Lecture6 Metrics

1. 衡量项目进度

- 进度
  - 工时
  - 报告的 Bug
  - 完成的故事点
- 项目

技术指标

项目的度量称为**技术指标 Technical Metrics**

- 代码的大小
  - 文件的个数
  - 类的个数
  - 进程个数
- 代码的复杂度
  - 依赖/内聚度/耦合度
  - 嵌套深度
  - 循环复杂度

非技术指标

- 参与项目的人数
- 花费的时间，金钱
- 发现/报告的 Bug
  - 测试/开发发现的
  - 用户发现的
- 修好的 Bug，添加的特性

2. 衡量系统指标

代码行数 Lines Of Code

- 很容易去衡量
- 所有的项目都会有代码
- 与构建的时间有关

不是一个很好的标准

```
copy(char *p,*q) {while(*p) *q++ = *p++;}
```

```
copy(char *p,*q) {
    while(p) {
        *q++ = *p++;
    }
}
```

- 上面两段代码是一样的，但是如何衡量行数呢?
- 使用**源代码行数（Source Lines of Code SLOC）**
    - 只计算源代码（忽略空白和注释）

语言的选择也是一个问题

- 汇编语言的代码可能是 C 代码的 2-3 倍长
- C 代码可能也是 Java 代码的 2-3 倍长
- ...

代码长度在以下情况下是一个技术指标

- 使用相同的语言
- 标准的格式
- 代码已被审核

复杂度 Complexity

复杂的系统

- 难以理解
- 难以更改
- 难以重用

一些复杂度的指标

- 循环复杂度 Cyclomatic complexity
- 功能点 Function Points
- 耦合和内聚 Coupling and Cohesion
- 面向对象特定指标 OO-specific metrics

循环复杂度 Cyclomatic complexity

> https://www.youtube.com/watch?v=UdjEb6t9-e4

- 逻辑复杂性的度量

- 表示为了执行每一个程序语句需要多少次测试

= Number of branches (if, while for) + 1

- 为什么要 +1，因为如果没有任何的分支，事实上你也需要至少一个测试来测试代码的功能

不同的工具会给出不同的循环复杂度

- 原来的论文对如何推导控制流图不清楚
  - 不同的实现会给相同的代码不同的值
  - 例如，在下面的代码情况下
    - Eclipse Metrics Plugin 会给出 2（一整个 if 分支 +1 = 2）
    - GMetrics 会给出 4（if 分支中的 3 个子句 + 1 = 4）
    - SonarQube 会给出 5

```
int foo(int a, int b){
    if(a > 17 && b < 42 && a+b < 55){
        return 1;
    }
    return 2;
}
```

测试视图 Testing View

- 循环复杂度是指通过该过程的**独立路径**的数量
- 给出了执行控制图的每条边所需的测试数量的上限

指标视图 Metrics View

- McCabe 发现，循环复杂度大于 10 的模块很难测试，而且容易出错
- 寻找具有高循环复杂性的程序并重写它们，重点测试它们，或者集中审查它们
  - 是重构、逆向工程、再造工程的良好目标

功能点 Function Points

- 测量系统“功能”的方法
- 测量一个系统应该有多大
- 用来预测大小
- 有几种计算功能点的方法，都很复杂
- 大多数都是特有的

衡量方法

- 计算输入的数量，输出的数量，算法的数量，数据库中的表数
- “功能点”是上面说到的功能，加上复杂性和开发人员专业知识的影响因素
- 需要训练来测量功能点

耦合和内聚 Coupling and Cohesion

- 耦合 Coupling：模块之间的依赖性（不好）
- 内聚 Cohesion：模块之内的依赖性（好）

衡量方法

- 共享变量的数量和复杂性
  - 模块中的函数应该共享变量
  - 不同的模块的函数**不应该共享变量**
- 参数的数量和复杂性
- 被调用的函数/模块的数量
- 调用我的函数/模块的数量

