LECTURE 01

Object-oriented Programming and UML

Software Development Principle 1~19

叶奔　　Ben

Office: A307b

E-mail: bye@must.edu.mo
zhyeben@126.com

Assessment Method

1. Attendance: 10%
2. Assignment: 10%
3. Midterm: 30%
3. Final exam: 50%

Attendance: Answer questions +1.
Assignment: 3 times.
Midterm Exam: 2022-11.
Final Exam: Undetermined

Review-UML(Class diagram)

古人將知識分爲“技”和“道”，習技固然可以成爲人傑，而悟道才能羽化升仙。

Object oriented programming is a more advanced programming method than process oriented programming （error）

Object oriented and process oriented are both software technologies, which adopt different methods to understand and describe the world. No school has said that process oriented is not as good as object-oriented. The main reason for designing object-oriented programs and methods is that the system has reached the complexity beyond its processing capacity.

In fact, object-oriented thinking is not complicated, but it is not easy for students who are used to understanding the world through process methods to fully understand and accept object-oriented thinking.

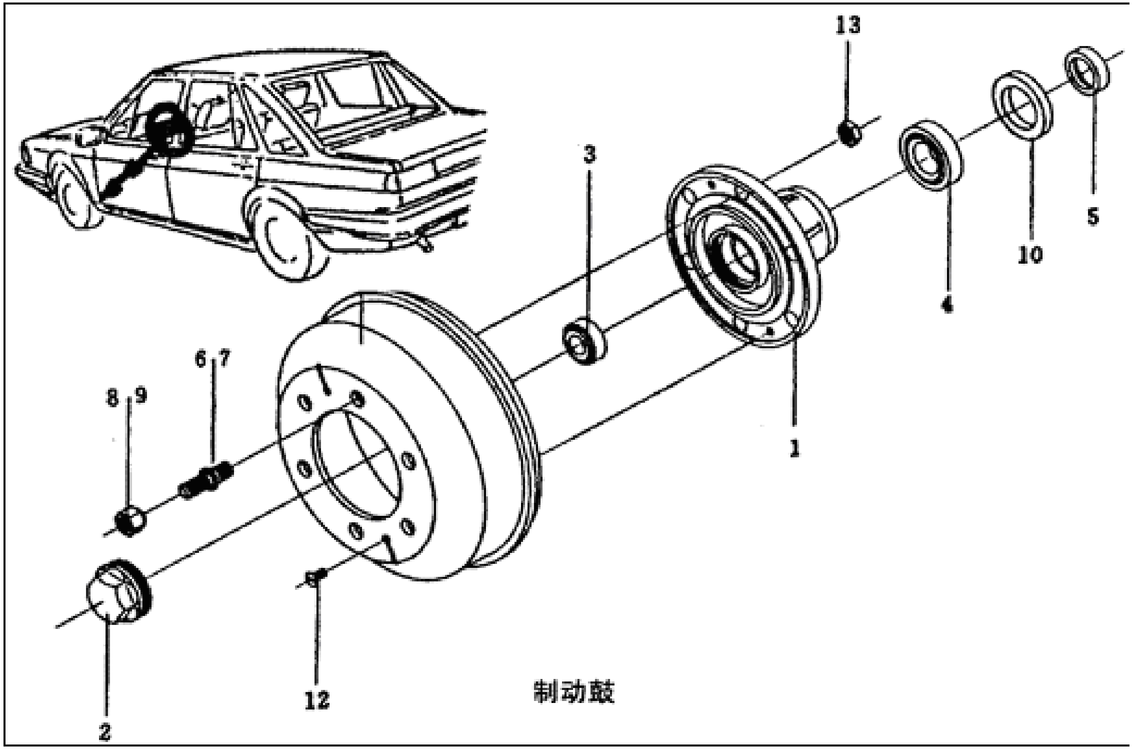
Process oriented emphasizes the integrity of events, while object-oriented emphasizes the separation of events and the relevance of cooperation.

Object-oriented Difficulties

When there are too many factors constituting a system, it is very difficult to analyze all the relevant causes and logic of a complex system.

e.g. In automobile manufacturing, we manufacture various small parts and small units. When the market changes and the business strategy changes, a new model can be quickly produced by changing small parts.

Course: Soft Design



Basic Concept of Object-oriented

Object oriented -> view the world as independent objects. Only under the drive of some external force can the objects transfer information according to certain laws. (the transfer of information between objects constitutes a "process")

An object has a hard shell. From the outside, except for the message channel that can be used to communicate with the outside world, the object is a black box inside, and nothing can be seen. This is called encapsulation.

Subclass objects will contain all the capabilities of the parent class, which is called inheritance.

Each object has multiple appearances, which can show different appearances under different circumstances, but there is only one essence, which is the interface.

Objects may have the same face, but behind the same face are different objects. They have different behaviors, which is polymorphism.

The object only knows a group of small partners around him, which is called dependency; And maintain information interaction with small partners, which is called coupling.

The process of finding similarity between objects becomes abstract.

Some objects are not only used for a special function, but also can be used in other places, which is called multiplexing.

Three problems of constructing abstraction

Why: Why should I be so abstract instead of so abstract.

How: How to know that a certain combination just meets the needs of the real world.

What: How can we understand what it means.

Abstract is not only the essence of object-oriented, but also the difficulty of object-oriented.

① A method of mapping the real world to the object world.
② A method of describing the real world from the object world.
③ A method of verifying whether the object world behavior correctly reflects the real world.

UML（統一建模語言）

UML is a kind of modeling language. UML uses a "visual" graphical way to define the language.

Through its meta model and representation, UML uses simple and intuitive graphics to express and expose, accurately and intuitively describe complex meanings, and turning words into graphics is the real meaning of UML visualization.

Course: Soft Design

Modeling: it is a model established to solve a certain problem. It is analyzed and predicted through mathematical calculation to find out the way to determine the problem.

