【软件构造】第六章第一节 可维护性的度量与构造原则

第六章第一节 可维护性的度量与构造原则

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——很抽象。

Outline

- 软件的维护和演化
- 可维护性的常见度量指标
- 聚合度与耦合度
- 面向对象五大原则SOLID
  - 单一职责原则SRP(Single Responsibility Principle)
  - 开放封闭原则OCP(Open - Close Principle)
  - 里式替换原则LSP(the Liskov Substitution Principle LSP)
  - 依赖倒置原则DIP(the Dependency Inversion Principle DIP)
  - 接口分离原则ISP(the Interface Segregation Principle ISP)

Notes

软件的维护和演化

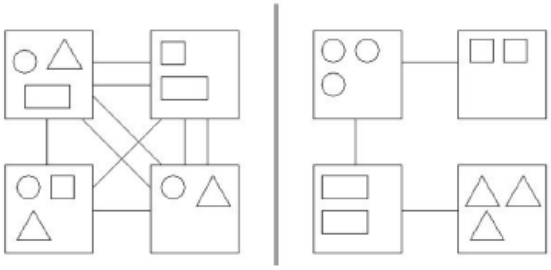
- 定义：软件可维护性是指软件产品被修改的能力，修改包括纠正、改进或软件对环境、需求和功能规格说明变化的适应。简而言之，软件维护：修复错误、改善性能。
- 类型：纠错性（25%）、适应性（25%）、完善性（50%）、预防性（4%）
- 演化：软件演化是一个程序不断调节以满足新的软件需求过程。
- 演化的规律：软件质量下降，延续软件生命
- 软件维护和演化的目标：提高软件的适应性，延续软件生命 。
- 意义：软件维护不仅仅是运维工程师的工作，而是从设计和开发阶段就开始了 。在设计与开发阶段就要考虑将来的可维护性 ， 设计方案需要“easy to change”
- 基于可维护性建设的例子：
  - 模块化
  - OO设计原则
  - OO设计模式
  - 基于状态的构造技术
  - 表驱动的构造技术
  - 基于语法的构造技术

可维护性的常见度量指标

- 可维护性：可轻松修改软件系统或组件，以纠正故障，提高性能或其他属性，或适应变化的环境。
- 除此之外，可维护性还有其他许多别名：可扩展性（Extensibility）、灵活性（Flexibility）、可适应性（Adaptability）、可管理性（Manageability）、支持性（Supportability）。总之，有好的可维护性就意味着容易改变，容易扩展。
- 软件可维护性的五个子特性：
  - 易分析性。软件产品诊断软件中的缺陷或失效原因或识别待修改部分的能力。
  - 易改变性。软件产品使指定的修改可以被实现的能力，实现包括编码、设计和文档的更改。如果软件由最终用户修改，那么易改变性可能会影响易操作性。
  - 稳定性。软件产品避免由于软件修改而造成意外结果的能力。
  - 易测试性。软件产品使已修改软件能被确认的能力。
  - 维护性的依从性。软件产品遵循与维护性相关的标准或约定的能力。
- 一些常用的可维护性度量标准：
  - 圈复杂度（CyclomaticComplexity）：度量代码的结构复杂度。
  - 代码行数（Lines of Code）：指示代码中的大致行数。
  - Halstead Volume：基于源代码中（不同）运算符和操作数的数量的合成度量。
  - 可维护性指数（MI）：计算介于0和100之间的索引值，表示维护代码的相对容易性。 高价值意味着更好的可维护性。
  - 继承的层次数：表示扩展到类层次结构的根的类定义的数量。 等级越深，就越难理解特定方法和字段在何处被定义或重新定义。
  - 类之间的耦合度：通过参数，局部变量，返回类型，方法调用，泛型或模板实例化，基类，接口实现，在外部类型上定义的字段和属性修饰来测量耦合到唯一类。
  - 单元测试覆盖率：指示代码库的哪些部分被自动化单元测试覆盖。

