5.1 Metrics and Construction Principles for Maintainability

可维护性的度量与构造原则

一级标题，二级标题，三级标题，四级标题，注，拓展，不懂

目录

5.1.1 definition

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：在软件产品已经发布后，出于修复软件的错误和改善软件的性能，对软件所做的修改。

软件维护的范畴不只包括软件的修复，还包括修复之后：对修复的测试，对代码的回归测试，以及对于文档的记录。

5.1.2 Types 类型

纠错性维护：当发布的软件产品出现错误时，进行的修复错误的维护。

适应性维护：对软件做出维护，使得其可以在变化的或者被改变的环境可以正常运行。

完善性维护：对软件做出维护，目的是扩展其原有的功能以及提升现有功能的效率。

预防性维护：对软件做出维护，预防可能出现的错误或者效率问题等等，未雨绸缪。

5.1.3 一些观点对于可维护性。

超过 90%的软件成本来自于软件的维护阶段。

软件维护时任何软件都不可避免要经历的过程，即使是最为成功的软件项目也不可避免。

软件维护不仅仅时运维工程师的工作，而是应该从软件的设计和开发阶段就应该需要考虑的部分。

5.1.4 可维护性的判断标准

Cyclomatic Complexity 圈复杂度

Definition

软件控制流不同路径的数量（由节点进行区别）。

具体计算方法：

下列计算均是建立在软件的控制流图被构建的基础上：

- ✧ 可以人为计算。
- ✧ 圈复杂度 = 分支流节点（谓词） + 1。
- ✧ 圈复杂度 = 控制流图中区域的个数。
- ✧ 圈复杂度 = 控制流图中边数 – 顶点数 + 2。

意义：

用于评判代码的结构复杂度，圈复杂度越高，代码的结构复杂度越高，需要的测试用例更多，可维护性越低。

Lines of Code 代码行数

意义：

是一种非常粗略的判断标准，当然行数越多，程序的复杂性越高，更难维护。

Halstead Volume

Halstead Volume 是基于操作符和操作数的数量的复杂性度量。

Maintainability Index (MI) 可维护性指数

计算一个在 0 到 100 之间的数字，基于 Halstead Volume (HV)，Cyclomatic Complexity (CC)，The average number of lines of code per module (LOC)，The percentage of comment lines per module (COM)等。

该值越高，说明该系统的可维护性更强。

Depth of Inheritance 继承的层次数

表示从继承的基类出发，所扩展的类的数量，大概就是继承关系树的高度。

该值越大，说明继承的关系树越深，越不易维护，越难以判断某个方法在某个地方被重写。

Class Coupling 类之间的耦合度

从参数，本地变量，返回值类型，方法调用，泛型实例化，模板实例化，接口的实现等方面，形式化的计算类之间的耦合度。

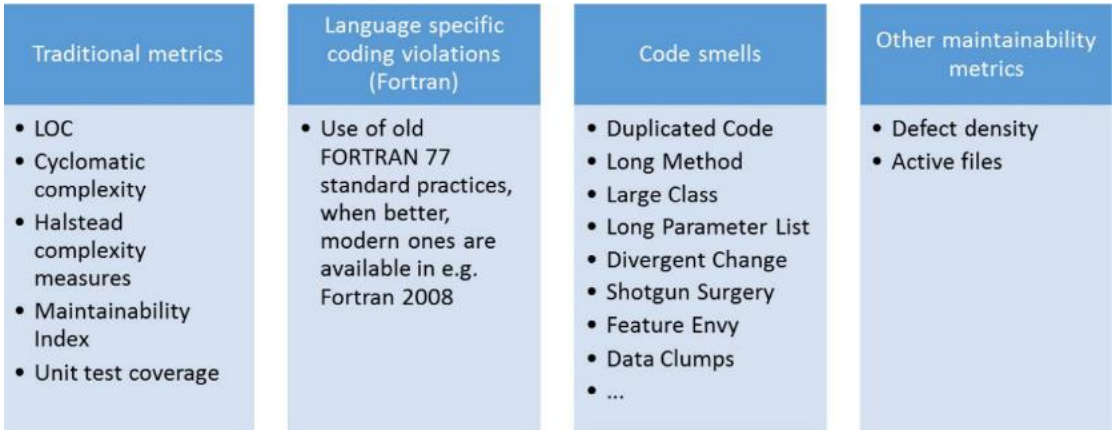
一个好的程序：多个模块之间是低耦合的，这样更加易于维护与复用，而每个模块内部又是高内聚的。