Theoretically, establishing a model means:

- Create a abstract function
- Get an understanding of things themselves
- Conceptualization

Internal structure & working principle

Modeling formula:

a) Problem area =$\sum_1^n$ Abstract angle
b) Abstract angle = Business objectives of participants outside the boundaries of the problem area = Business use case
c) Business use case = $\sum_1^n$ Specific scenarios
d) Specific scenarios = Static transaction + Specific conditions +Specific action
e) Specific Things = Specific transaction+Specific rule+Specific Human-behavior

UML Core view

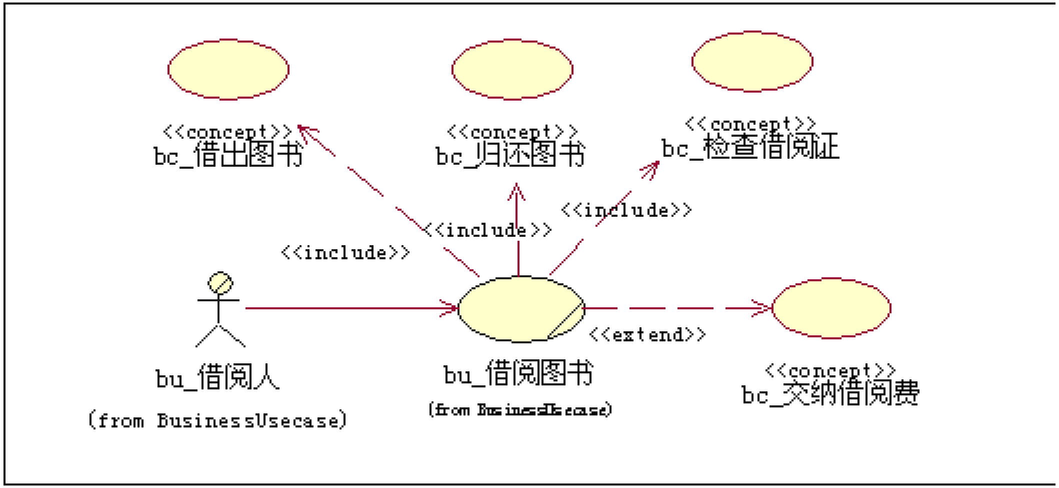
Static view:

- Use case diagram
- Class diagram
- Package diagram

Dynamic view:

- Activity diagram
- State diagram
- Sequence diagram
- Collaboration diagram

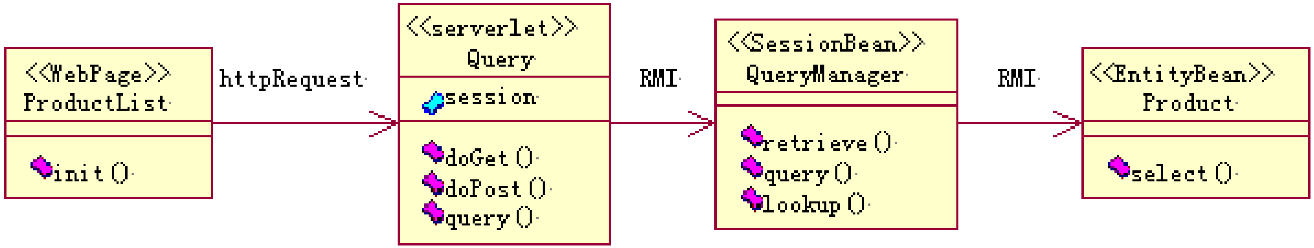
Course: Soft Design



Use case view of borrowing book concept

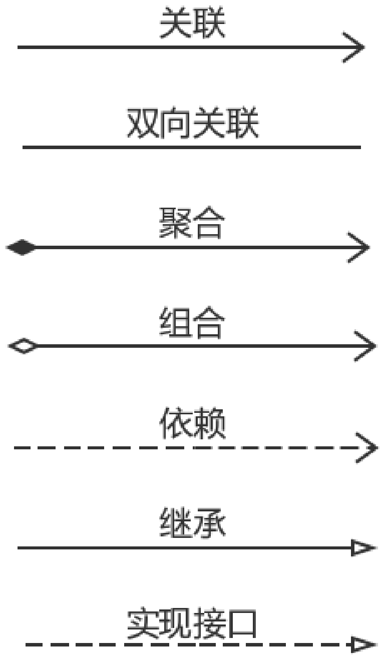
Use case diagram: Participants and forces are used as basic elements to show the functional requirements of the system from different perspectives.

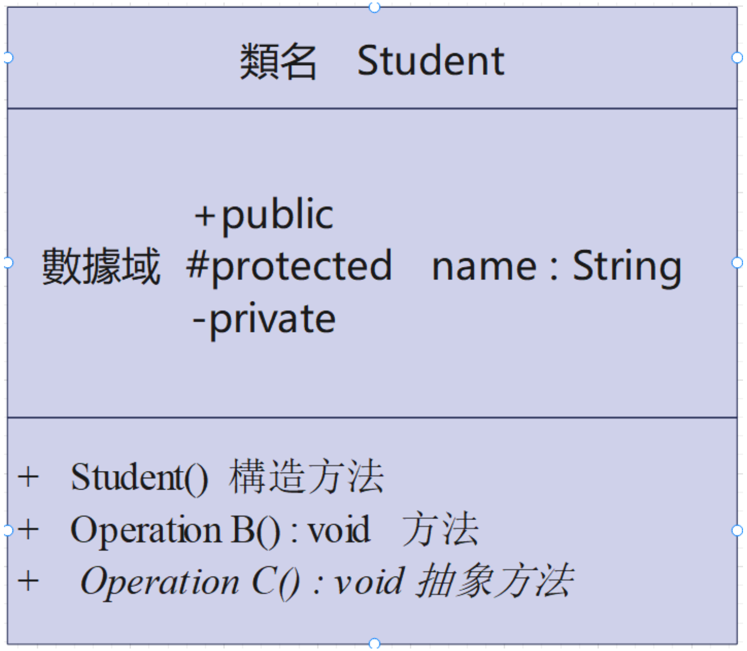
Course: Soft Design



Class diagram: used to show the classes in the system and their relationships.

