



## Chapter 5: Maintainability-Oriented Software Construction Approaches

# 5.1 Metrics and Construction Principles for Maintainability

## 可维护性的度量与构造原则

April 18, 2020

# Outline

- **Software Maintenance and Evolution**
- **Metrics of Maintainability**
- **Modular Design and Modularity Principles**
- **OO Design Principles: SOLID**
- **OO Design Principles: GRASP**
- **Summary**

本章面向另一个质量指标：可维护性——软件发生变化时，是否可以以很小的代价适应变化？

本节是宏观介绍：

- (1) 什么是软件维护；
- (2) 可维护性如何度量；
- (3) 实现高可维护性的设计原则

后面两节介绍具体的面向可维护性的构造技术

# Reading

- 软件工程--实践者的研究方法：第23章
- Object-Oriented Software Construction：第3章



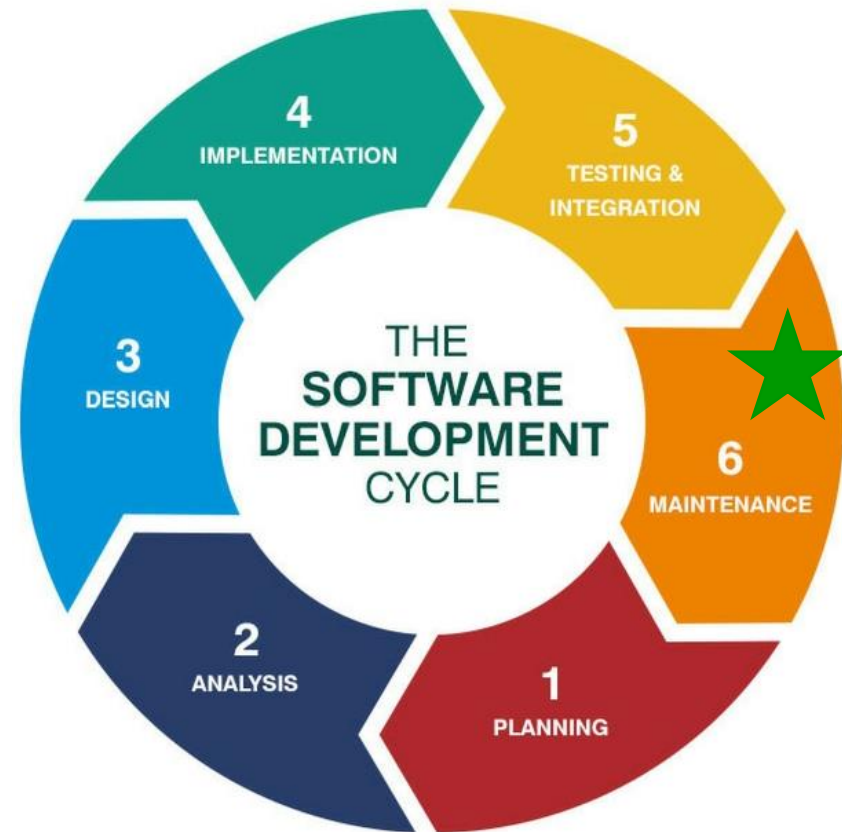


# 1 Software Maintenance and Evolution



# What is **Software Maintenance**?

- **Software maintenance** in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes. 软件维护：修复错误、改善性能
- In "ISO/IEC 14764:2006 Software Engineering – Software Life Cycle Processes – Maintenance"



# Operation & Maintenance Engineer 运维工程师

- Maintenance is one of the most difficult aspects of software production because maintenance incorporates aspects of all other phases 运维是软件开发中最困难的工作之一
- A fault is reported from users and is to be handed by a maintenance engineer. 处理来自用户报告的故障/问题
- A maintenance engineer must have superb debugging skills
  - The fault could lie anywhere within the product, and the original cause of the fault might lie in the by now non-existent specifications or design documents (bug/issue localization).
  - Superb diagnostic skills, testing skills and documentation skills are required (testing, fix, and documenting changes).



# Operation & Maintenance Engineer 运维工程师

## 拿什么拯救你:苦逼的IT运维工程师!

2014-12-26 17:03 it168网站 原创 作者: 皮丽华 编辑: 皮丽华

5条评论

【IT168 评论】目前国内的IT运维还处于救火队员的初级阶段，除了脏活就是累活，天天疲于奔命。什么网络中断、应用卡顿、响应速度慢，服务器宕机啊，各种突发故障都可能让业务成交失败，而查找系统运行的日志又特别费时费力，挖故障如同大海捞针啊，这着实让运维人员急得团团转。作为运维工程师的您，有木有遇到过很苦逼的经历?ChinaUnix社区的网友呼声一片，各自谈起了自己的伤心往事。 (<http://bbs.chinaunix.net/thread-4162292-1-1.html>)





# Operation & Maintenance Engineer 运维工程师

## ■ 阿里运维工程师职责：

- 负责系统稳定性工作；
- 生产系统部署、上线；
- 维护生产系统网络安全、稳定、可靠；
- 维护生产系统数据备份；

## ■ 岗位要求：

- 深入理解运维体系结构，精于容量规划、架构设计、性能优化；
- 熟悉服务管理、单元部署、自动扩容等运维系统建设，对成本控制和效能提升有深刻的理解和实践
- 熟悉故障、监控、限流、降级、预案、扩容工作原理；
- 熟悉java虚拟机，对java应用的部署及系统优化有一定的经验；
- 熟悉自动化发布工具、熟悉docker技术优先；
- ...

### 阿里毕玄：智能时代，运维工程师在谈什么？



阿里云栖社区  
已认证的官方帐号

+ 关注他

14 人赞同了该文章

目前业界真正的智能化运维的落地实践其实并不多，大多还是停留在自动化甚至人工化阶段，然而智能化运维是大势所趋。阿里又是如何应对呢？下面请来自阿里巴巴研发效能团队负责人、阿里研究员毕玄的演讲《智能时代的新运维》。

大多数运维任务并非需要涉及代码修改。

但是有时候不得不回到代码层面...



# After fixing the code

## ■ More Steps:

- Test that the modification works correctly: use specially constructed test cases 测试所做的修改
- Check for regression faults: use stored test data, and add specially test cases to stored test data for future regression testing 回归测试
- Document all changes 记录变化

## ■ How to minimize regression faults 除了修复问题，修改中不能引入新的故障

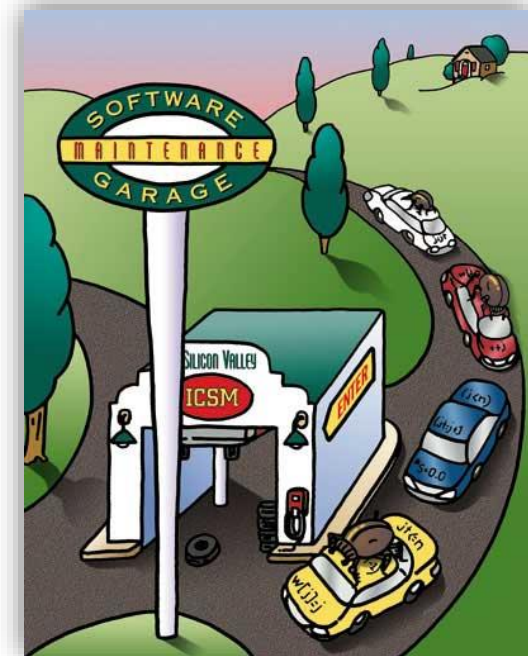
- Consult the detailed documentation and make use of constructed test cases.

## ■ What usually happens: no enough documentation/ test cases 最大的问题：修改后没有足够的文档记录和测试

- The operation engineer has to deduce from the source code itself all the information needed to avoid introducing a regression fault.