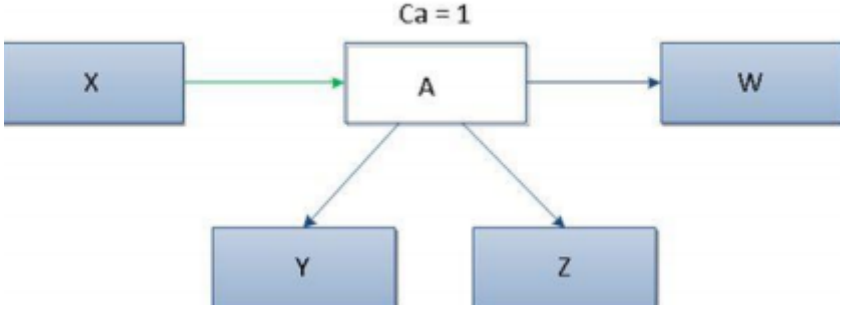
Dhama 耦合指标

```
Module coupling = 1 / ( number of input parameters +
                        number of output parameters +
                        number of global variables used +
                        number of modules called +
                        number of modules calling
                        )
```

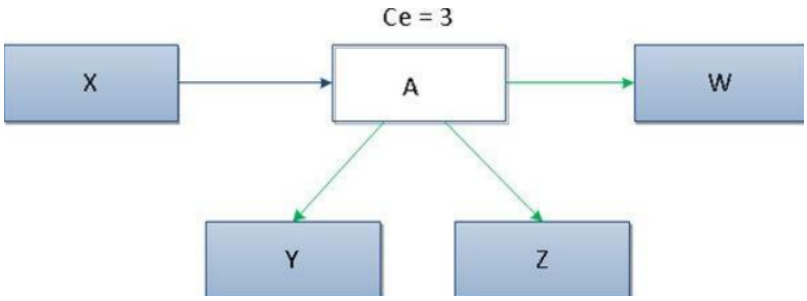
- 0.5 是低耦合
- 0.001 是高耦合

Martin 耦合指标

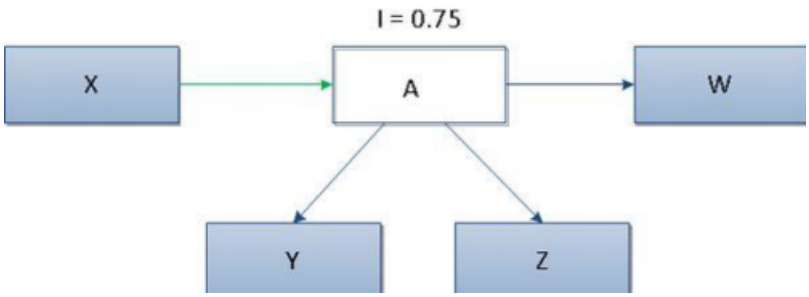
- Ca：**传入耦合 Afferent Coupling**：此模块**之外**依赖于此模块内部的类的类的数量



  - 测量传入依赖项
  - 使我们能够测量剩余包对分析包中变化的敏感性
  - 高的 Ca 指标通常表明高的成分稳定性
- Ce：**传出耦合 Efferent Coupling**：此模块内依赖于此模块外的类的类数



    - 给定包中的许多类，这取决于其他包中的类
    - 度量类之间的相互关系
    - 使我们能够衡量包对它所依赖的包的变化的脆弱性
    - Ce> 20的高值表示包的不稳定性
- 不稳定性 Instability $= \frac{Ce}{Ce+Ca}$



    - 很多的 Ce 和不是很多的 Ca（值 I 接近1）是不稳定的
    - 很多的 Ca 和不是很多的 Ce（值 I 接近 0），意味着组件的稿稳定性，因为他们的责任更大，所以更难修改

