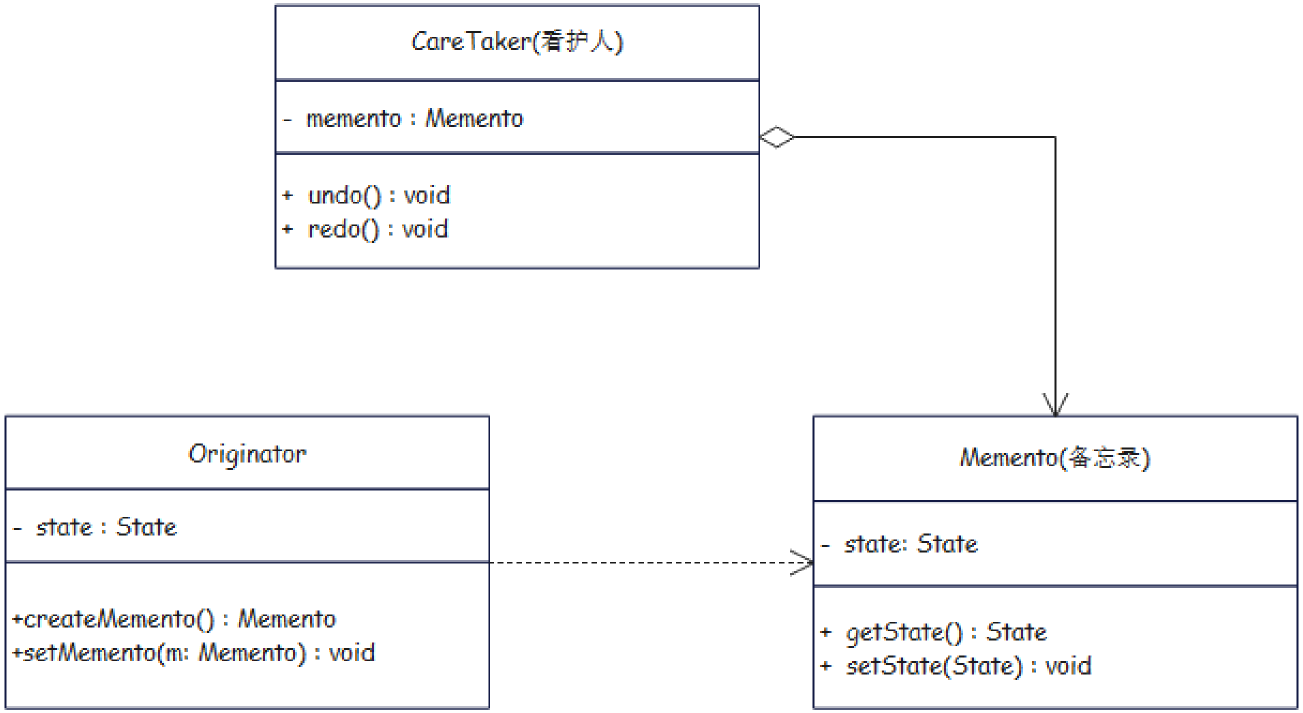
Sometimes it's necessary to record the internal state of an object. This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.

Applicability

Use the Memento pattern when

- ➢ a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, *and*
- ➢ a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

* Class Diagram



Participants

Memento

- stores internal state of the Originator object.
- protects against access by objects other than the originator.

Originator

- creates a memento containing a snapshot of its current internal state.
- uses the memento to restore its internal state.

CareTaker

- is responsible for the memento's safekeeping.
- never operates on or examines the contents of a memento.

Collaborations

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator, as the following interaction diagram
- Mementos are passive. Only the originator that created a memento will assign or retrieve its state.

Consequences

The Memento pattern has several consequences:

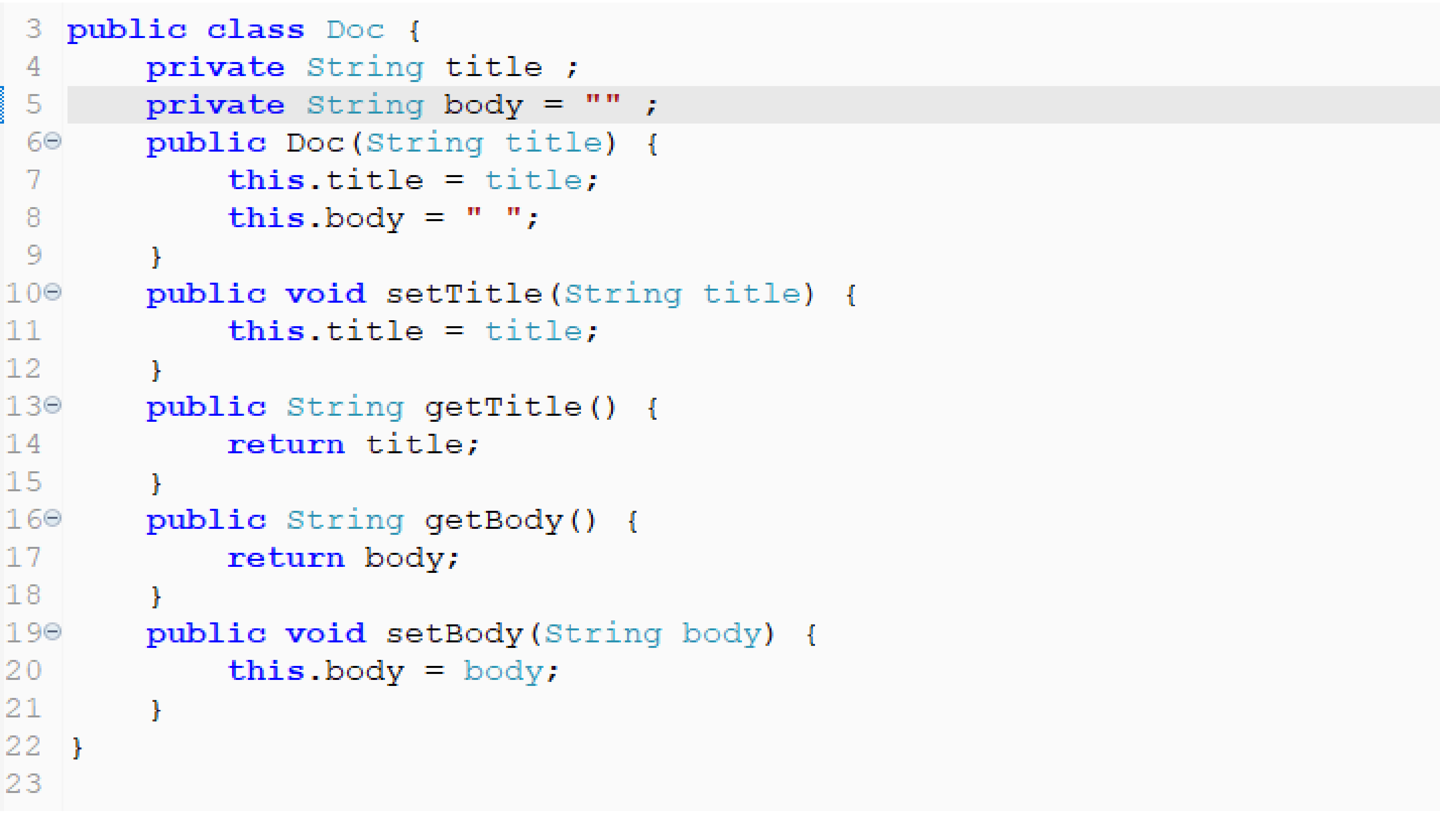
- ◆ Preserving encapsulation boundaries.
- ◆ It simplifies Originator.
- ◆ Using mementos might be expensive.
- ◆ Defining narrow and wide interfaces.
- ◆ Hidden costs in caring for mementos.

Implementation

Here are two issues to consider when implementing the Memento pattern:

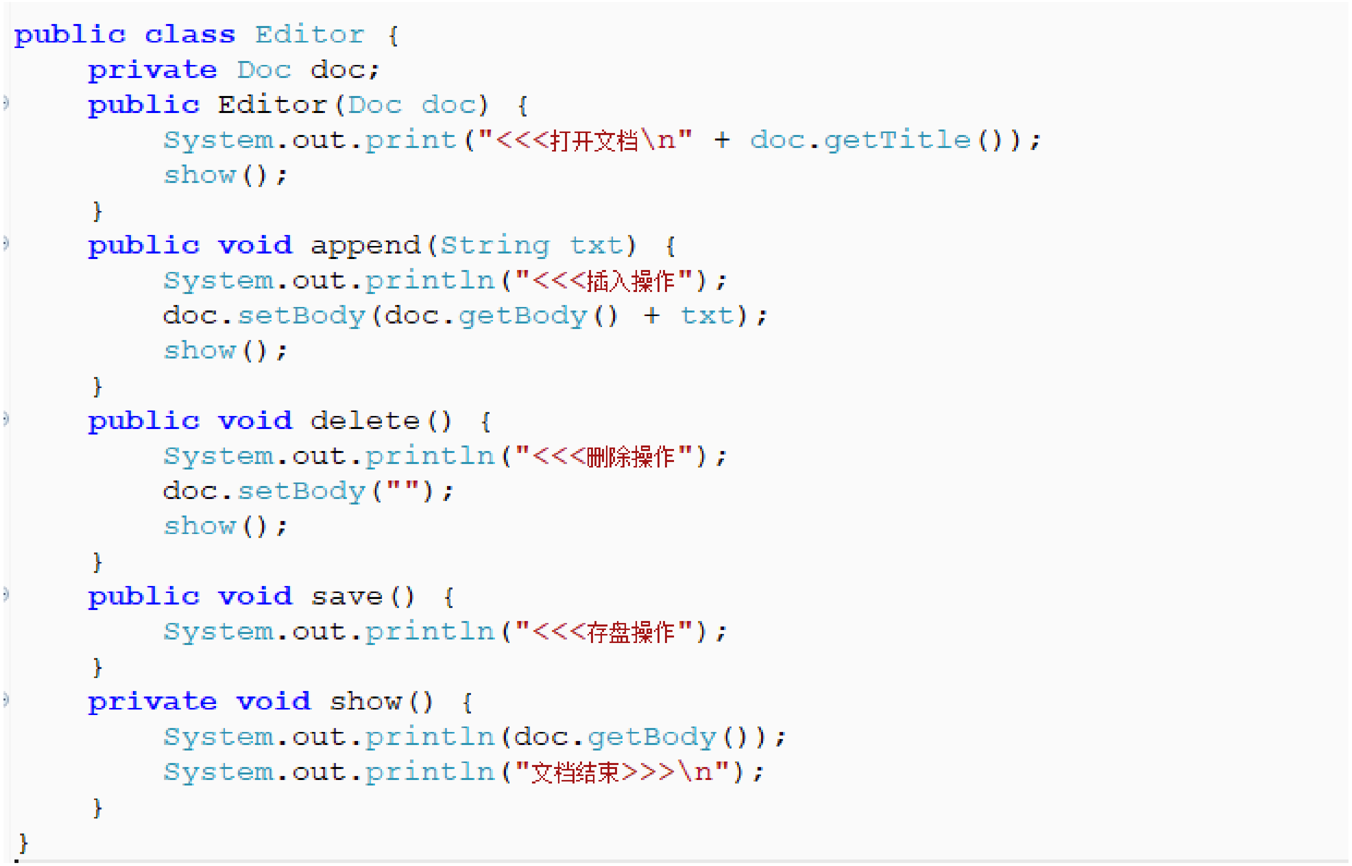
- *Language support.*
- Storing incremental changes.

Course: Soft Design



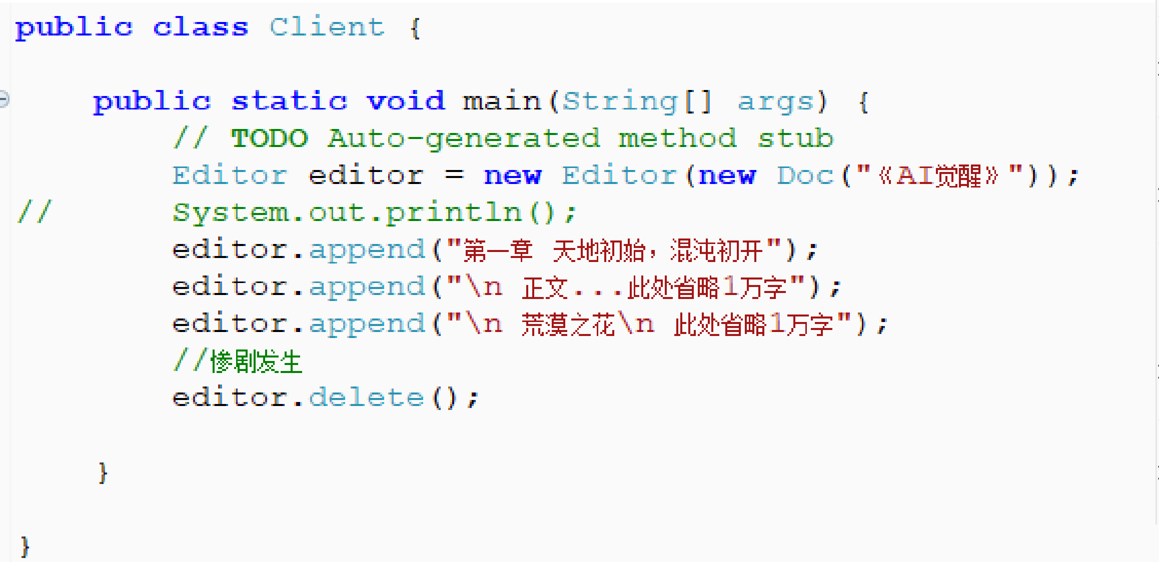
```java
public class Doc {
    private String title ;
    private String body = "" ;
    public Doc(String title) {
        this.title = title;
        this.body = " ";
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getTitle() {
        return title;
    }
    public String getBody() {
        return body;
    }
    public void setBody(String body) {
        this.body = body;
    }
}
```