# Types of software maintenance

- **Corrective maintenance**      25%      纠错性
  - Reactive modification of a software product performed after delivery to correct discovered problems;
- **Adaptive maintenance**      21%      适应性
  - Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment;
- **Perfective maintenance**      50%      完善性
  - Enhancement of a software product after delivery to improve performance or maintainability;
- **Preventive maintenance**      4%      预防性
  - Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.



# Software Evolution

- Software evolution is a term used in software maintenance, referring to the process of developing software initially, then repeatedly updating it for various reasons. 软件演化：对软件进行持续的更新
- Over 90% of the costs of a typical system arise in the maintenance phase, and that any successful piece of software will inevitably be maintained. 软件的大部分成本来自于维护阶段

# Maintenance is not just the task of op engineers...

- Maintenance is not just the task of maintenance and operation engineers, but also a potential task of software designers and developers. 软件维护不仅仅是运维工程师的工作，而是从设计和开发阶段就开始了
- For them, it is mandatory to consider future potential changes/extensions of the software during the design and construction phases; 在设计与开发阶段就要考虑将来的可维护性
- So that flexible and extensible design/constructions are comprehensively considered, in other words, “easy to change / extension”. 设计方案的 “easy to change”
- This is what’s called “maintainability”, “extensibility” and “flexibility” of software construction.

# Examples of maintainability-oriented construction

- **Modular design and implementation** 模块化
  - Low coupling and high cohesion
- **OO design principles** OO设计原则
  - SOLID、GRASP
- **OO design patterns** OO设计模式
  - Factory method pattern, Builder pattern
  - Bridge pattern, Proxy pattern
  - Memento pattern, State pattern
- **State-based construction (Automata-based programming)** 基于状态的构造技术
- **Table-driven construction** 表驱动的构造技术
- **Grammar-based construction** 基于语法的构造技术

These are what  
to be studied in  
this chapter



## 2 Metrics of Maintainability





# Many names of maintainability

Ready for Change  
Ready for Extension

- **Maintainability** 可维护性 – "The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment".
- **Extensibility** 可扩展性 – Software design/implementation takes future growth into consideration and is seen as a systemic measure of the ability to extend a system and the level of effort required to implement the extension.
- **Flexibility** 灵活性 – The ability of software to change easily in response to user requirements, external technical and social environments, etc.
- **Adaptability** 可适应性 – The ability of an interactive system (adaptive system) that can adapt its behavior to individual users based on information acquired about its user(s) and its environment.

# Many names of maintainability

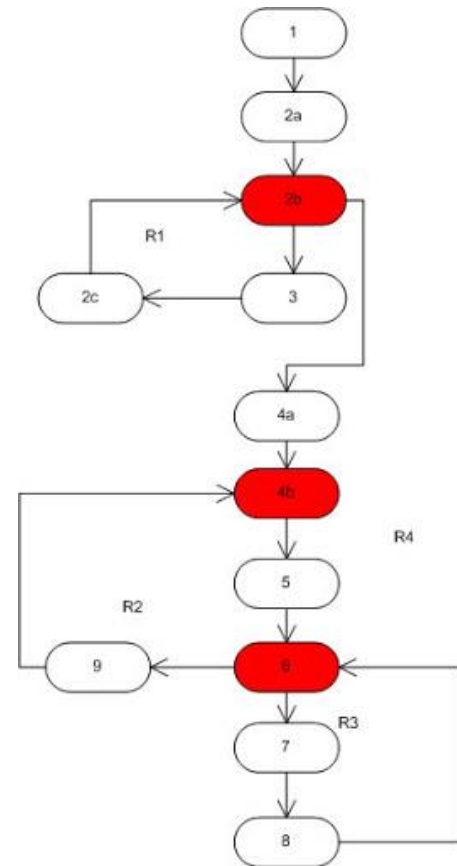
- **Manageability** 可管理性 — How **efficiently and easily** a software system can be monitored and **maintained** to keep the system performing, secure, and running smoothly.
- **Supportability** 支持性 — How **effectively** a software can be **kept** running **after deployment**, based on resources that include quality documentation, diagnostic information, and knowledgeable and available technical staff.

# Questions about maintainability

- **Code review**的时候经常问的关于可维护性的问题:
  - **Structural and design simplicity**: how easy is it to change things? 设计结构是否足够简单?
  - Are things tightly or **loosely coupled** (i.e., separation of concerns)? 模块之间是否松散耦合?
  - Are all elements in a package/module **cohesive** and their responsibilities clear and **closely related**? 模块内部是否高度聚合?
  - Does it have overly **deep inheritance hierarchies** or does it favor **composition over inheritance**? 是否使用了非常深的继承树, 是否使用了 **delegation** 替代继承?
  - How many independent paths of execution are there in the method definitions (i.e., **cyclomatic complexity**)? 代码的圈复杂度是否太高?
  - How much **code duplication** exists? 是否存在重复代码?
  - ...

# Some common-used maintainability metrics

- **Cyclomatic Complexity** 圈复杂度 - Measures the structural complexity of the code.
  - It is created by calculating the number of different code paths in the flow of the program.
  - A program that has complex control flow will require more tests to achieve good code coverage and will be less maintainable.
  - $CC = E - N + 2$ ,  $CC = P + 1$ ,  $CC = \text{number of areas}$
- **Lines of Code** 代码行数 - Indicates the approximate number of lines in the code.
  - A very high count might indicate that a type or method is trying to do too much work and should be split up.
  - It might also indicate that the type or method might be hard to maintain.



# Some common-used maintainability metrics

For a given problem, Let:

- $\eta_1$  = the number of distinct operators
- $\eta_2$  = the number of distinct operands
- $N_1$  = the total number of operators
- $N_2$  = the total number of operands

From these numbers, several measures can be calculated:

- Program vocabulary:  $\eta = \eta_1 + \eta_2$
- Program length:  $N = N_1 + N_2$
- Calculated program length:  $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
- Volume:  $V = N \times \log_2 \eta$
- Difficulty :  $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$
- Effort:  $E = D \times V$

The difficulty measure is related to the difficulty of the program to write or understand

The effort measure translates into actual coding time using the following relation,

- Time required to program:  $T = \frac{E}{18}$  seconds

Halstead's delivered bugs (B) is an estimate for the number of errors in the implementation.

- Number of delivered bugs :  $B = \frac{E^{\frac{2}{3}}}{3000}$  or, more recently,  $B = \frac{V}{3000}$  is accepted

**Halstead Volume:** a composite metric based on the number of (distinct) operators and operands in source code.

## Some common-used maintainability metrics

- **Maintainability Index (MI)** 可维护性指数 - Calculates an index value between 0 and 100 that represents the relative ease of maintaining the code.
- **A high value means better maintainability. It is calculated based on:**
  - Halstead Volume (HV)
  - Cyclomatic Complexity (CC)
  - The average number of lines of code per module (LOC)
  - The percentage of comment lines per module (COM).

$$171 - 5.2 \ln(HV) - 0.23CC - 16.2 \ln(LOC) + 50.0 \sin \sqrt{2.46 * COM}$$



# Some common-used maintainability metrics

- **Depth of Inheritance** 继承的层次数 - Indicates the number of class definitions that extend to the root of the class hierarchy. The deeper the hierarchy the more difficult it might be to understand where particular methods and fields are defined or/and redefined.
- **Class Coupling** 类之间的耦合度 - Measures the coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration.
  - Good software design dictates that types and methods should have high cohesion and low coupling.
  - High coupling indicates a design that is difficult to reuse and maintain because of its many interdependencies on other types.
- **Unit test coverage** 单元测试的覆盖度 - indicates what part of the code base is covered by automated unit tests.