類圖的使用細節


关联
双向关联
聚合
组合
依赖
继承
实现接口


類名 Student
數據域 +public #protected -private name : String
+ Student() 構造方法
+ Operation B() : void 方法
+ *Operation C() : void 抽象方法*

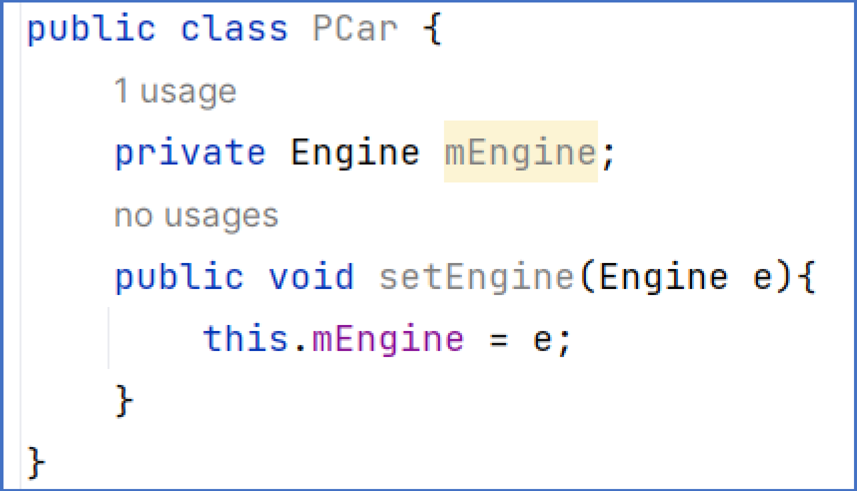
Association

```
class Student{
private Teacher mTeacher;
}
class Teacher{
}
```

Bidirectional association

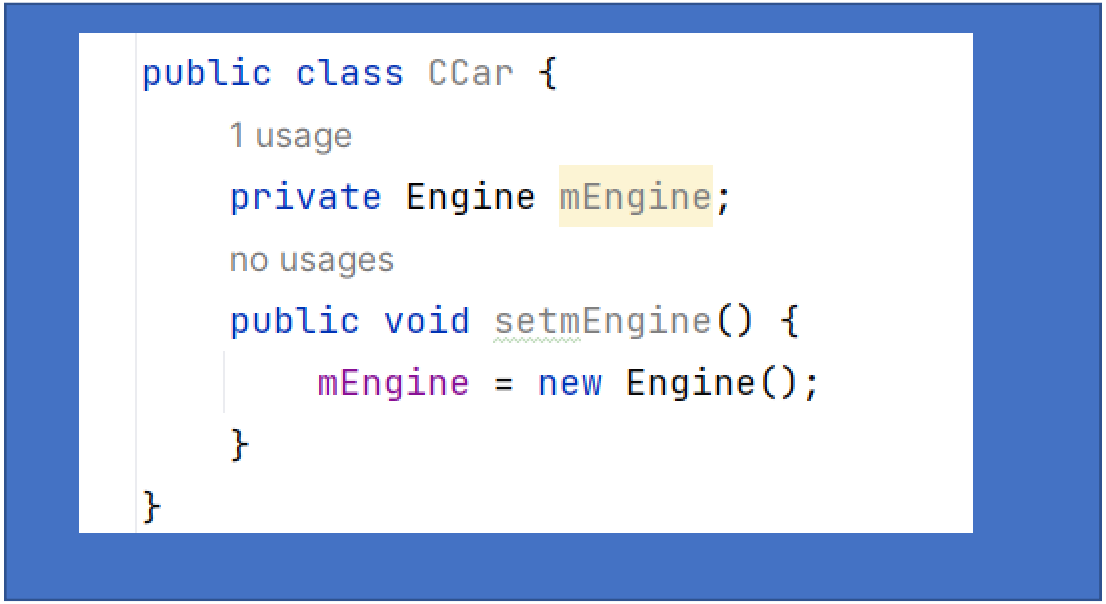
```
class Student{
private Teacher mTeacher;
}
clsass Teacher{
private Student mStuent;
}
```

Polymerization



```
public class PCar {
    1 usage
    private Engine mEngine;
    no usages
    public void setEngine(Engine e){
        this.mEngine = e;
    }
}
```

Combination

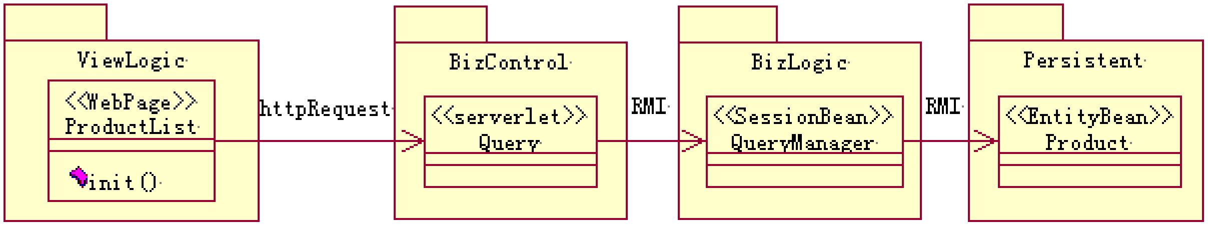


```
public class CCar {
    1 usage
    private Engine mEngine;
    no usages
    public void setmEngine() {
        mEngine = new Engine();
    }
}
```