模块化设计规范：聚合度与耦合度

- 模块化编程的含义：模块化编程是一种设计技术，它强调将程序的功能分解为独立的可互换模块，以便每个模块都包含执行所需功能的一个方面。
- 设计规范：高内聚低耦合
- 评估模块化的五个标准：
  - 可分解性：将问题分解为各个可独立解决的子问题
  - 可组合性：可容易的将模块组合起来形成新的系统
  - 可理解性：每个子模块都可被系统设计者容易的理解
  - 可持续性：小的变化将只影响一小部分模块，而不会影响整个体系结构
  - 出现异常之后的保护：运行时的不正常将局限于小范围模块内
- 模块化设计的五条原则：
  - 直接映射：模块的结构与现实世界中问题领域的结构保持一致
  - 尽可能少的接口：模块应尽可能少的与其他模块通讯
  - 尽可能小的接口：如果两个模块通讯，那么它们应交换尽可能少的信息
  - 显式接口：当A与B通讯时，应明显的发生在A与B的接口之间
  - 信息隐藏：经常可能发生变化的设计决策应尽可能隐藏在抽象接口后面



【内聚性】

- 又称块内联系。指模块的功能强度的度量，即一个模块内部各个元素彼此结合的紧密程度的度量。若一个模块内各元素（语名之间、程序段之间）联系的越紧密，则它的内聚性就越高。
- 所谓高内聚是指一个软件模块是由相关性很强的代码组成，只负责一项任务，也就是常说的单一责任原则。

【耦合性】

- 也称块间联系。指软件系统结构中各模块间相互联系紧密程度的一种度量。模块之间联系越紧密，其耦合性就越强，模块的独立性则越差。模块间耦合高低取决于模块间接口的复杂性、调用的方式及传递的信息。
- 对于低耦合，粗浅的理解是：一个完整的系统，模块与模块之间，尽可能的使其独立存在。也就是说，让每个模块，尽可能的独立完成某个特定的子功能。模块与模块之间的接口，尽量的少而简单。如果某两个模块间的关系比较复杂的话，最好首先考虑进一步的模块划分。这样有利于修改和组合。

更多请参考 王永迪的专栏 浅谈高内聚低耦合

SOLID原则

更多请参考 我理解的**SOLID** 浅谈**SOLID**原则的具体使用

S.O.L.I.D是面向对象设计和编程**(OOD&OOP)**中几个重要编码原则**(Programming Priciple)**的首字母缩写。

综述：设计模式前五个原则，恰恰是告诉我们用抽象构建框架，用实现扩展细节的注意事项而已：

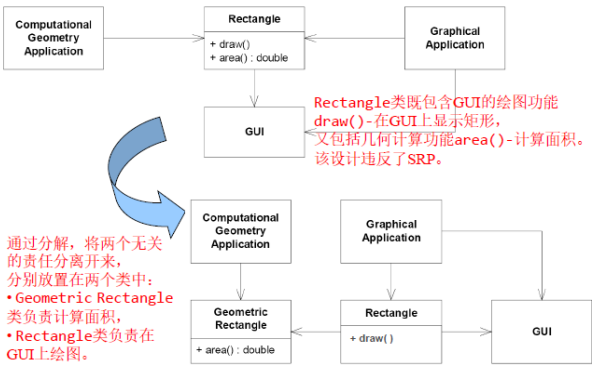
单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。而开闭原则是总纲（实现效果），它告诉我们要对扩展开放，对修改关闭。

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	里氏替换原则
ISP	The Interface Segregation Principle	接口分离原则
DIP	The Dependency Inversion Principle	依赖倒置原则

【**SRP** 单一责任原则】

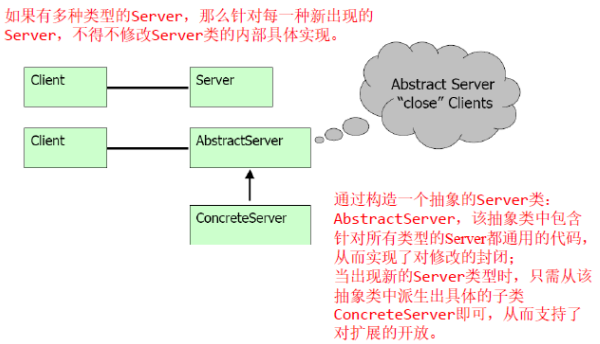
- 含义：需要修改某个类的时候原因有且只有一个。换句话说就是让一个类只做一种类型责任，当这个类需要承当其他类型的责任的时候，就需要分解这个类。
- 如果一个类包含了多个责任，那么将引起不良后果：引入额外的包，占据资源；导致频繁的重新配置、部署等。
- SRP是最简单的原则，却是最难做好的原则。

- SRP的一个反例：



【**OCP** 开放封闭原则】

- 软件实体应该是可扩展，而不可修改的。也就是说，对扩展是开放的，而对修改是封闭的。这个原则是诸多面向对象编程原则中最抽象、最难理解的一个。
  - 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。
  - 模块自身的代码是不应被修改的
  - 扩展模块行为的一般途径是修改模块的内部实现
  - 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。
- 关键解决方案：抽象技术。 使用继承和组合来改变类的行为。
- OCP的一个反例：