Course: Soft Design

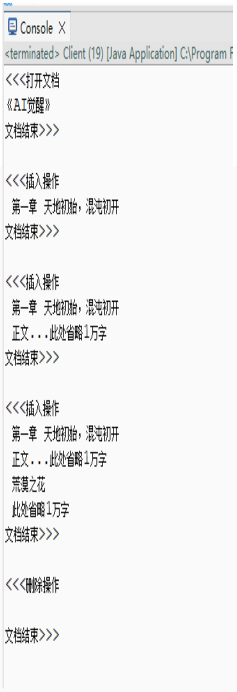


```java
public class Editor {
	private Doc doc;
	public Editor(Doc doc) {
		System.out.print("<<<打开文档\n" + doc.getTitle());
		show();
	}
	public void append(String txt) {
		System.out.println("<<<插入操作");
		doc.setBody(doc.getBody() + txt);
		show();
	}
	public void delete() {
		System.out.println("<<<删除操作");
		doc.setBody("");
		show();
	}
	public void save() {
		System.out.println("<<<存盘操作");
	}
	private void show() {
		System.out.println(doc.getBody());
		System.out.println("文档结束>>>\n");
	}
}
```

Course: Soft Design

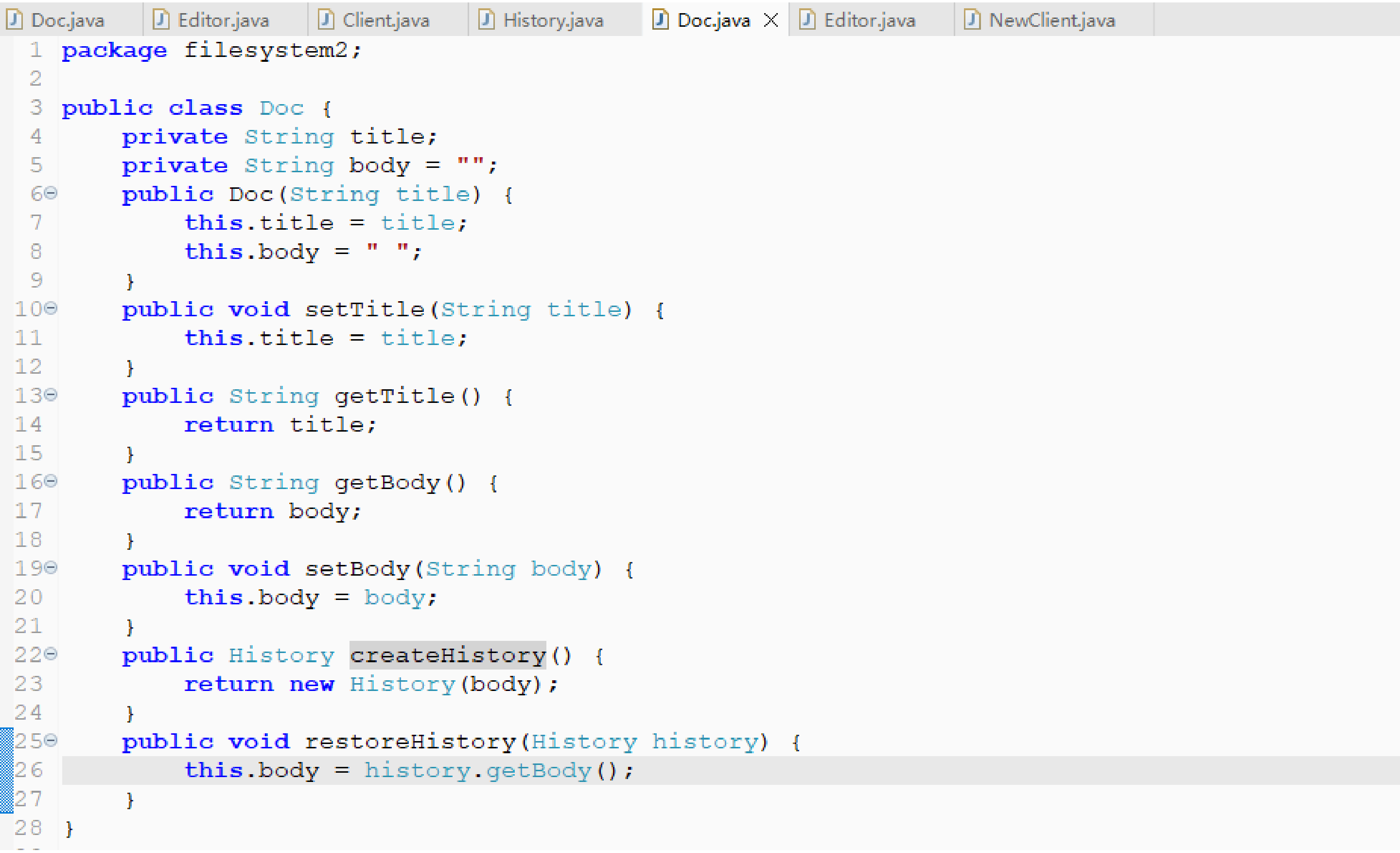


```
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Editor editor = new Editor(new Doc("《AI觉醒》"));
//      System.out.println();
        editor.append("第一章  天地初始，混沌初开");
        editor.append("\n 正文...此处省略1万字");
        editor.append("\n 荒漠之花\n 此处省略1万字");
        //惨剧发生
        editor.delete();

    }

}
```