Unit test coverage 单元测试的覆盖度

表示一个程序进行测试的完备度。

当然越高，越可以说明源程序出现问题的可能性小，而且测试越完备，对于修改之后的回归测试更加方便。

其他评判方法：



5.1.5 Modular Design（复用性以及可维护性的编程模式：模块化编程与原则）

模块化编程是一种设计技术，强调将程序的功能分为独立的，可被替换的模块，模块与模块之间存在高内聚，低耦合的关系，以使每个模块都包含执行所需功能的一个方面所必需的所有内容。

模块化编程的方式是：分离功能点，而后对每个模块进行编程。这样的方法有利于降低编程的总体复杂程度。而且模块之间的接口定义良好的话，可以信息隐藏，增强模块的复用性。

注：模块化编程的思想不仅在面向对象的语言中存在，而且存在于结构化编程的部分。

模块化编程的标准

Decomposability (可分解性)

对于整个程序，可以将其拆分成为多个模块，这些模块都具有某种功能，这些可以被复用。

Composability (可组合性)

对于这些模块而言，可以容易进行使用，将之组合成为新的系统。

Understandability (可理解性)

对于拆分的每个模块，这些模块以及模块之间的关系都容易被理解。

Continuity (可持续性)

对于程序较小的改变不会引起体系结构的变化，而是只会影响有限的几个部分。

Protection (出现异常之后的保护)