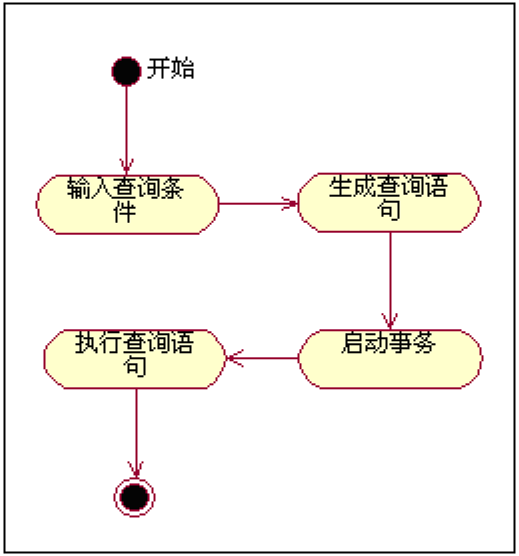
Rely on

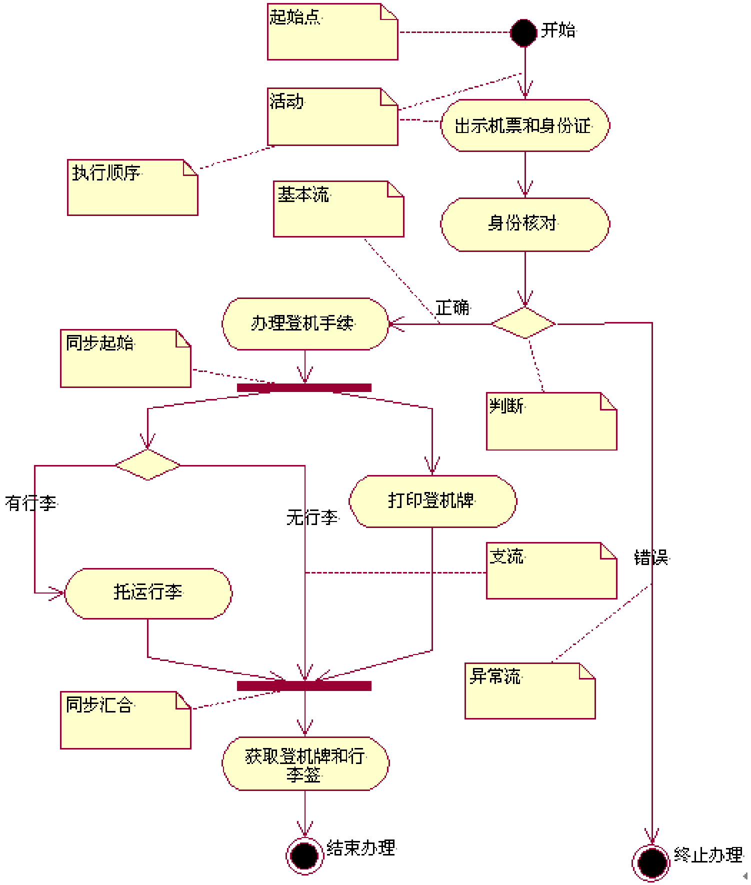
```
class Car{
private Horn mHorn;
public void whistle(){
mHorn.whistle();
}
```

Course: Soft Design



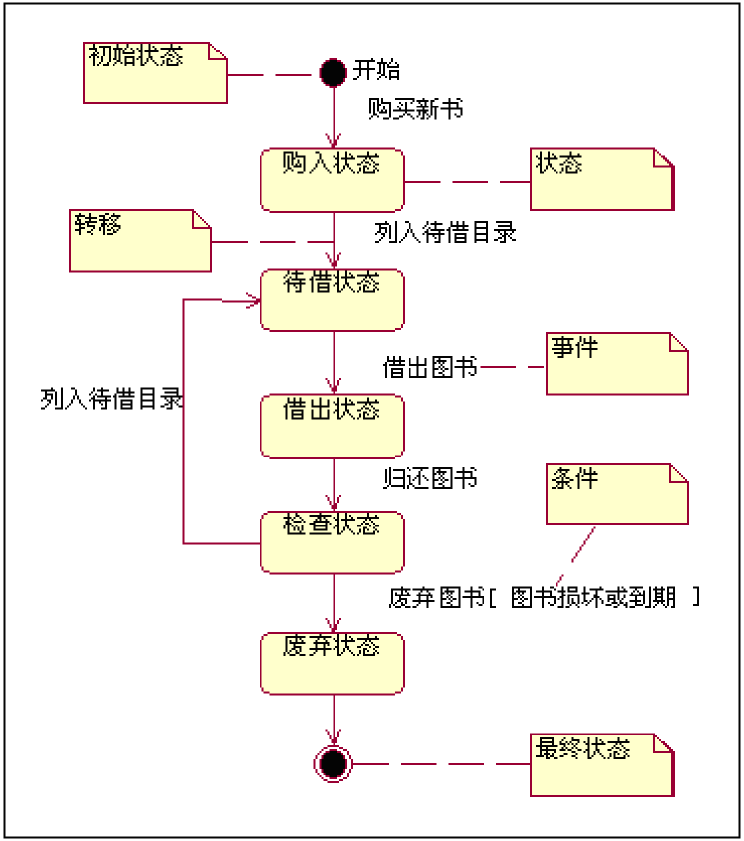
Package diagram: generally used to show high-level views.





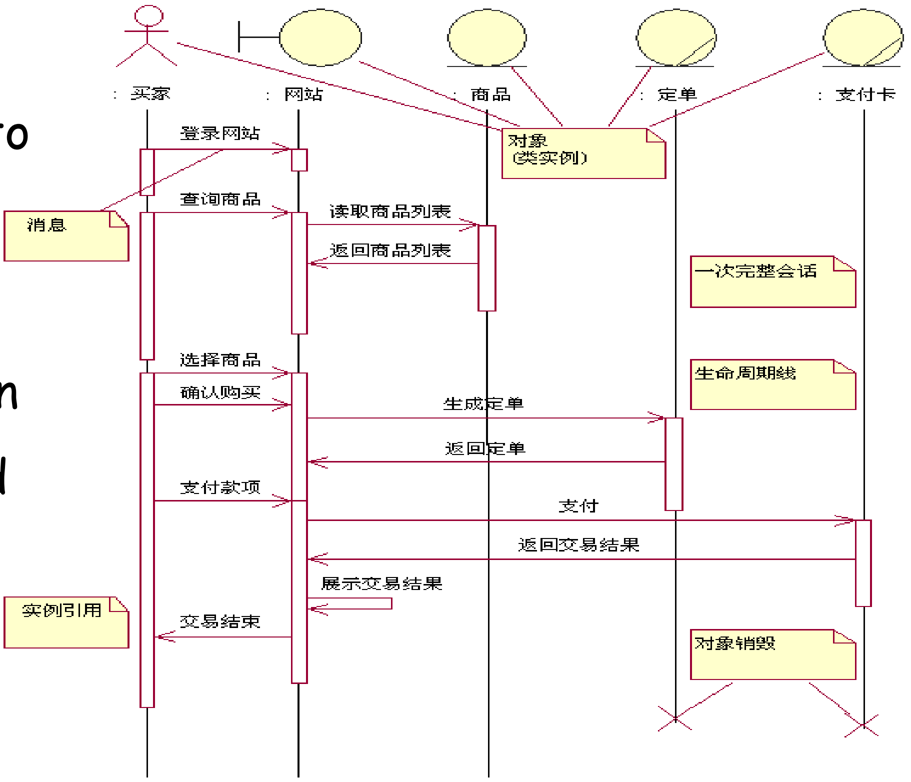
Activity diagram: describes the activities needed to complete a certain goal and the execution sequence of these activities .