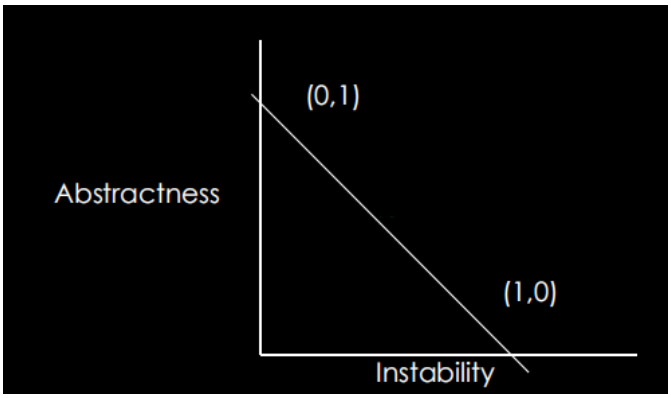
抽象性和不稳定性 Abstractness & Instability

$$I = \frac{Ce}{Ce + Ca}$$

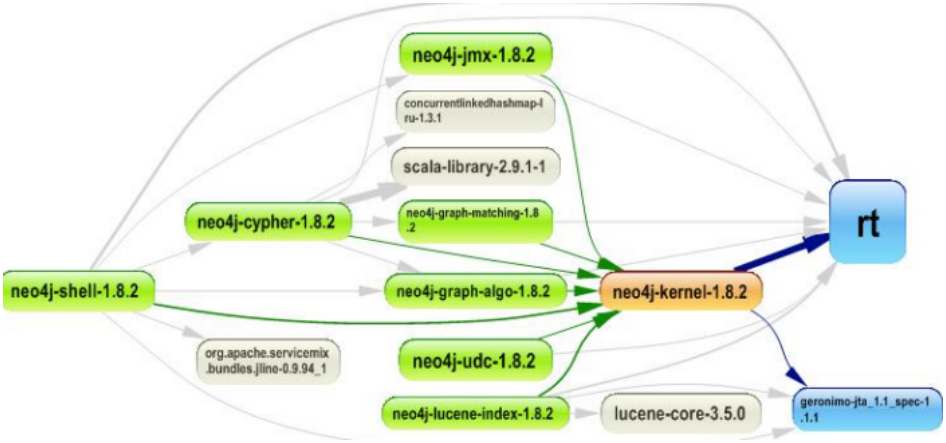
- 传出耦合 / (传入耦合 + 传出耦合)

$$A = \frac{T_{abstract}}{T_{abstract} + T_{concrete}}$$

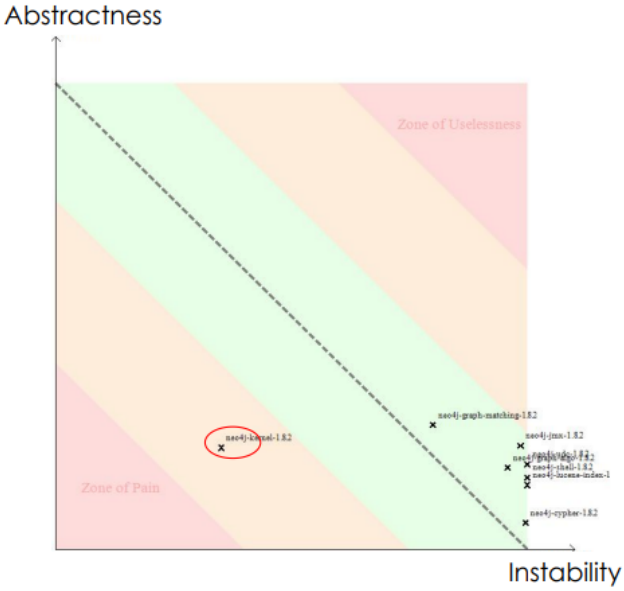
- 模块内的抽象类个数 / 模块内的类的个数



示例：Neo4J（图形数据库）



- Neo4j 包含了很多 jar 包，并且它们都依赖于 Neo4j-Kernel
- 引入的依赖项很多，而传出的依赖项不多(值 I 接近 0)，因为他们的责任更大，所以更难修改



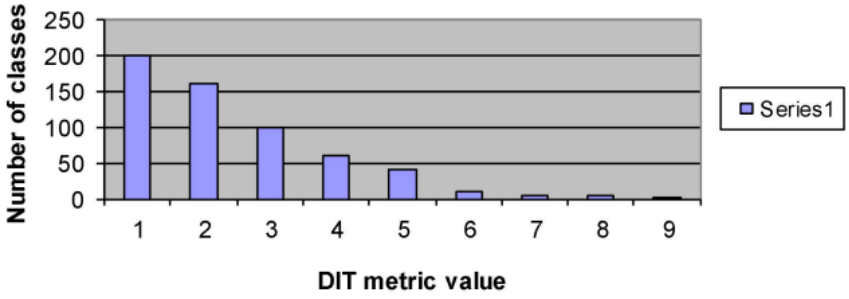
- Neo4j 内核有很多类依赖于它，所以它位于左侧
  - 离开橙色区域到绿色区域应该更抽象一些
- 避开 Zone of Pain 部分很重要
  - 如果一个 jar 在这个区域内，对它的任何更改都会影响很多类，并且很难维护或演化这个模块