# 3 Modular Design and Modularity Principles



# Modular programming 模块化编程

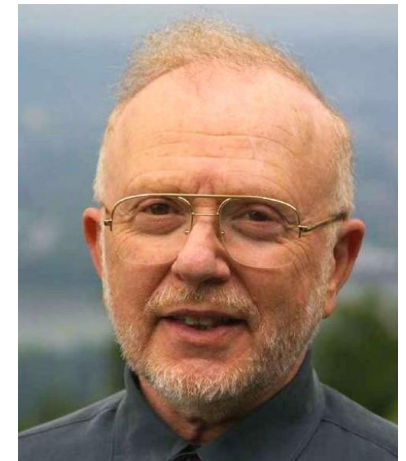
- **Modular programming** is a design technique that emphasizes separating the functionality of a program into **independent, interchangeable modules**, such that each contains everything necessary to execute only one aspect of the desired functionality.
- **High-level decomposition of the code of an entire program into pieces in both Structured Programming and OOP.**



Niklaus Wirth (1934-)  
Turing Award 1984



Edsger Dijkstra (1930-2002)  
Turing Award 1972



David L. Parnas (1941- )

# Modular programming

- The goal of design is to partition the system into modules and assign responsibility among the components in a way that:
  - High cohesion within modules 高内聚
  - Loose coupling between modules 低耦合
- Modularity reduces the total complexity a programmer has to deal with at any one time assuming:
  - Functions are assigned to modules in away that groups similar functions together (Separation of concerns) 分离关注点
  - There are small, simple, well-defined interfaces between modules (Information hiding) 信息隐藏
- The principles of cohesion and coupling are probably the most important design principles for evaluating the maintainability of a design.



# (1) Five Criteria for Evaluating Modularity



# Five Criteria for Evaluating Modularity

- **Decomposability (可分解性)**
  - Are larger components decomposed into smaller components?
- **Composability (可组合性)**
  - Are larger components composed from smaller components?
- **Understandability (可理解性)**
  - Are components separately understandable?
- **Continuity (可持续性) — — 发生变化时受影响范围最小**
  - Do small changes to the specification affect a localized and limited number of components?
- **Protection (出现异常之后的保护) — — 出现异常后受影响范围最小**
  - Are the effects of run-time abnormalities confined to a small number of related components?





## (2) Five Rules of Modularity Design



# Five Rules of Modularity Design

- 
- Direct Mapping (**直接映射**)
  - Few Interfaces (**尽可能少的接口**)
  - Small Interfaces (**尽可能小的接口**)
  - Explicit Interfaces (**显式接口**)
  - Information Hiding (**信息隐藏**)

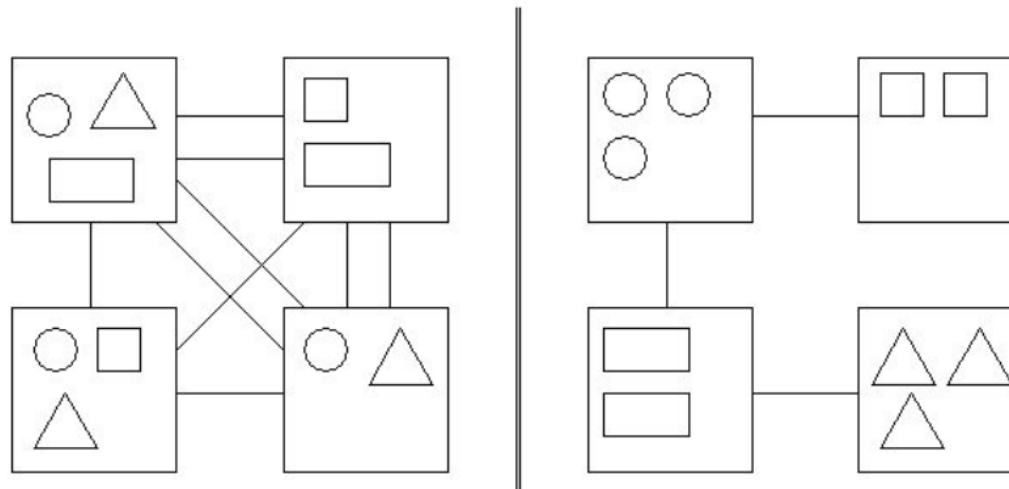


## (3) Coupling and Cohesion



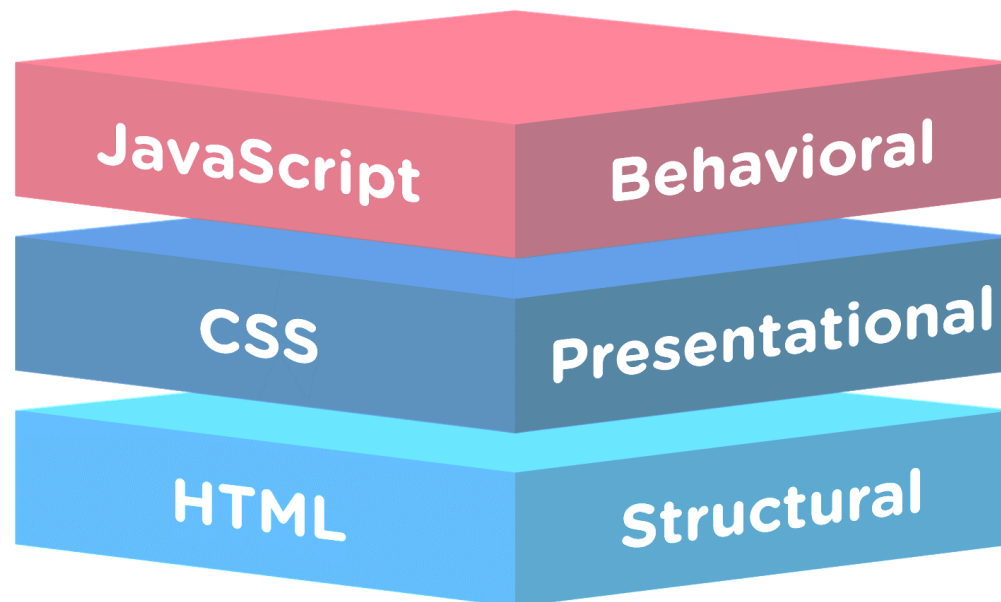
# Coupling

- **Coupling is the measure of dependency between modules.** A dependency exists between two modules if a change in one could require a change in the other.
- **The degree of coupling between modules is determined by:**
  - The number of interfaces between modules (quantity), and
  - Complexity of each interface (determined by the type of communication) (quality)



# Coupling between HTML, CSS and JavaScript

- **A well-designed web app modularizes around:**
  - HTML files which specify data and semantics
  - CSS rules which specify the look and formatting of HTML data
  - JavaScript which defines behavior/interactivity of page



# Coupling between HTML, CSS and JavaScript

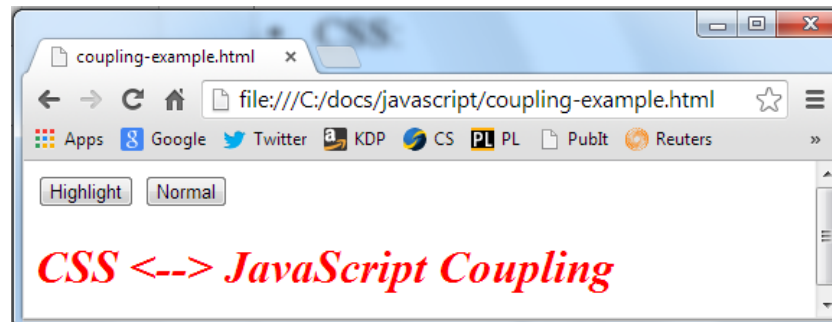
## ■ HTML:

```
<!doctype html>
<html>
<head>
  <script type="text/javascript" src="base.js"></script>
  <link rel="stylesheet" href="default.css">
</head>
<body>
  <button onclick="highlight2()">Highlight</button>
  <button onclick="normal2()">Normal</button>
  <h1 id="title" class="NormalClass">CSS <--> JavaScript Coupling</h1>
</body>
</html>
```

## ■ CSS:

```
.NormalClass {
  color:inherit;
  font-style:normal;
}
```

## ■ Output:





# Coupling between HTML, CSS and JavaScript

- JavaScript code modifies the style attribute of HTML element.