State diagram: the state diagram is usually used to describe the possible states of business roles or business entities - the events leading to state transition and the changes after state transition.



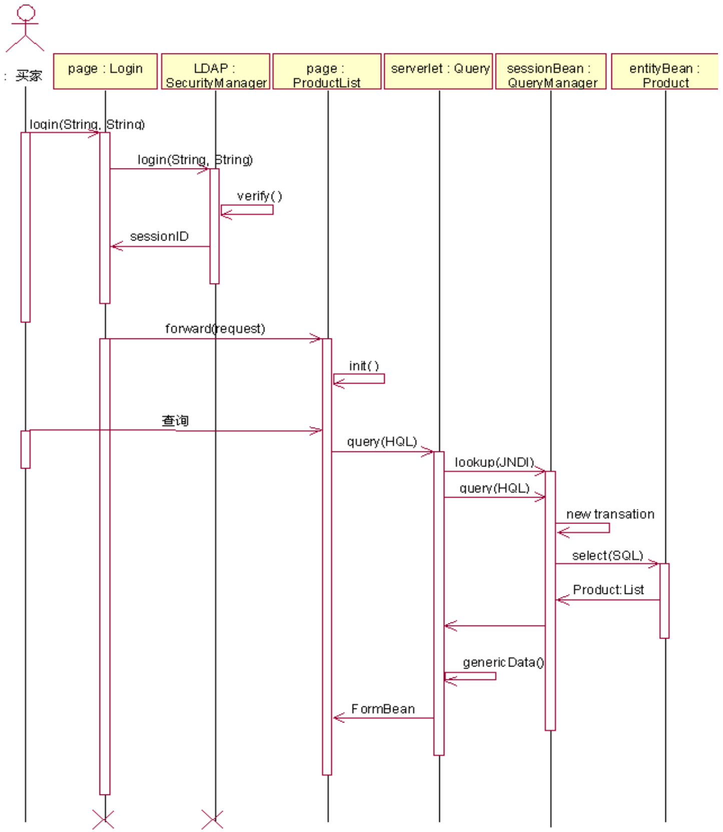
Course: Soft Design

A sequence diagram is used to describe the interaction patterns between objects arranged in chronological order.



Course: Soft Design

The collaboration diagram describes a mode of interaction between cashing countries; It displays the objects participating in the interaction through the links between the objects and the messages they send to each other.



Design pattern

software design experience

When evaluating an object-oriented system, one of the methods used is to judge whether the design of the system emphasizes the public cooperative relationship between objects.

$$\text{Design pattern} \left\{ \begin{array}{l} \text{Flexibility} \\ \\ \\ \text{Reusability} \end{array} \right.$$

Help you design more flexible, modular, reusable and easy to understand software

To design object-oriented software, we must find relevant objects, classify them with appropriate strength, define the interface and inheritance level of classes, and establish the basic relationship between objects

$$\left\{ \begin{array}{l} \text{Pertinence} \\ \\ \text{Generality} \end{array} \right.$$

Novelists and screenwriters rarely design plots from scratch, such as tragic heroes and romantic novels

Course: Soft Design

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter
				Template Method
	Object	Abstract Factory	Adapter	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Interator
		Singleton	Decorator	Mediator
			Façade	Memento
			Flyweight	Observer
				State
				Strategy
				Visitor

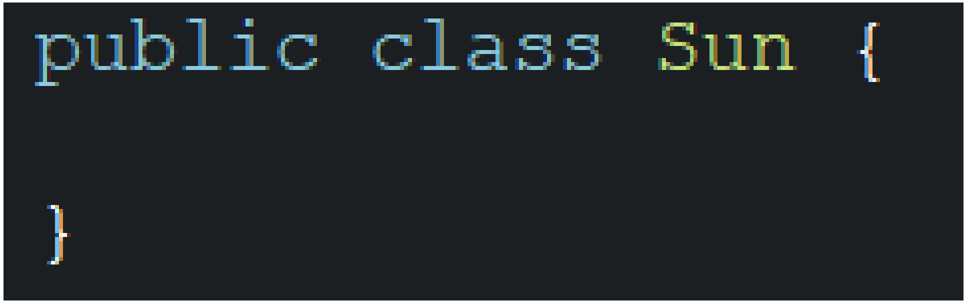
Singleton (單例模式)

This is a very simple and easy to understand design pattern.

Ensure a class has only one instance and provide a global point of access to it.

As the name implies, a single instance is a single instance. To be more precise, it means that there is only one instance in a certain system, and a centralized and unified access interface is provided to make the system behavior consistent. "A collection with only one element.".

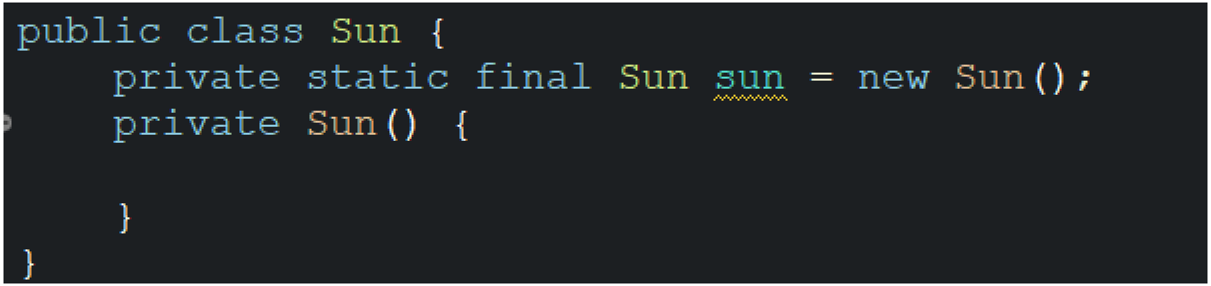
Currently, there is nothing in the sun class. Next, we have to make sure that no one can create an instance of the sun. Otherwise, once someone calls new sun(), we need to create a new Houyi class to solve this problem



```
public class Sun {

}
```