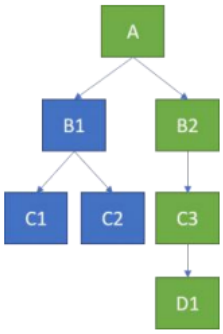
面向对象特定指标 OO-specific metrics

每类加权方法 Weighted Methods Per Class (WMC)

- 类的 WMC 是类中方法复杂度的总和
- 可能的方法复杂度
  - 1（方法的个数）
  - 代码行数
  - 方法调用的数量
  - 循环复杂度
- 方法的数量和方法的复杂性预测了开发和维护一个类所需的时间和精力
- 类中方法的数量越多，对子类的潜在影响就越大
- 具有大量方法的类更有可能是特定于应用程序的，而且可重用性更低

继承树的深度 Depth of Inheritance Tree (DIT)





- 如上图所示，树的深度是 4
- 从类到根的最大长度
- 一个类在层次结构中越深，它继承的方法就越多，因此更难预测
- 一个类在层次结构中越深，它重用的方法就越多
- 越深的树越复杂

子类个数 Number of Children (NOC)

- 直接子类的数量
- 子类越多，重用就越多
- 由于子类的误用，一个类可能有很多子类
- 一个有很多子类的类可能是非常重要和需要的
- 几乎所有的类都只有 0 个子类
- 只有非常少的类会有 5 个以上的子类

对象类的耦合度 Coupling between Object Classes (CBO)

- 与一个类耦合的其他类的数量
- 如果类 A 中有调用类 B 方法的方法，则类 A 与类 B 耦合
- 希望只与继承层次中的抽象类耦合
- 耦合使得设计难以更改
- 耦合使类难以重用
- 耦合是一个类测试难度的度量

类的响应 Response for a Class (RFC)

- 类中或被类调用的方法的数量
- 类的响应集是一组方法，可以在响应该类对象接收到的消息时执行这些方法
- 如果一个消息的响应要调用大量的方法，测试就会变得更加复杂
- 可以从类中调用的方法越多，类的复杂性就越大

方法的内聚性缺乏 Lack of Cohesion in Methods (LCOM)

- 不共享实例变量的方法的个数减去共享实例变量的方法的个数
- $Num_{don't\ share-instance} - Num_{shared-instance}$
- 方法的内聚性是封装的标志
- 缺乏内聚意味着类应该被拆分

3. 使用指标

一些使用方式

- 衡量每个程序员每月产生的代码量，给高产量的程序员大幅加薪
    - 有很多评估的方式，有时候会使用这个
- 衡量模块的复杂性，找到最复杂的那个然后进行重写
- 使用功能点 function point/ 故事点来确定一个系统需要多少行代码，当您编写很多行代码时，停止并交付系统
- 跟踪进度、代码评审、计划

跟踪进度 Tracking Progress

管理评审 Manager Review

- 经理周期性写报告
- SLOC 的增长
- 提出和被修复的 Bug 数量
- 每个功能点的 SLOC (用户描述)，每个程序员的 SLOC (每个程序员的用户描述)

信息发射源 Information Radiator

- 让整个团队都知道项目的状态
- 绿色/红色测试运行程序
- 预测/实际用户故事挂图
- 预测/实际SLOC 挂图
- 显示指标每日变化的Web页面(通过每日构建计算)

代码评审 Code Review

- 在代码评审之前查看指标
    - 代码覆盖率：没有覆盖测试的代码通常可能有 bug
    - 大的方法/类
    - 报告了很多 bug 的代码

计划 Planning

- 需要预测付出的努力
    - 确保你想在你的项目中这样做
    - 按时交货
    - 雇佣足够多的人
- 预测组件和 SLOC
- 预测故事点 & 花费的时间（XP）

- 开发人员/技术领导/管理层的意见