- OCP的一个例子：

```
// 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)
        {....}
}
```

```
14
15 class Shape { int m_type; }
16 class Rectangle extends Shape { Rectangle() { super.m_type=1; } }
17 class Circle extends Shape { Circle() { super.m_type=2; } }
```

上面代码存在的问题：

- 不可能在不修改GraphEditor的情况下添加新的Shape
- GraphEditor和Shape之间的紧密耦合
- 不调用GraphEditor就很难测试特定的Shape

改进之后的代码：

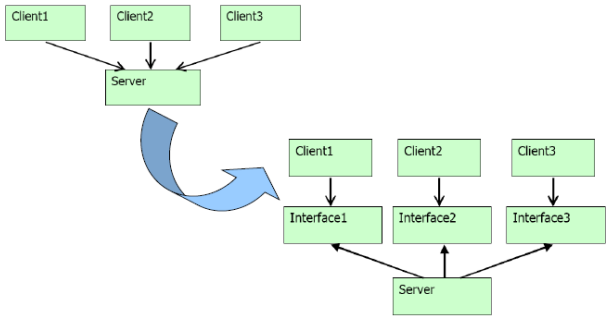
```
// Open-Close Principle - Good example
class GraphicEditor {
public void drawShape(Shape s) {
    s.draw();
    }
}
class Shape { abstract void draw(); }
class Rectangle extends Shape { public void draw() { // draw the rectangle } }
```

【LSP 里氏替换原则】

- Liskov's 替换原则意思是："子类型必须能够替换它们的基类型。"或者换个说法："使用基类引用的地方必须能使用继承类的对象而不必知道它。" 这个原则正是保证继承能够被正确使用的前提。通常我们都说，“优先使用组合（委托）而不是继承”或者说“只有在确定是 is-a 的关系时才能使用继承”，因为继承经常导致”紧耦合“的设计。
- 【软件构造】第五章第二节 设计可复用的软件 中有所描述

【ISP 接口分离原则】

- 含义:客户端不应依赖于它们不需要的方法。换句话说，使用多个专门的接口比使用单一的总接口总要好。
- 客户模块不应该依赖大的接口，应该裁减为小的接口给客户模块使用，以减少依赖性。如Java中一个类实现多个接口，不同的接口给不用的客户模块使用，而不是提供给客户模块一个大的接口。
- “胖”接口具有很多缺点。
  - 胖接口可分解为多个小的接口；
  - 不同的接口向不同的客户端提供服务；
  - 客户端只访问自己所需要的端口。
- 下图展示出了这种思想：



- ISP的一个反例

```
//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};
}
```

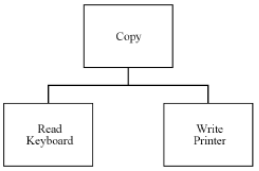
Solution: split into two

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

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

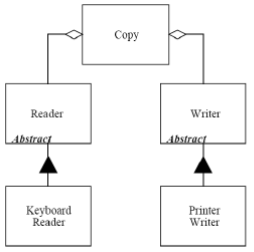
【DIP 依赖转置原则】

- 定义：
  - 高层模块不应该依赖于低层模块，二者都应该依赖于抽象
  - 抽象不应该依赖于细节，细节应该依赖于抽象
- 这个设计原则的亮点在于任何被DI框架注入的类很容易用mock对象进行测试和维护，因为对象创建代码集中在框架中，客户端代码也不混乱。有很多方式可以实现依赖倒置，比如像AspectJ等的AOP（Aspect Oriented programming）框架使用的字节码技术，或Spring框架使用的代理等。
  - 高层模块不要依赖低层模块；
  - 高层和低层模块都要依赖于抽象；
  - 抽象不要依赖于具体实现；
  - 具体实现要依赖于抽象；
  - 抽象和接口使模块之间的依赖分离。
- 一个具体的例子：




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

进行抽象改进后：




```
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);
  }
}
```

【SOLID 总结】

1. 一个对象只承担一种责任，所有服务接口只通过它来执行这种任务。
2. 程序实体，比如类和对象，向扩展行为开放，向修改行为关闭。
3. 子类应该可以用来替代它所继承的类。
4. 一个类对另一个类的依赖应该限制在最小化的接口上。
5. 依赖抽象层(接口)，而不是具体类。