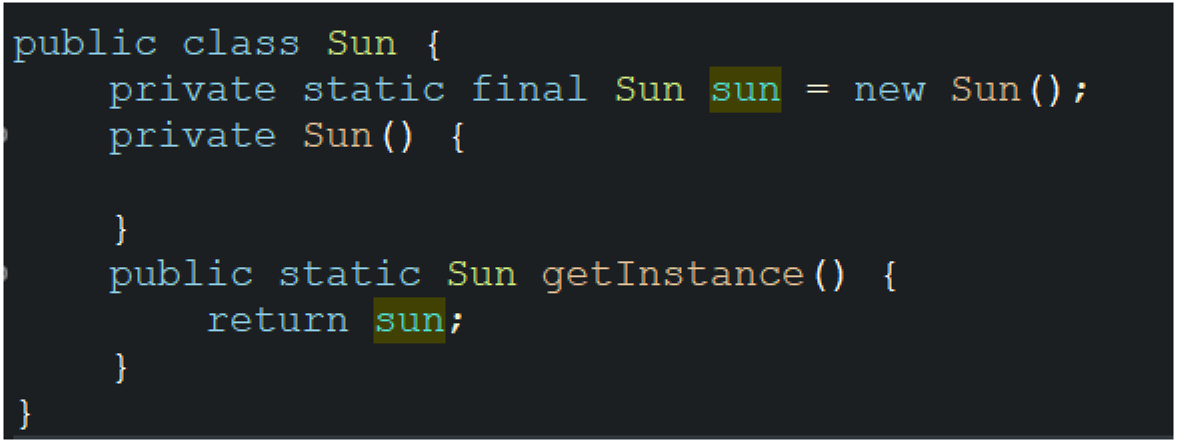
Now we set all properties and methods in the sun to be constant and private. For other objects, they are private, invisible and inaccessible.



```
public class Sun {
    private static final Sun sun = new Sun();
    private Sun() {

    }
}
```

The sun object is finished, but it is too selfish. How can the outside access it? We need to provide an external access to him.



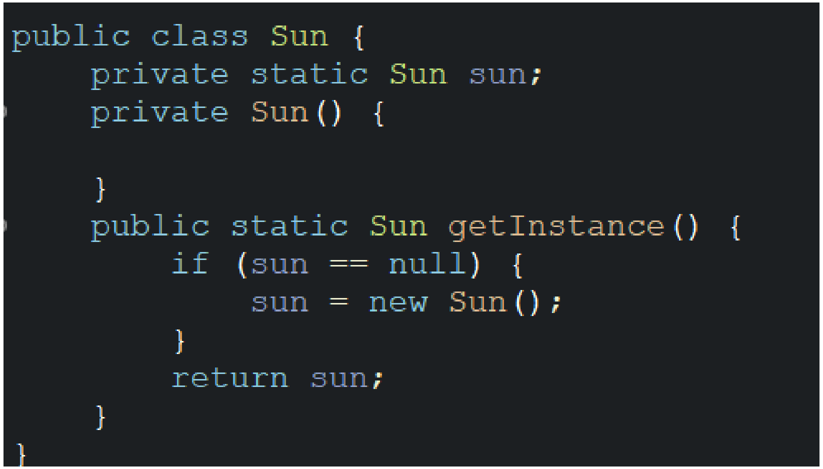
```
public class Sun {
    private static final Sun sun = new Sun();
    private Sun() {

    }
    public static Sun getInstance() {
        return sun;
    }
}
```

In this way, we create a sun that will always exist in memory. For the outside, we can only see it through the method given by the sun, and whoever gets it, or gets it several times, is the same instance of the sun.

This method of creation is called "hungry initialization". Even if it is actively instantiated in the initial stage, the requirement state is always maintained, regardless of whether the singleton is used or not.

But the class created by eager initialization is not used by anyone. Isn't that a waste of resources? In order to reduce the risk, it is stipulated that any object creation must be scheduled. This is lazy initialization, so we continue to modify it:



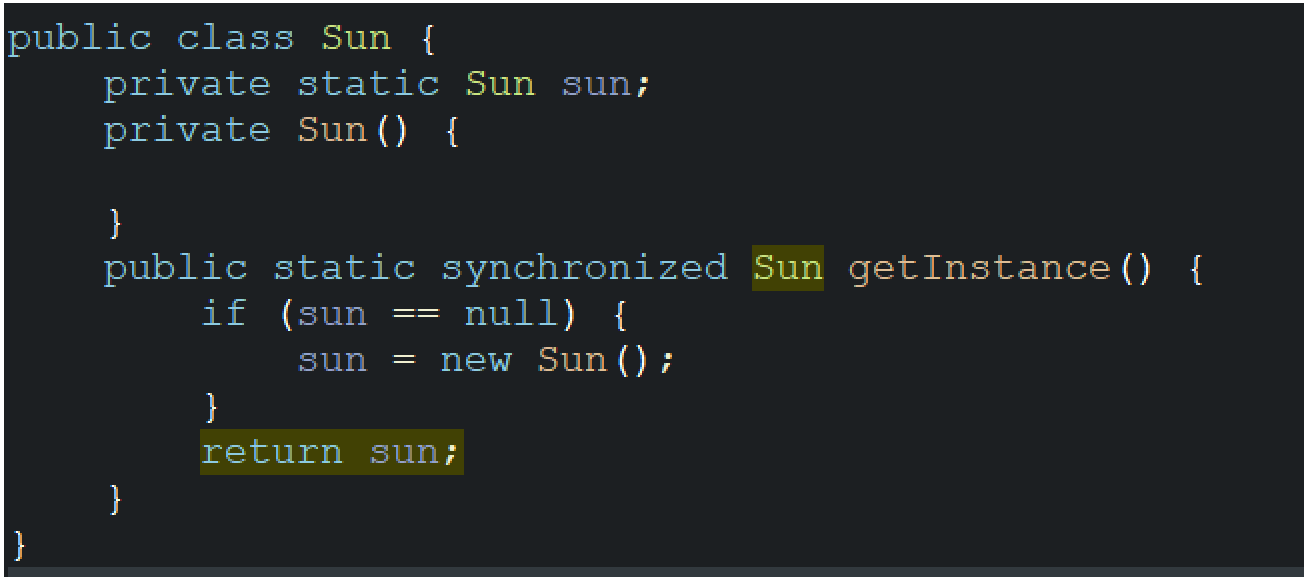
```
public class Sun {
    private static Sun sun;
    private Sun() {

    }
    public static Sun getInstance() {
        if (sun == null) {
            sun = new Sun();
        }
        return sun;
    }
}
```

Course: Soft Design

There seems to be no problem with the modified program, but if we still remember the deadlocks encountered when using the database, we will find that if we call the program with a multi-threaded method, it is likely that the empty logic and the program created by the execution will be overwritten at the same time. Therefore, we need to add a synchronization lock to the requested method, So we won't call this method at the same time when we use it!