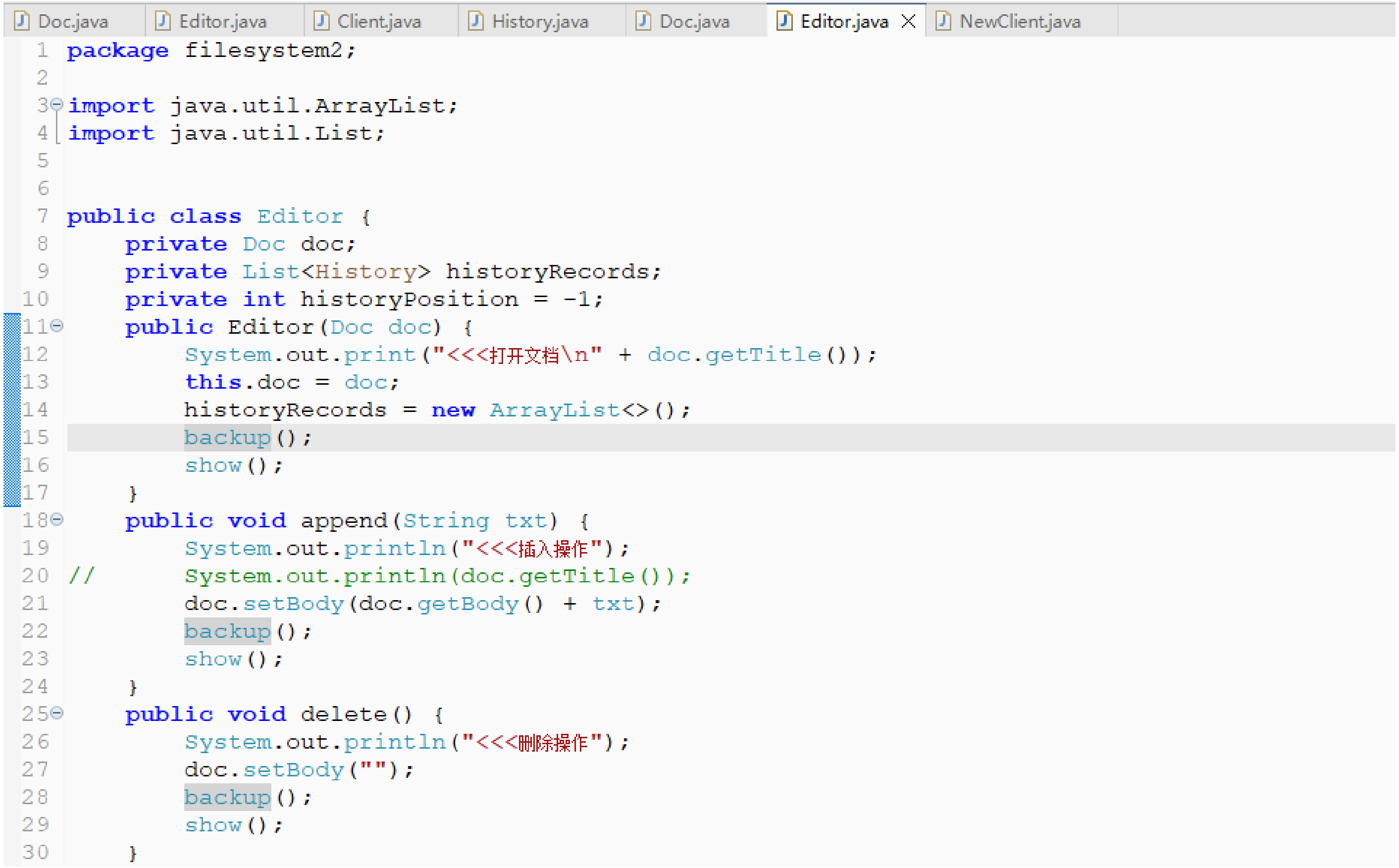
```
Console X
<terminated> Client (19) [Java Application] C:\Program F
<<<打开文档
《AI觉醒》
文档结束>>>

<<<插入操作
 第一章  天地初始，混沌初开
文档结束>>>

<<<插入操作
 第一章  天地初始，混沌初开
 正文...此处省略1万字
文档结束>>>

<<<插入操作
 第一章  天地初始，混沌初开
 正文...此处省略1万字
 荒漠之花
 此处省略1万字
文档结束>>>

<<<删除操作

文档结束>>>
```