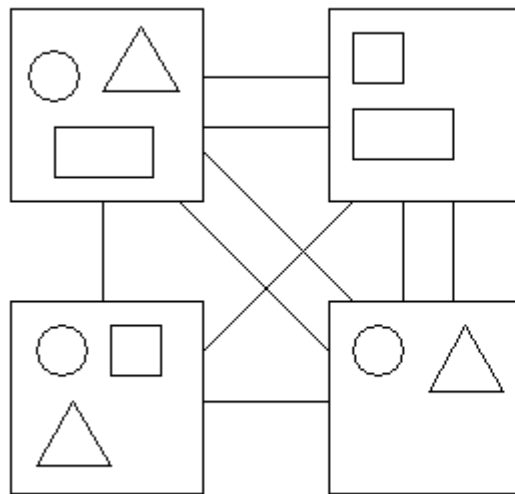
```
function highlight() {  
    document.getElementById("title").style.color="red";  
    document.getElementById("title").style.fontStyle="italic";  
}  
  
function normal() {  
    document.getElementById("title").style.color="inherit";  
    document.getElementById("title").style.fontStyle="normal";  
}
```

- Or, JavaScript code modifies the class attribute of HTML element.

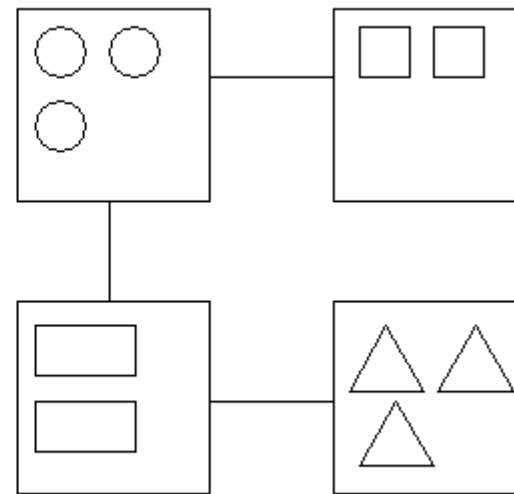
```
function highlight() {  
    document.getElementById("title").className = "HighlightClass";  
}  
  
function normal() {  
    document.getElementById("title").className = "NormalClass";  
}
```

# Cohesion

- **Cohesion is a measure of how strongly related the functions or responsibilities of a module are.**
- **A module has high cohesion if all of its elements are working towards the same goal.**



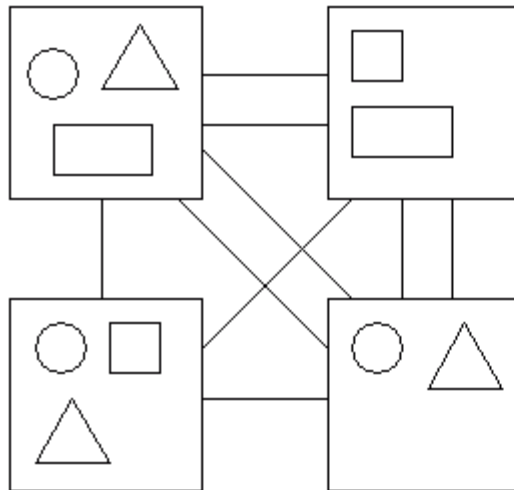
Low cohesion and high coupling



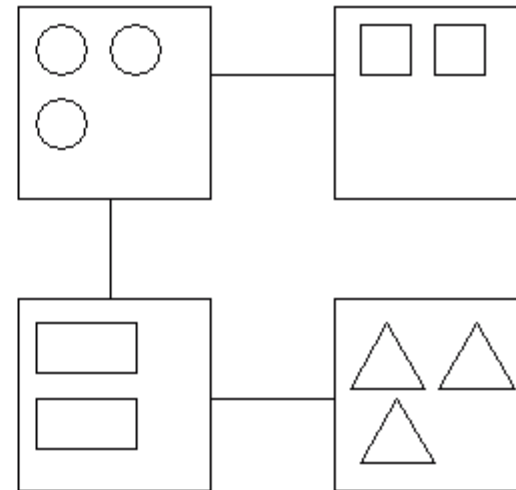
High cohesion and low coupling

# Cohesion and Coupling

- The best designs have high cohesion (also called **strong cohesion**) within a module and low coupling (also called **weak coupling**) between modules.

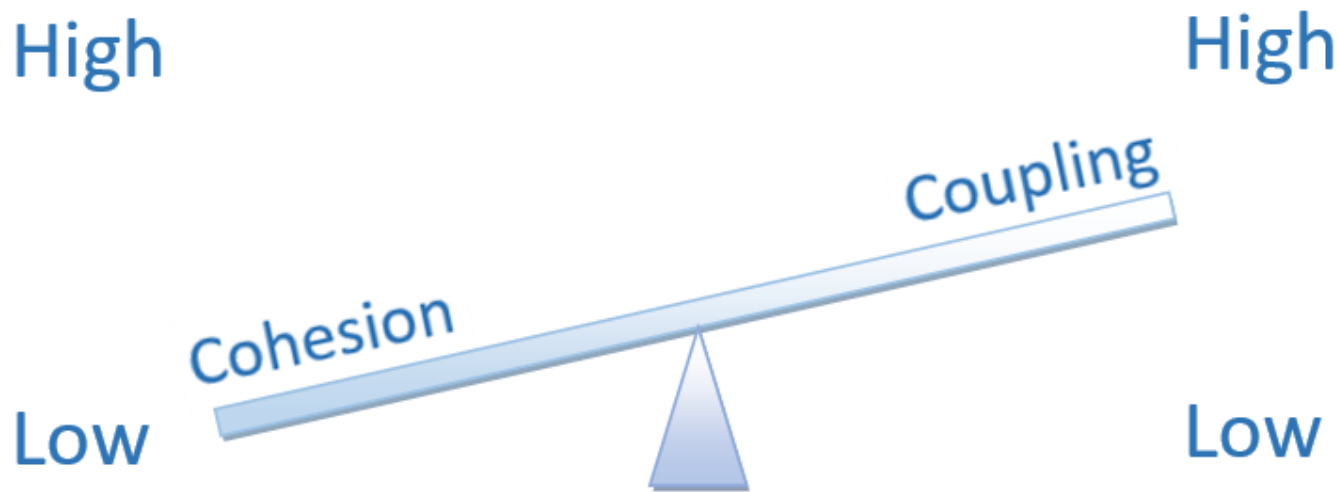


Low cohesion and high coupling



High cohesion and low coupling

# Coupling and Cohesion are with trade-off



- When Coupling is high, cohesion tends to be low and vice versa.