Course: Soft Design

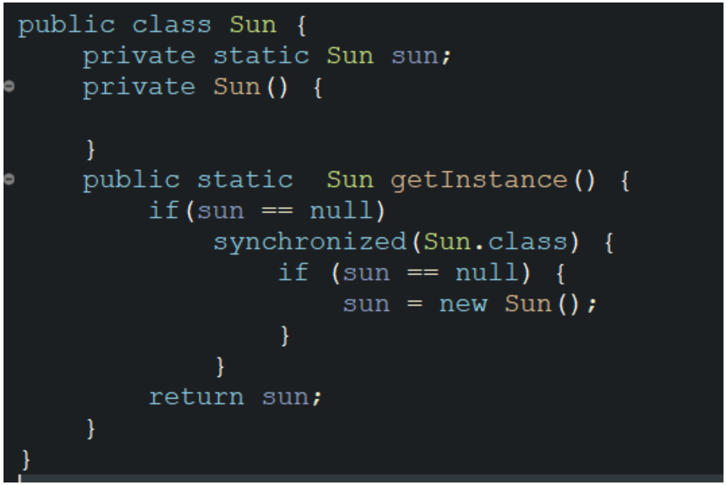


```
public class Sun {
    private static Sun sun;
    private Sun() {

    }
    public static synchronized Sun getInstance() {
        if (sun == null) {
            sun = new Sun();
        }
        return sun;
    }
}
```

Course: Soft Design

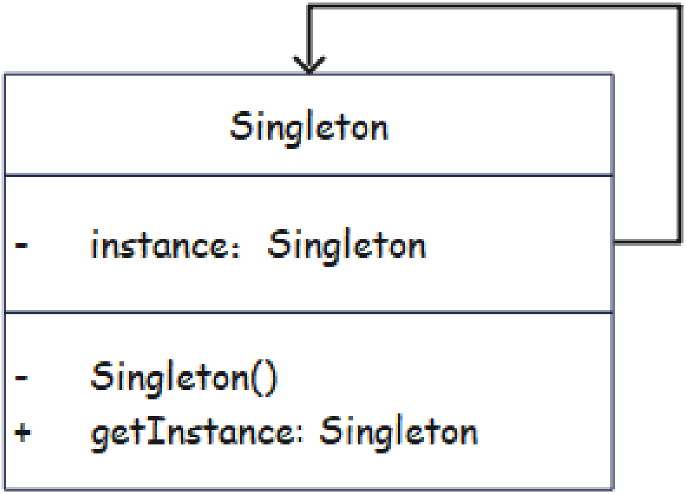
It seems simple to queue the whole class method, but the cost is too high. We just want to instantiate an object! Therefore, it is enough for us to modify an appropriate synchronization lock



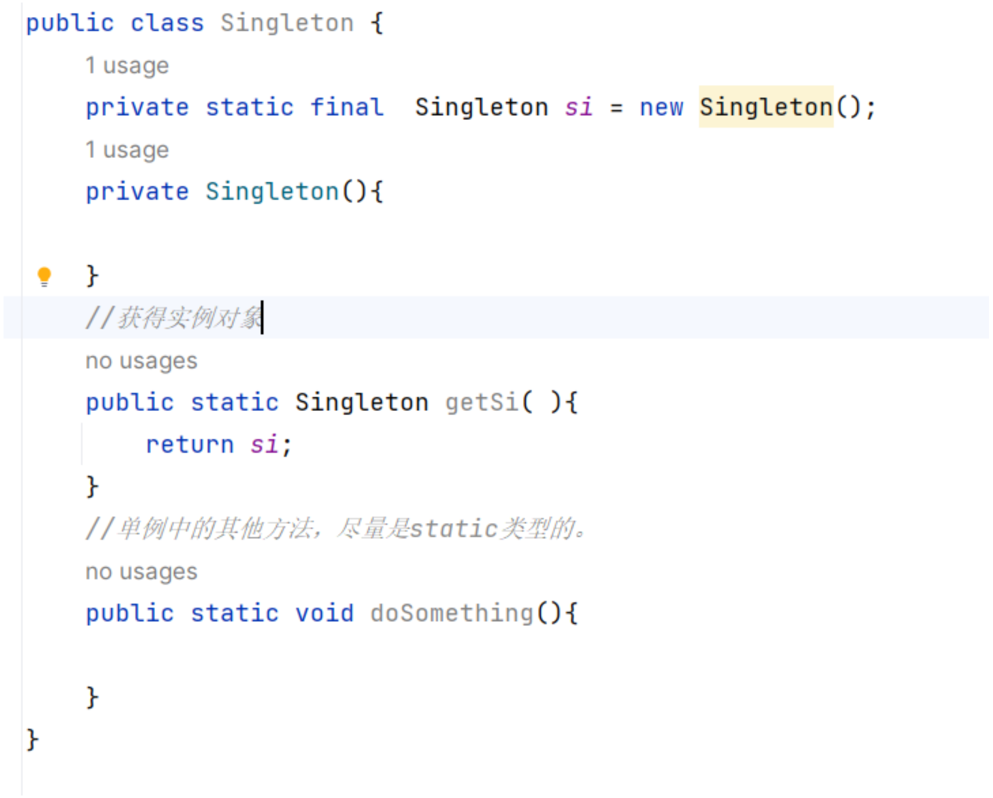
```
public class Sun {
    private static Sun sun;
    private Sun() {

    }
    public static  Sun getInstance() {
        if(sun == null)
            synchronized(Sun.class) {
                if (sun == null) {
                    sun = new Sun();
                }
            }
        return sun;
    }
}
```

Singleton: it contains the properties of its own class instance, hides the constructor with the private keyword, and only provides a get method to obtain the singleton object. (in most cases, we actually use the starving man mode)



Singleton: Common Code



```
public class Singleton {
    1 usage
    private static final  Singleton si = new Singleton();
    1 usage
    private Singleton(){

    }
    //获得实例对象
    no usages
    public static Singleton getSi( ){
        return si;
    }
    //单例中的其他方法，尽量是static类型的。
    no usages
    public static void doSomething(){

    }
}
```

GENERAL PRINCIPLES

Principle 1~19

Principle 1*

Quality is #1

A customer will not tolerate a product with poor quality, regardless of the definition of quality. Quality must be quantified and mechanisms put into place to motivate and reward its achievement. There is no trade-off to be made. The first requirement must be quality.

