Some common-used maintainability metrics

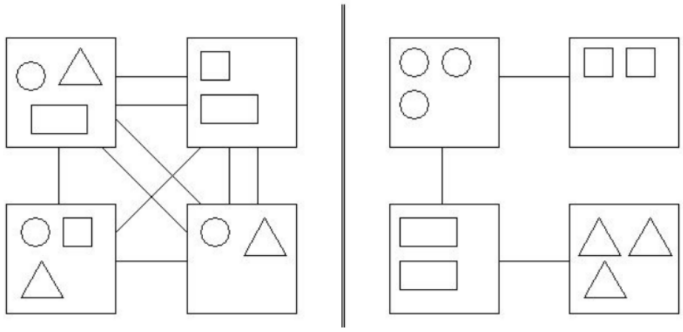
1. Cyclomatic Complexity 圈复杂度
2. Lines of Code 代码行数
3. Maintainability Index (MI) 可维护性指数。A high value means better maintainability
4. Depth of Inheritance 继承的层次数
5. Class Coupling 类之间的耦合度
6. Unit test coverage单元测试的覆盖度

模块化编程

coupling

Coupling

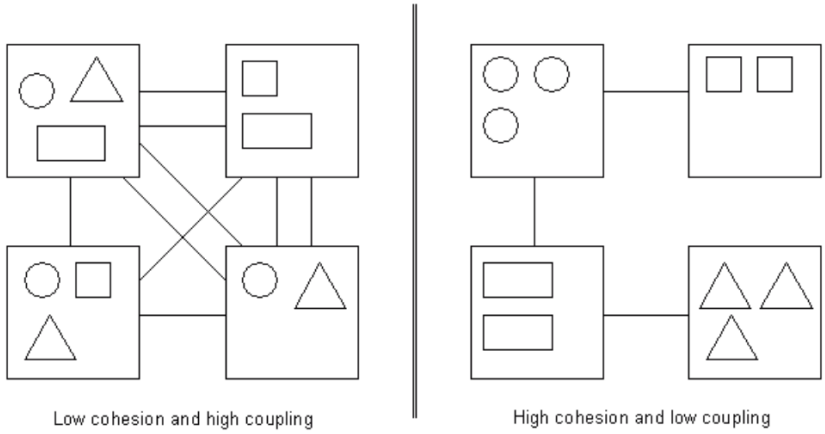
- **Coupling is the measure of dependency between modules. A dependency exists between two modules if a change in one could require a change in the other.**
- **The degree of coupling between modules is determined by:**
  - The number of interfaces between modules (quantity), and
  - Complexity of each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



cohesion

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.**



OO Design Principles: SOLID

SRP

(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)**

例子：

Single Responsibility Principle

- **Two resposibilities**
  - 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();
}
```

总结： 一个人只负责一件事

OCP

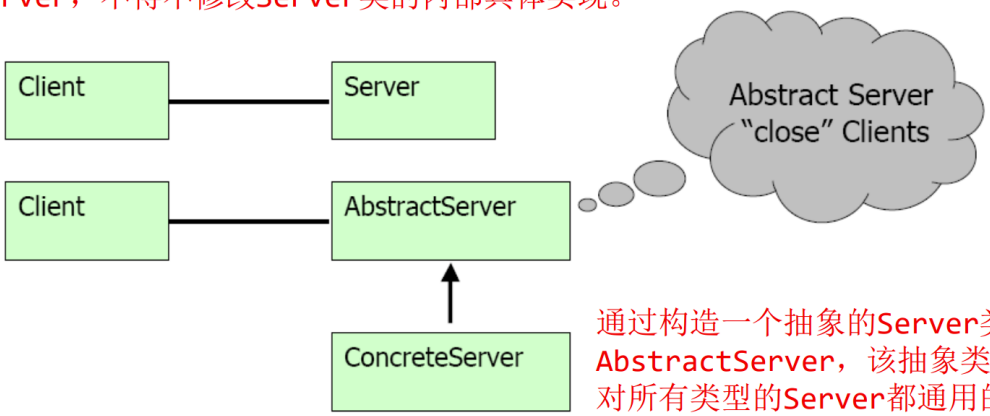
(OCP) The Open-Closed Principle

- **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)

例子：

如果有多种类型的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，从而实现了对修改的封闭；
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。

总结： 利用继承进行模块的扩展

LSP

见reuse

ISP

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 so implement methods they can't (Swing/Java)**
- **Don't pollute interfaces with a lot of methods**
- **Avoid 'fat' interfaces**

例子：

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 Appliciable
                for a RobotWorker};
}
```

Solution: split into two

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

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

总结： 感觉像接口的SRP，接口不应过大，应该将fat接口按照功能分成几个小的接口

DIP

(DIP) The Dependency Inversion Principle

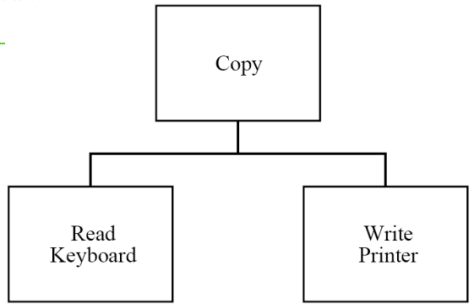
- **High level modules should not depend upon low level modules. Both should depend upon abstractions.**
  - Abstractions should not depend upon details (抽象的模块不应依赖于具体的模块)
  - Details should depend upon abstractions (具体应依赖于抽象)

- **Lots of interfaces and abstractions should be used!**

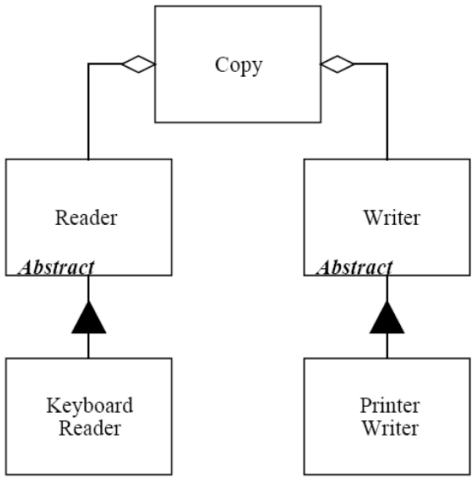
例子：

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);
  }
}
```



总结：感觉还是要更加抽象，分工明确，在设置参数的时候尽量用interface等高抽象的来做编译类型，这样的话自由度更大，让用户有更多的选择权力

正则表达式

正则语法：简化之后 可以表达为一个产生式而不包含任何非终止节点

https://www.runoob.com/java/java-regular-expressions.html