Course: Soft Design



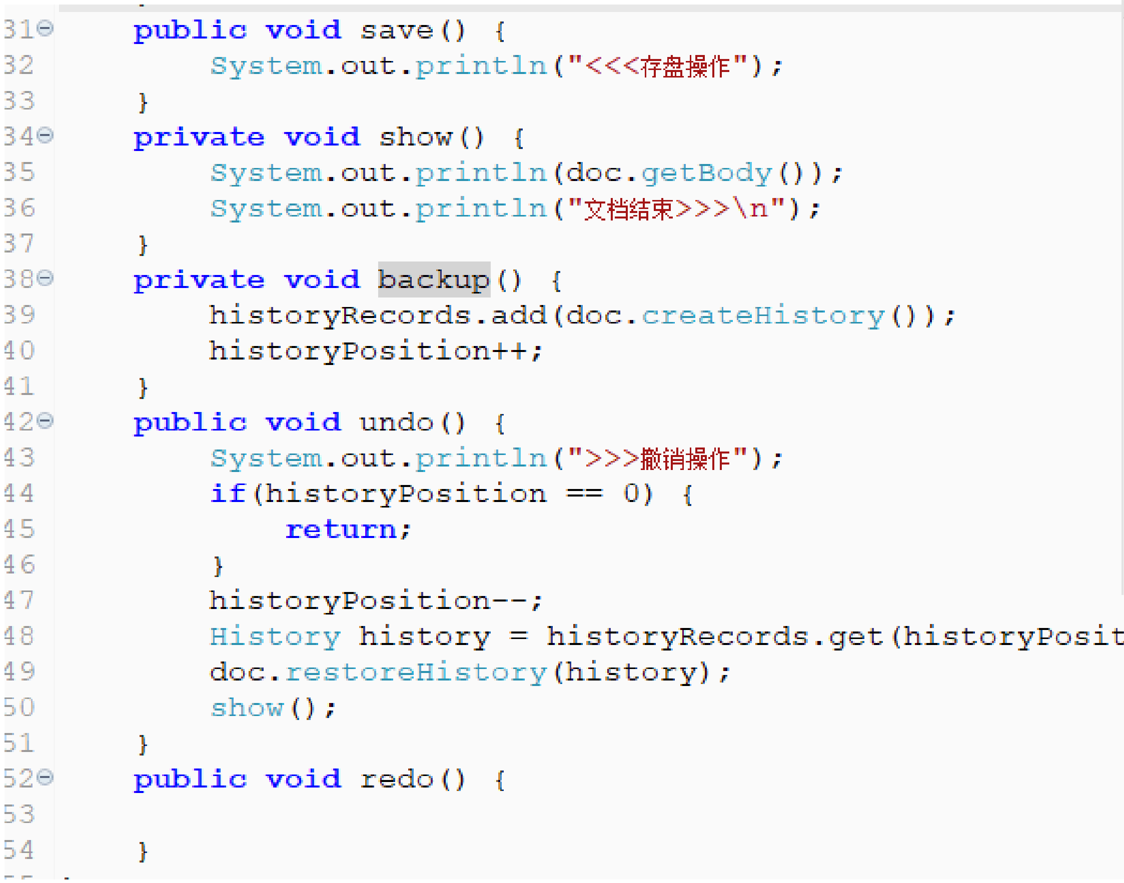
```java
package filesystem2;

public class Doc {
    private String title;
    private String body = "";
    public Doc(String title) {
        this.title = title;
        this.body = " ";
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public String getTitle() {
        return title;
    }
    public String getBody() {
        return body;
    }
    public void setBody(String body) {
        this.body = body;
    }
    public History createHistory() {
        return new History(body);
    }
    public void restoreHistory(History history) {
        this.body = history.getBody();
    }
}
```