Principle 2

Quality is in the eyes of the beholder!

There is no one definition of software quality. To developers, it might be an elegant design or elegant code. To users who work in stress environments, it might be response time or high capacity. For cost-sensitive projects, it might be low development cost. For some customers, it might be satisfying all their perceived and not-yet-perceived needs.

Principle 3

Productivity and quality are inseparable

There is a clear relationship between productivity (measured by numbers of widgits—whether they be lines of code or function points—per person-month) and quality.

Principle 4

High-quality software is possible

Although our industry is saturated with examples of software systems that perform poorly; that are full of bugs, or that otherwise fail to satisfy users' needs, there are counter examples. Large-scale software systems can be built with very high quality; but for a steep price tag: on the order of $1000 per line of code.

Principle 5

Don't try to retrofit quality

Quality cannot be retrofit into software. This applies to any definition of quality: maintainability; reliability adaptability; testability, safety and soon. We have a very difficult time building quality into software during development when we try to. How can we possibly expect to achieve quality when we don't try?

Principle 6

Poor reliability is worse than poor efficiency

Poor reliability is not only more difficult to detect, it is also more difficult to fix. A system's poor reliability may not become apparent until years after the system is deployed—and it kills somebody. Once the poor reliability manifests itself, it is often difficult to isolate its cause.

Principle 7*

Give products to customers early

No matter how hard you try to learn users' needs during the requirements phase, the most effective means to ascertain their real needs is to give them a product and let them play with it. If the appropriate features were built into the prototype, the highest-risk user needs will become better known and the product is more likely to be user-satisfactory. Constructing a quick and dirty prototype early in die development process.

Principle 8*

Communicate with customers/users

Never lose sight of why software is being developed to satisfy real needs, to solve real problems. The only way to solve real needs is to communicate with those who have the needs. The customer or user is the most important person involved with your project.

Principle 9

Align incentives for developer and customer

To help align the two organizations' goals: (1) Prioritize requirements (Principle 50) so that developers understand their relative importance, (2) reward the developer based on the relative priorities (for example, all high- priority requirements must be satisfied, each medium priority requirement earns the developer a small additional bonus of some kind, and each low priority requirement satisfied earns a very small bonus), and (3) use strict penalties for late delivery.

Principle 10

Plan to throw one away

The first should at least check out the critical design issues and the operational concept. Furthermore, Royce recommended that such a prerelease version should be developed with approximately 25 percent of the total system development resources.

Principle 11*

Build the right kind of prototype

There are two types of prototypes: throwaway and evolutionary. Throwaway prototypes are built in a quick and dirty manner; are given to the customer for feedback, and are thrown away once the desired information is learned. The desired information is captured in a requirements specification for a full-scale product development. Evolutionary prototypes are built in a quality manner are given to the customer for feedback, and are modified once the desired information is learned to more closely approximate the needs of the users. This process is repeated until the product con- verges to the desired product.

Principle 12

Build the right features into prototype

When constructing a throwaway prototype, build only features that are poorly understood. After all, if you build well understood features, you will learn nothing, and you will have wasted resources.

Principle 13

Build throwaway prototypes quickly

If you've decided to build a throwaway prototype, build it as quickly as you can. Don't worry about quality. Use a one-page requirements specification. Don't worry about design or code documentation. Use any avail- able tool. Use any language that facilitates the quick development of soft- ware applicable to your application. Do not worry about the inherent maintainability of the language.

Principle 14

Grow systems incrementally

One of the most effective techniques to reduce risk in building software is to grow it incrementally. Start small, with a zoorking system that imple- ments only a few functions. Then grow it to cover larger and larger subsets of the eventual functionality. The advantages are (1) lower risk with each build, and (2) seeing a version of the product often helps users envision other functions they would like. The disadvantage is that, if an inappro- priate architecture is selected early; a complete redesign may be necessary to accommodate later changes. Reduce this risk by building throwaway prototypes (Principles 11, 12, and 13) prior to starting the incremental development.

Principle 15*

The more seen, the more needed

It has been witnessed over and over again in the software industry: The more functionality (or performance) that is provided to a user; the more functionality (or performance) that the user will want. Every aspect of both management and engineering processes should be aware that, as soon as the customers see the product, they will want more.This means that every document produced should be stored and organized in a fashion conducive for change.

Principle 16

Change during development is inevitable

To prepare yourself for these changes, be sure that all products of a software development are appropriately cross-referenced to each other (Principles 43, 62, and 107), that change management procedures are in place (Principles 174 and 178 through lfe)z and ttiat budgets and schedules have enough leeway so that you are not tempted to ignore necessary changes just to meet budgets and schedules (Principles 147,148/ and 160).

Principle 17

If possible, buy instead of build

The single most effective technique to reduce escalating software develop- ment costs and risk is to buy software off the shelf instead of building it from scratch. It is true that off-the-shelf software may solve only 75 percent of your problems. But consider the alternative: Pay at least ten times as much, take the risk that the software is 100 percent over budget and late (if finished at all!), and, when it is all done, accept that it still may meet only 75 percent of your expectations.

Principle 18*

Build software so that it need a short users manual

One way to measure the quality of a software system is to look at the size of its users' manual. The shorter the manual, the better the software is. The use of well-designed software should be mostly self-evident. Use standard interfaces. Use industry experts to design self-evident icons, commands, protocols, and user scenarios. Usually customers want simple, clean, clear interfaces—not tricks.

Principle 19

Every complex problem has a solution

Wlad Turski said, 'To every complex problem, there is a simple solution ... and it is wrong!" Be highly suspicious of anybody who offers you something like, "Just follow these 10 simple steps and your software quality problems will disappear.'