# 4 OO Design Principles: SOLID



# SOLID: 5 classes design principles

- (SRP) The Single Responsibility Principle **单一责任原则**
- (OCP) The Open-Closed Principle **开放-封闭原则**
- (LSP) The Liskov Substitution Principle **Liskov替换原则**
- (DIP) The Dependency Inversion Principle **依赖转置原则**
- (ISP) The Interface Segregation Principle **接口聚合原则**



# (1) Single Responsibility Principle (SRP)

单一责任原则

# Single Responsibility Principle

- “There should never be more than one reason for a class to change”, i.e., a class should concentrate on doing one thing and one thing only. 不应该有多于1个原因让你的ADT发生变化，否则就拆分开



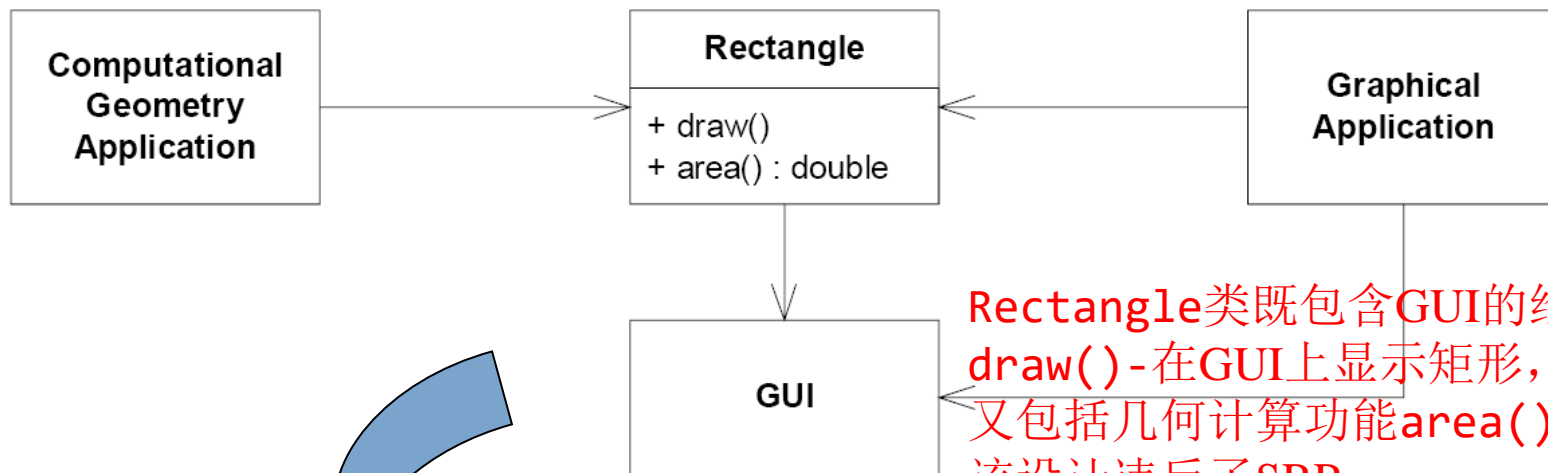
Just because you can, doesn't mean you should



# (SRP) The Single Responsibility Principle

- Responsibility: “a reason for change.” (责任: 变化的原因)
- SRP:
  - There should never be more than one reason for a class to change. (不应有多于1个的原因使得一个类发生变化)
  - One class, one responsibility. (一个类, 一个责任)
- 如果一个类包含了多个责任, 那么将引起不良后果:
  - 引入额外的包, 占据资源
  - 导致频繁的重新配置、部署等
- The SRP is one of the simplest of the principle, and one of the hardest to get right. (最简单的原则, 却是最难做好的原则)

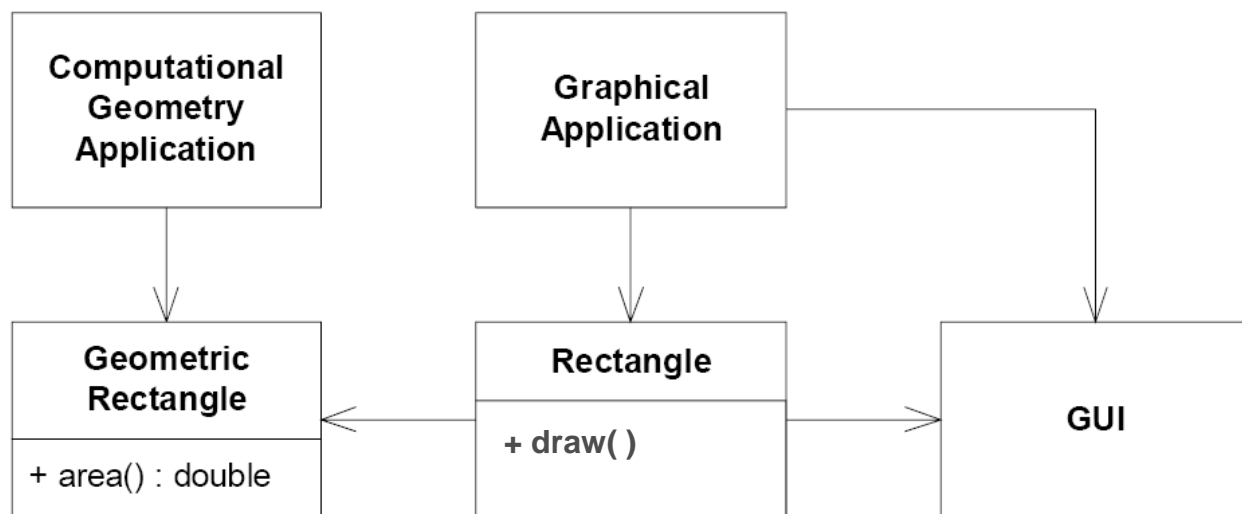
# SRP的一个反例



**Rectangle**类既包含GUI的绘图功能 **draw()** - 在GUI上显示矩形，又包括几何计算功能 **area()** - 计算面积。该设计违反了SRP。

通过分解，将两个无关的责任分离开来，分别放置在两个类中：

- **Geometric Rectangle** 类负责计算面积，
- **Rectangle** 类负责在GUI上绘图。



# Single Responsibility Principle

## ■ Two responsibilities

- Connection Management
- Data Communication

```
interface Modem {  
    public void dial(String pno);  
    public void hangup();  
  
    public void send(char c);  
    public char recv();  
}
```

```
interface DataChannel {  
    public void send(char c);  
    public char recv();  
}
```

```
interface Connection {  
    public void dial(String phn);  
    public char hangup();  
}
```



## (2) Open/Closed Principle (OCP)

(面向变化的) 开放/封闭原则

# (OCP) The Open-Closed Principle

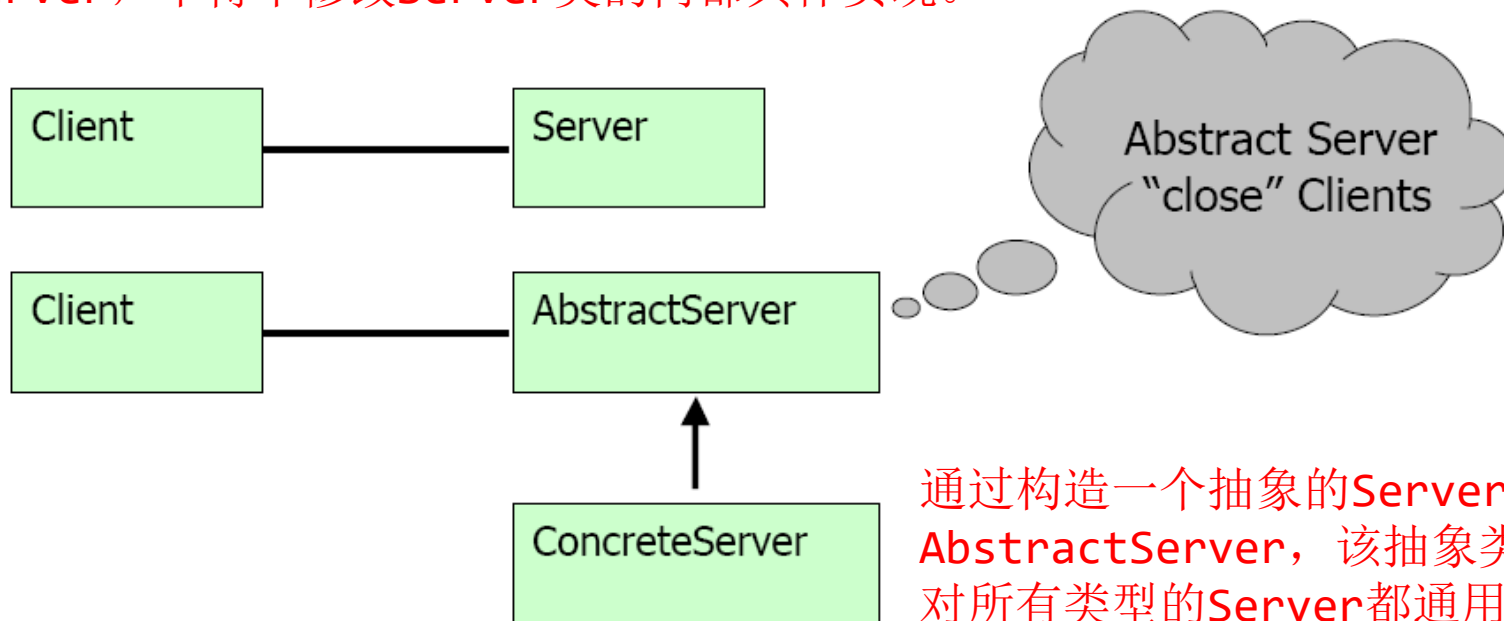
- **Classes should be open for extension (对扩展性的开放)**
  - This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. (模块的行为应是可扩展的, 从而该模块可表现出新的行为以满足需求的变化)
  