当程序运行产生异常时，这些异常所影响的范围应当尽量小。

模块化设计的原则（针对于模块的分类）

Direct Mapping 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。

由于现实世界中的结构一般都是非常稳定的，所以以此划分的模块结构可以很好满足可持续性。可分解性时模块化设计最为基本的一个标准，直接映射当然满足。

Few Interfaces 更少的交互

模块应当尽可能少于其他模块进行通讯。

这样会降低模块之间的耦合度，更少的交互大概对于除可分解性之外的性质都有较好的影响。

Small Interfaces 交互量少

如果两个模块进行通讯，那么它们之间应该交换尽可能少的信息。

这同样会将降低模块之间的耦合度，并且会对可持续性和保护性产生影响。

Explicit Interface 显式的通讯

如果两个模块需要进行通讯，那么二者之间应当具有显式（直接）的交互，尽量不通过其他媒介。

显然，显式的通讯会增强程序的可理解性。显式的通讯也会使得程序模块之间的关系更加明确，增强其可分解性。显示的通讯使得程序模块之间的交互通道更加清晰。

Information Hiding 信息隐藏

将经常变化的设计决策尽量隐藏在抽象接口之后。也就是一个模块由两部分组成：接口类(展示功能）以及接口类的具体实现类。

影响可持续性 ，对于具体实现类的改变，不会造成程序需要大量改变。

拓展：Coupling and Cohesion（耦合度与内聚度）

Coupling（耦合度）

Coupling（耦合度）是衡量两个模块之间的依赖关系的一个指标，两个模块之间的依赖关系存在，如果一个模块的变化需要另一个模块做出对应变化。

耦合度的评判依据为：两个模块之间交互通道的数量，以及相互通道的复杂度。

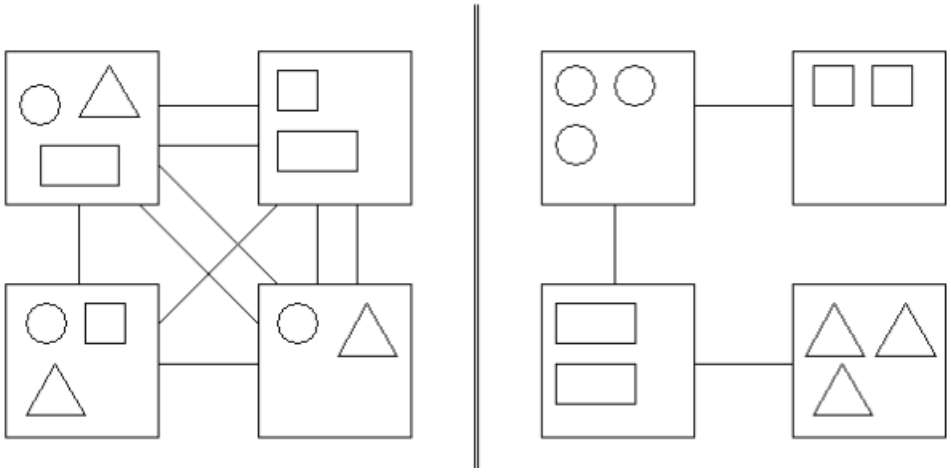
Cohesion（聚合度）

Cohesion（聚合度）是衡量一个模块之内的函数的关联密切程度。如果某个模块内的元素都为同一个目标工作，那么它就是高内聚的。

注：聚合度和耦合度是负相关关系。

Ex：左边图每个模块之间连线要明显多于右图，说明右的耦合度低于左。

左边图中每个模块内的函数明显不如右图中的函数统一，说明右的聚合度高于左。



5.1.6 OO Design Principles: SOLID（面向对象的设计原则：SOLID）

总结：

1. The Single Responsibility Principle（单一责任原则）是关于模块的划分，没什么用。
2. OCP 原则和 DIP 原则都是用抽象类来代替具体实现，隔离变化，易于扩展。
3. LSP 原则就是子类型，是抽象类的构成部分。
4. ISP 原则，接口的有效组合

The Single Responsibility Principle（单一责任原则）

There should never be more than one reason for a class to change.（一个类中不应该有多余 1 个原因使其发生变化。）

这里的原因并非指的是 bug 修复错误或者重构，而是人们对于该类的所要展示的功能需要发生变化，进而对其进行修改。

Gather together the things that change for the same reasons. Separate those things that change for different reasons.（这是该原则的另一个解释，与模块化编程的模块化分类有些相似）

个人认为，在某种程度上，该原则比较鸡肋，很难有具体实际意义。

但是这里所讲的“该类的所要展示的功能”（也就是职责）的颗粒定义是不怎么清晰的。以下列图示进行说明：如果说将连接管理与数据传送定义为两个职责，那么进行拆分是合理的，但是如果将之定义为一个职责，也无不可。

这个问题需要具体情况进行具体分析，可以根据是否会预料到其改变进行划分。如果说一直进行细分，指导每个函数，这样会大大增加代码的层次结构，得不偿失。

```
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();
}
```

The Open-Closed Principle（OCP 原则）

使用抽象类，达到该类对于扩展性是可以的，但是不可以对该类进行修改。

具体实现就是对每个类中的某一对象使用抽象类进行表示，之后再客户端中，调用该函数时，使用不同的实现类进行赋值，和那个策略模式的思想差不多。

Ex：给出一个简单例子，其他可以参见 4-3 中的策略模式。

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

```
class Shape {
    int m_type;
 }

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

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

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

```
// 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
   }
}
```

Liskov Substitution Principle (LSP)

详见 4-2

Interface Segregation Principle (ISP)

接口中的函数应当是：其需要实现类的所有的共性方法。

接口中不应该存在某个实现类不需要的方法，如果出现该情况，需要对接口进行分离，实现类可以进行其需要实现的接口的自由组合。

Ex:

```
//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();
}
```

优化后的实现：

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

interface Feedable{
   public void eat();
}
ManWorker implements Workable, Feedable {
    void work() {…};
    void eat() {…};
}
RobotWorker implements Workable {
    void work() {…};
}
```

Dependency Inversion Principle（依赖倒置原则）

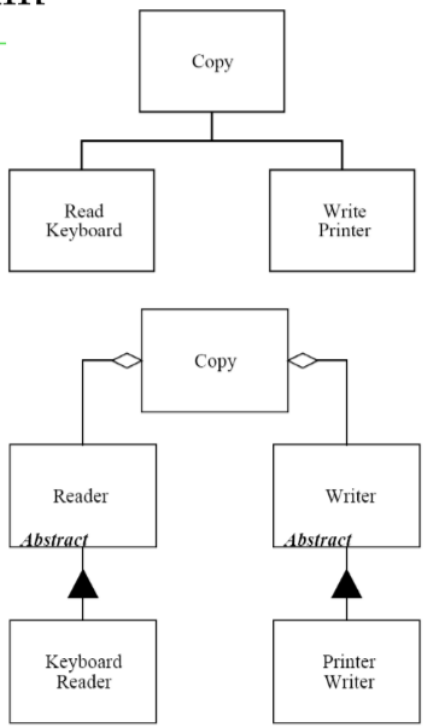
高层模块不应该依赖于低层 模块,二者都应该依赖于抽象。
抽象不应该依赖于实现细节,实现细节应该依赖于抽象
这个与 OCP 相同，均是用抽象隔离变化。与策略模式相同

Ex:

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);
  }
}
```



5.1.7 OO Design Principles: GRASP

GRASP 是如果为类指派职责（也就是功能划分）的一系列原则。
没有进行研究，可以在维基百科中进行补充.