Course: Soft Design

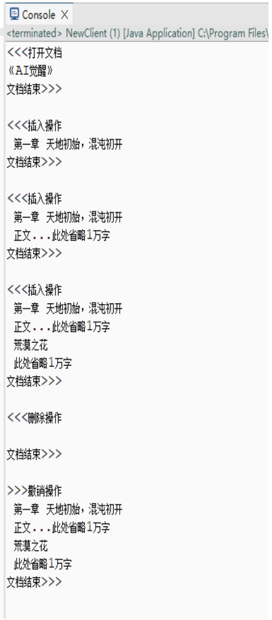


```java
package filesystem2;

import java.util.ArrayList;
import java.util.List;


public class Editor {
    private Doc doc;
    private List<History> historyRecords;
    private int historyPosition = -1;
    public Editor(Doc doc) {
        System.out.print("<<<打开文档\n" + doc.getTitle());
        this.doc = doc;
        historyRecords = new ArrayList<>();
        backup();
        show();
    }
    public void append(String txt) {
        System.out.println("<<<插入操作");
//      System.out.println(doc.getTitle());
        doc.setBody(doc.getBody() + txt);
        backup();
        show();
    }
    public void delete() {
        System.out.println("<<<删除操作");
        doc.setBody("");
        backup();
        show();
    }
```

Course: Soft Design



```java
    public void save() {
        System.out.println("<<<存盘操作");
    }
    private void show() {
        System.out.println(doc.getBody());
        System.out.println("文档结束>>>\n");
    }
    private void backup() {
        historyRecords.add(doc.createHistory());
        historyPosition++;
    }
    public void undo() {
        System.out.println(">>>撤销操作");
        if(historyPosition == 0) {
            return;
        }
        historyPosition--;
        History history = historyRecords.get(historyPosition);
        doc.restoreHistory(history);
        show();
    }
    public void redo() {

    }
```



Console
<terminated> NewClient (1) [Java Application] C:\Program Files\

```
<<<打开文档
《AI觉醒》
文档结束>>>

<<<插入操作
 第一章  天地初始，混沌初开
文档结束>>>

<<<插入操作
 第一章  天地初始，混沌初开
 正文...此处省略1万字
文档结束>>>

<<<插入操作
 第一章  天地初始，混沌初开
 正文...此处省略1万字
 荒漠之花
 此处省略1万字
文档结束>>>

<<<删除操作

文档结束>>>

>>>撤销操作
 第一章  天地初始，混沌初开
 正文...此处省略1万字
 荒漠之花
 此处省略1万字
文档结束>>>
```

Principle 116

A test case includes expected results

Documentation for a test case must include the detailed description of the expected correct results. If these are omitted, there is no way for the tester to determine whether the software succeeded or failed. Furthermore, a tester may assess an incorrect result as correct because there is always a subconscious desire to see a correct result. Even worse, a tester may assess a correct result as incorrect, causing a flurry of designer and coder activity to "repair" the correct code.

Principle 117

Test invalid inputs

It is natural and common to produce test cases for as many acceptable input scenarios as possible. What is equally important—but also uncommon——is to produce an extensive set of test cases for all invalid or unexpected inputs.

Principle 118

Always stress test

Software design often behaves just fine when confronted with"normal"loads of inputs or stimuli. The true test of software is whether it can stay operational when faced with severe loads.

Principle 119

Get it right before you make it faster

As a project nears its delivery deadline and the software is not ready, desperation often takes over. Suppose the schedule called for two months of unit testing, two months of integration testing, and two months of software system testing. It is now one month from the scheduled delivery. Suppose 50 percent of the components have been unit-tested. A back-of-the-enve-lope calculation indicates that you are five months behind schedule. You have two choices:

1. Admit the five-month delay to your customer: Ask for a postponement.
2. 2. Put all the components together (including the 50 percent not yet unit-tested) and hope for the best.

Principle 120

Ase MCCABE complexity measure

Although many metrics are available to report the inherent complexity of software, none is as intuitive and as easy-to-use as Tom McCabe's cyclomatic number measure of testing complexity. Although not absolutely fool-proof, it results in fairly consistent predictions of testing difficulty. Simply draw a graph of your program, in which nodes correspond to sequences of instructions and arcs correspond to nonsequential flow of control.

Principle 121

Use effective test completion measures

Many projects proclaim the end of testing when they run out of time. This may make political sense, but it is irresponsible. During test planning, define a measure that can be used to determine when testing should be completed.If you have not met your goal when time runs out,you can still make the choice of whether to ship the product or slip the milestone, but at least you know whether you are shipping a quality product.