- **But closed for modification. (对修改的封闭)**
  - The source code of such a module is inviolate. No one is allowed to make source code changes to it. (但模块自身的代码是不应被修改的)
  - The normal way to extend the behavior of a module is to make changes to that module. (扩展模块行为的一般途径是修改模块的内部实现)
  - A module that cannot be changed is normally thought to have a fixed behavior. (如果一个模块不能被修改, 那么它通常被认为是具有固定的行为)

# Open Closed Principle

- Key: abstraction (**关键的解决方案：抽象技术**)
- "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification", i.e., change a class' behavior using **inheritance** and **composition/delegation**

# OCP的一个反例

如果有多多种类型的**Server**，那么针对每一种新出现的**Server**，不得不修改**Server**类的内部具体实现。



通过构造一个抽象的**Server**类：**AbstractServer**，该抽象类中包含针对所有类型的**Server**都通用的代码，从而实现了修改的封闭；当出现新的**Server**类型时，只需从该抽象类中派生出具体的子类**ConcreteServer**即可，从而支持了对扩展的开放。

# Open Closed Principle

```
// Open-Close Principle - Bad example
class GraphicEditor {

    public void drawShape(Shape s) {
        if (s.m_type==1)
            drawRectangle(s);
        else if (s.m_type==2)
            drawCircle(s);
    }
    public void drawCircle(Circle r)
        {...}
    public void drawRectangle(Rectangle r)
        {...}
}
```

一大堆复杂的  
if-else  
/switch-case  
结构，维护起来  
非常麻烦

```
class Shape {
    int m_type;
}

class Rectangle extends Shape {
    Rectangle() {
        super.m_type=1;
    }
}

class Circle extends Shape {
    Circle() {
        super.m_type=2;
    }
}
```



# Open Closed Principle - Improved

// Open-Close Principle - Good example

```
class GraphicEditor {  
    public void drawShape(Shape s) {  
        s.draw();  
    }  
}
```

```
class Shape {  
    abstract void draw();  
}
```

```
class Rectangle extends Shape {  
    public void draw() {  
        // draw the rectangle  
    }  
}
```



# (3) Liskov Substitution Principle (LSP)



Liskov替换原则

# (LSP) The Liskov Substitution Principle

- "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it", i.e., subclasses should behave nicely when used in place of their base class
- LSP: Subtypes must be substitutable for their base types. (子类型必须能够替换其基类型)
- Derived Classes must be usable through the base class interface without the need for the client to know the difference. (派生类必须能够通过其基类的接口使用，客户端无需了解二者之间的差异)

→ Already discussed in Section 4-2 Reusability



## (4) Interface Segregation Principle (ISP)

接口隔离原则

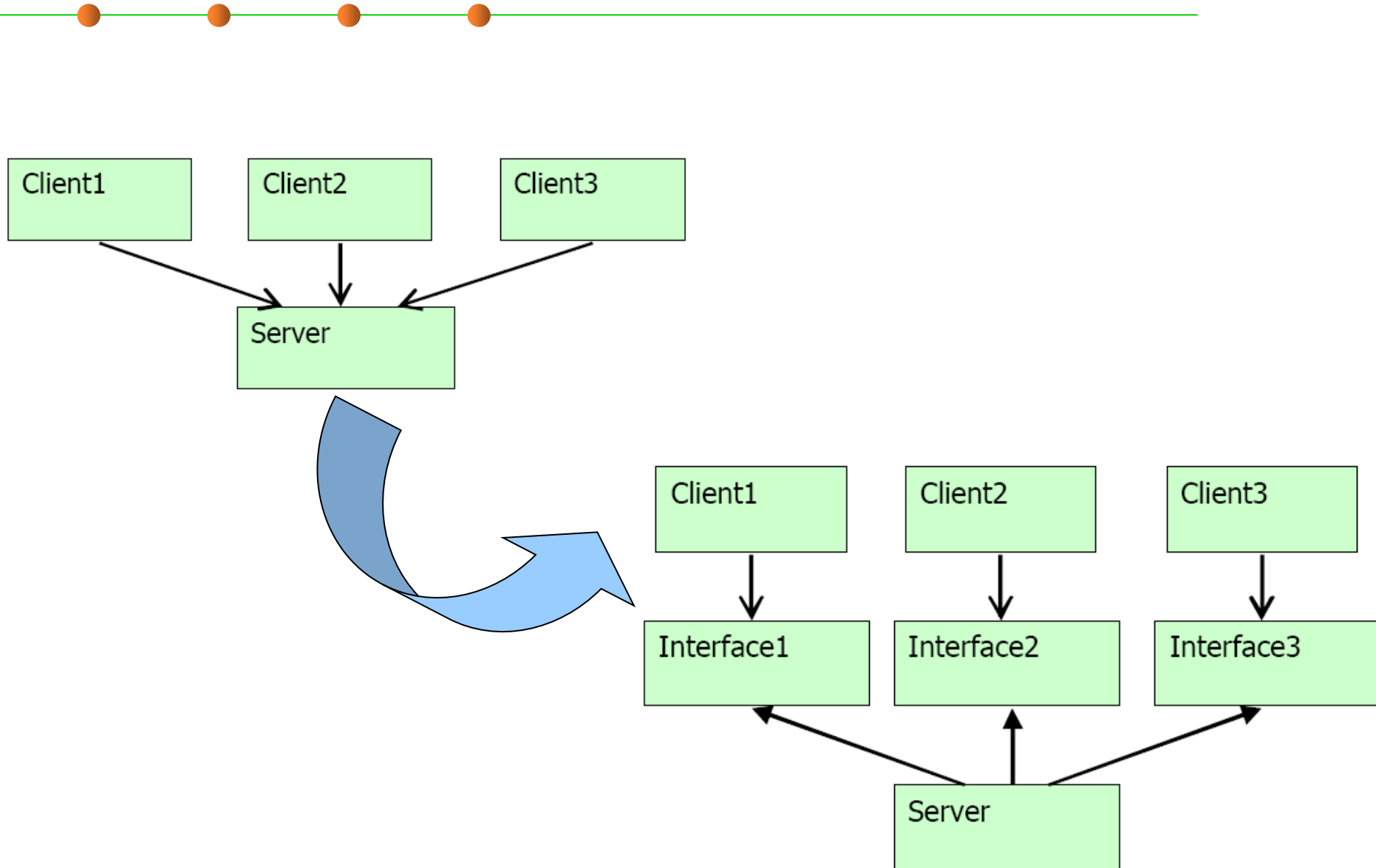
# Interface Segregation Principle

- “Clients should not be forced to depend upon interfaces that they do not use”, i.e., keep interfaces small. 不能强迫客户端依赖于它们不需要的接口：只提供必需的接口
- Don't force classes to implement methods they can't (Swing/Java)
- Don't **pollute** interfaces with a lot of methods
- Avoid **'fat'** interfaces

# (ISP) The Interface Segregation

- Clients should not be forced to depend upon methods that they do not use. (客户端不应依赖于它们不需要的方法)
- Interfaces belong to clients, not to hierarchies.
- This principle deals with the disadvantages of “fat” interfaces. (“胖”接口具有很多缺点)
- Classes that have “fat” interfaces are classes whose interfaces are not cohesive. (不够聚合)
  - The interfaces of the class can be broken up into groups of member functions. (胖接口可分解为多个小的接口)
  - Each group serves a different set of clients (不同的接口向不同的客户端提供服务).
  - Thus some clients use one group of member functions, and other clients use the other groups. (客户端只访问自己所需要的端口)

# (ISP) The Interface Segregation Principle



# Interface Segregation Principle

//bad example (polluted interface)

```
interface Worker {
```

```
    void work();
```

```
    void eat();
```

```
}
```

```
ManWorker implements Worker {
```

```
    void work() {...};
```

```
    void eat() {...};
```

```
}
```

```
RobotWorker implements Worker {
```

```
    void work() {...};
```

```
    void eat() {//Not Applicable  
                for a RobotWorker};
```

```
}
```

Solution: split into two

```
interface Workable {  
    public void work();
```

```
}
```

```
interface Feedable{  
    public void eat();
```

```
}
```



# Interface Segregation Principle

```
interface Workable {  
    public void work();  
}
```

```
interface Feedable{  
    public void eat();  
}
```

```
ManWorker implements Workable, Feedable {  
    void work() {...};  
    void eat() {...};  
}
```

```
RobotWorker implements Workable {  
    void work() {...};  
}
```



## (5) Dependency Inversion Principle (DIP)

依赖转置原则

# (DIP) The Dependency Inversion Principle

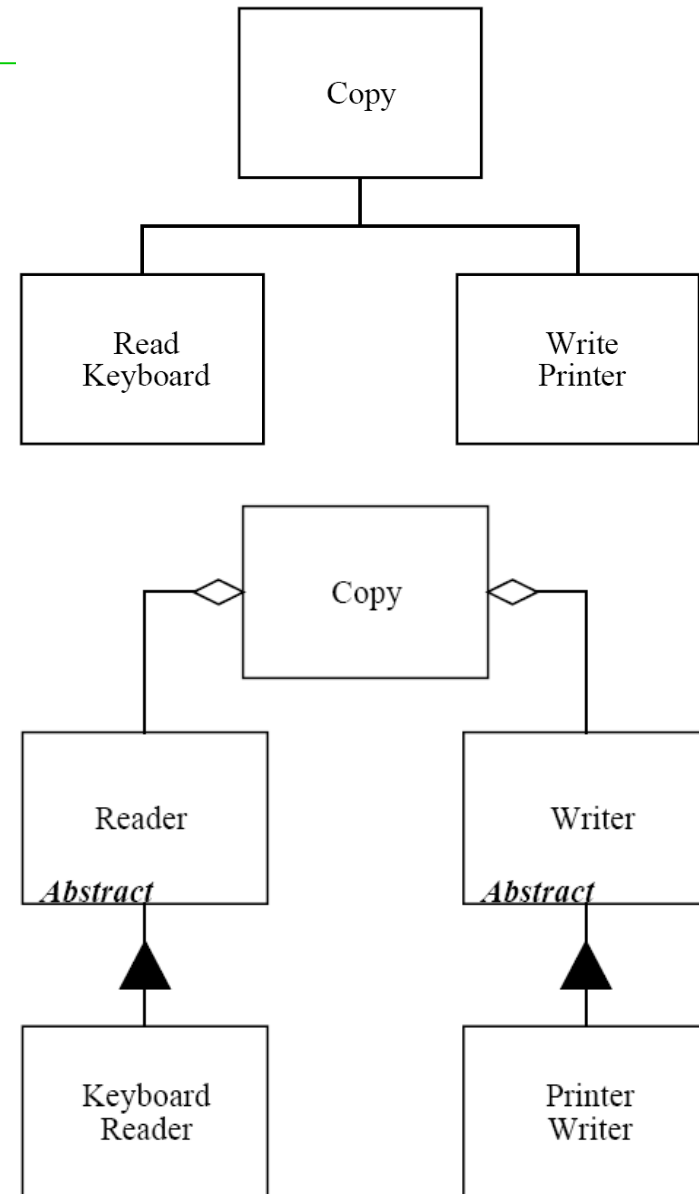
- **High level modules should not depend upon low level modules. Both should depend upon abstractions.**
  - Abstractions should not depend upon details (抽象的模块不应依赖于具体的模块)
  - Details should depend upon abstractions (具体应依赖于抽象)
- **Lots of interfaces and abstractions should be used!**

# Example: the “Copy” program

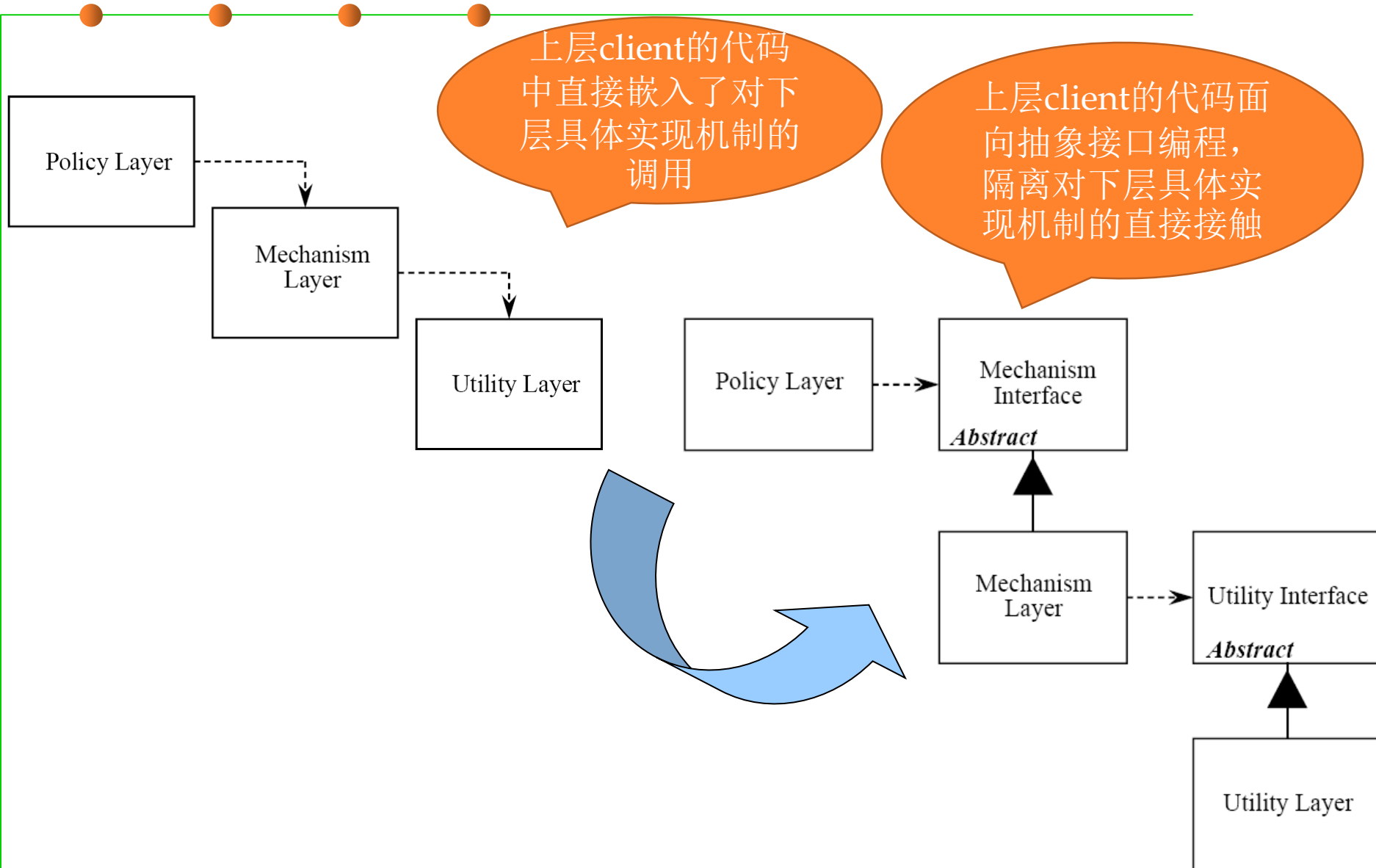
```
void Copy(OutputStream dev) {
    int c;
    while ((c = ReadKeyboard()) != EOF)
        if (dev == printer)
            writeToPrinter(c);
        else
            writeToDisk(c);
}
```