Principle 122

Achieve effective test coverage

In spite of the fact that testing cannot prove correctness, it is still important to do a thorough job of testing. Metrics exist to determine how thoroughly the code was exercised during test plan generation or test execution.

Principle 123

Don't integrate before unit testing

Under normal circumstances components are separately unit-tested. As they pass their unit tests, a separate organization integrates them into meaningful sets to exercise their interfaces. Components that have not been separately unit-tested are often integrated into the subsystem in a vain attempt to recapture a lost schedule. Such attempts actually cause more schedule delays. This is because a failure of a subsystem to satisfy an integration test plan may be caused now either by a fault in the interface or by a fault in the previously untested component. And much time is spent trying to determine which is the cause.

Principle 124

Instrument your software

When testing software, it is often difficult to determine why the software failed. One way of uncovering the reasons is to instrument your software, that is, embed special instructions in the software that report traces, anomalous conditions, procedure calls, and the like. Of course, if your debugging system provides these capabilities, don't instrument manually.

Principle 125

Analyze causes for errors

Errors are common in software. We spend enormous amounts of resources detecting and fixing them. It is far more cost-effective to reduce their impact by preventing them from occurring in the first place. One way to do this is to analyze the causes for errors as they are detected. The causes are broadcast to all developers with the idea being that we are less apt to make an error of the same type as one whose cause was thoroughly analyzed and learned from.

Principle 126

Don't take errors personally

Writing software requires a level of detail and perfection that no human can reach. We should dedicate ourselves to constant improvement, not perfection. When an error is detected in your code either by you or by others, discuss it openly. Instead of castigating yourself, use it as a learning experience for yourself and others (see more on this in Principle 125).

Management is the set of activities for planning, controlling, monitoring,100% and reporting on all the engineering activities that encompass software development.

Management principles

Management is the set of activities for planning, controlling, monitoring,100% and reporting on all the engineering activities that encompass software development.

Principle 127

Good management is more important than good technology

Good management motivates people to do their best. Poor management demotivates people. All the great technology (CASE tools, techniques, computers, word processors, and the like) will not compensate for poor management. And good management can actually produce great results even with meager resources. Successful software startups do not become successful because they have great process or great tools (or great products for that matter!).Most have been successful because of great management and great marketing

Principle 128

Use appropriate solutions

A technical problem needs a technical solution. A management problem needs a management solution. A political problem needs a political solution. Do not try to throw an inappropriate solution at a problem.

Principle 129

Don't belive everything you read

As a general rule, people who believe in a particular philosophy search for data that supports that philosophy and discard data that does not. Someone who wants to convince others of a position obviously uses supportive, not unsupportive, data.

Principle 130

Understand the customers' priorities

If you are communicating with your customers, you should be sure you know their priorities. These can easily be recorded in the requirements specification (Principle 50), but the real challenge is to understand the possibly ever shifting priorities. In addition, you must understand the customers' interpretation of "essential,""desirable," and "optional." Will they really be happy with a system that satisfies none of the desirable and optional requirements?

Principle 131

People are the key to success

Highly skilled people with appropriate experience, talent, and training are key to producing software that satisfies user needs on time and within the budget. The right people with insufficient tools, languages, and process will succeed. The wrong people (or the right people with insufficient train-ing or experience) with appropriate tools, languages, and process will probably fail.

Principle 132

A few good people are better than many less skilled people

This follows immediately from Principle 131, which says that you should always hire the best engineers. This principle says that you are better off allocating just a few good, experienced engineers on a critical task than to put many inexperienced engineers on it.

Principle 133

Listen to your people

The people who work for you must be trusted. If they're not trustworthy (or if you don't trust them), your project will fail. If they don't trust you, your project will also fail. Your people can tell as quickly that you don't trust them as you can when your boss doesn't trust you.

Principle 134

Trust your people

In general, if you trust people, they will be trustworthy. If you treat people as if you don't trust them, they will give you reason not to trust them. When you trust others and give them no reason not to trust you, they will trust you. Mutual trust is essential for successful management.

Principle 135

Expect excellence

Your employees will do much better if you have high expectations of them. Studies by Warren Bennis prove conclusively that, the more you expect, the more results will be achieved (obviously with some limit).In many experiments, heterogeneous groups were divided into two subgroups with identical goals. One subgroup was treated as if excellence was expected. The other subgroup was treated as if mediocrity was expected. In every experiment, the group for whom excellence was expected outperformed the other group.

Principle 136

Get it right before you make it faster

When recruiting personnel for your project, don't underestimate the importance of teamwork and communication. The best designer becomes a poor asset if he/she is unable to communicate,convince,listen,and compromise.