---

```
interface Reader {
    public int read();
}
interface Writer {
    public int write(c);
}
class Copy {
    void Copy(Reader r, Writer w) {
        int c;
        while (c=r.read() != EOF)
            w.write(c);
    }
}
```



# DIP: another example



# Dependency Inversion Principle

//DIP - bad example

```
public class EmployeeService {  
    private EmployeeFinder emFinder; //concrete class, not abstract.  
    //Can access a SQL DB for instance  
    public Employee findEmployee(...) {  
        emFinder.findEmployee(...)  
    }  
}
```

//DIP - fixed

```
public class EmployeeService {  
    private IEmployeeFinder emFinder  
    //depends on an abstraction, not an implementation  
  
    public Employee findEmployee(...) {  
        emFinder.findEmployee(...)  
    }  
}
```

换句话说：  
delegation的时候，  
要通过interface建立  
联系，而非具体子类

- Now its possible to change the finder to be a XmEmployeeFinder, DBEmployeeFinder, FlatFileEmployeeFinder, MockEmployeeFinder....



# 5 OO Design Principles: GRASP



# What's GRASP patterns

- General Responsibility Assignment Software Patterns (principles), abbreviated **GRASP**, consist of guidelines for **assigning responsibility to classes and objects in OOP**.
- The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way.
- This approach to understanding and using design principles is based on patterns of **assigning responsibilities to classes**.

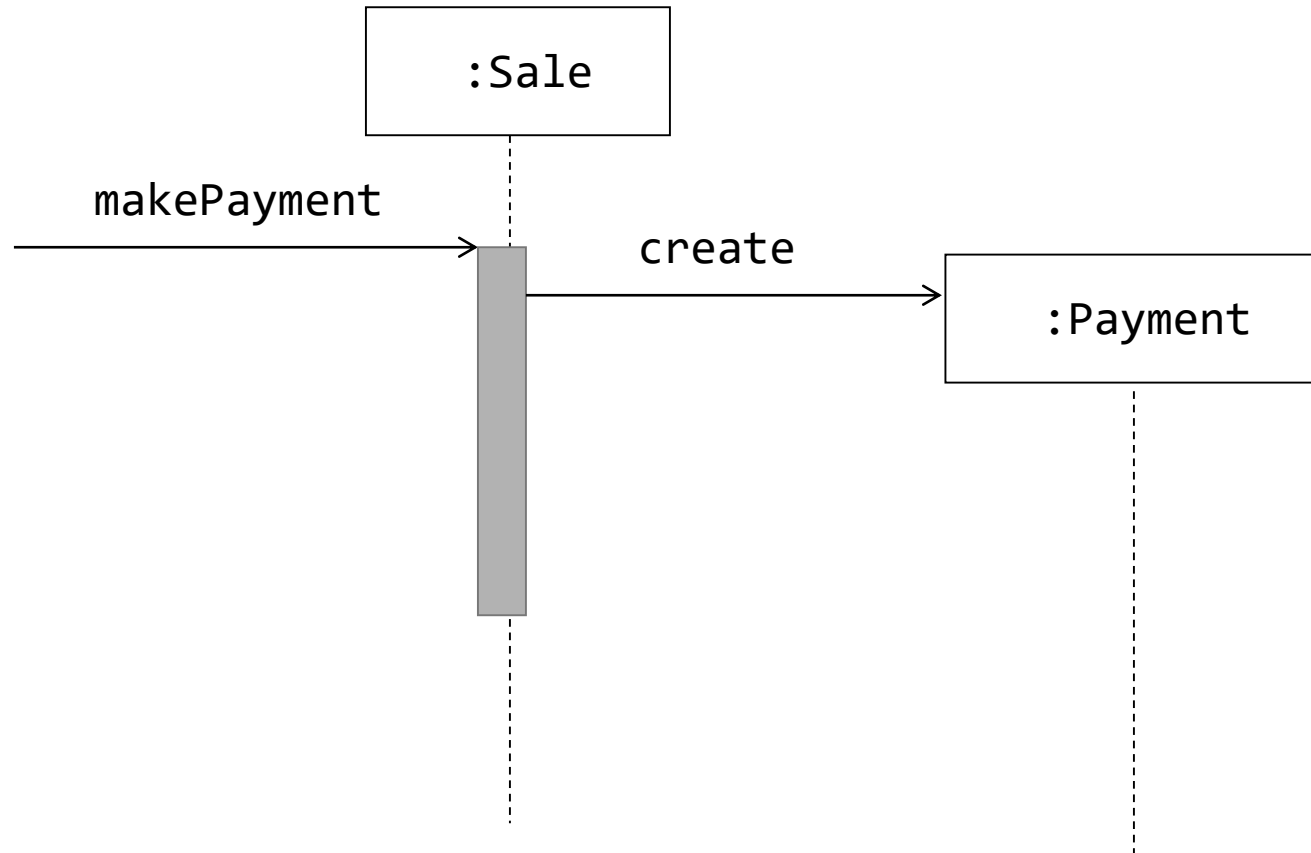
**GRASP是关于如何为“类”和“对象”指派“职责”的一系列原则**



# What's responsibility


- **Responsibility of an object:** related to the **obligations** of an object
- **Knowing:**
  - Knowing about private encapsulated data
  - Knowing about related objects
  - Knowing about things it can derive or calculate
- **Doing:**
  - Doing something itself, such as creating an object or doing a calculation
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects.

# Responsibilities and methods



Responsibilities are implemented using methods:  
`makePayment` implies `Sale` object has a responsibility to create  
a `Payment` object

# What is GRASP composed of?

- 
- **Controller**
  - **Information expert**
  - **Creator**
  - **Low coupling**
  - **High cohesion**
  - **Indirection**
  - **Polymorphism**
  - **Protected variations**
  - **Pure fabrication**


Learn GRASP by yourself  
from  
[CMU course](#)  
and  
[Wikipedia](#)



# Summary



# Summary

- 
- **Software Maintenance and Evolution**
  - **Metrics of Maintainability**
  - **Modular Design and Modularity Principles**
  - **OO Design Principles: SOLID**
  - **OO Design Principles: GRASP**



The end

April 18, 2020