第一单元总结训练

OO课程组

2019

计算机学院

本单元教学目标设置

- 建立面向对象程序的认识
  - 识别出至少三个关键类：输入处理、主控、核心数据管理
- 认识对象的关键特性
  - 可变性、相似性
- 理解和初步掌握层次化抽象和设计方法
  - 按照数据/行为建立抽象层次
  - 层次化是架构设计的最重要/基础方法
- 训练
  - 递进式的三次求导作业
  - 两次在线实验

面向对象是一种思维方法

- 对象是一组具体的个体，各自维护自己的状态
  - 对象具有自治性
- 由统一的类所定义/实例化出的对象，虽然能力相同，但却是不同对象，相互独立发展。一个对象出错，不意味另外一个对象必然出错
- 对象对外提供无差别服务，任意一个对象只要找到该对象，均可使用其提供的服务
- 所谓“面向”，即把对象作为基本单位来规划设计程序的行为
  - 数据管理行为
  - 对象协同行为
  - 用户交互行为
  - 计算控制行为

面向对象是一种思维方法

- 三个基本问题
  - 如何管理对象
  - 如何建立对象之间的层次关系
  - 如何管理层次关系

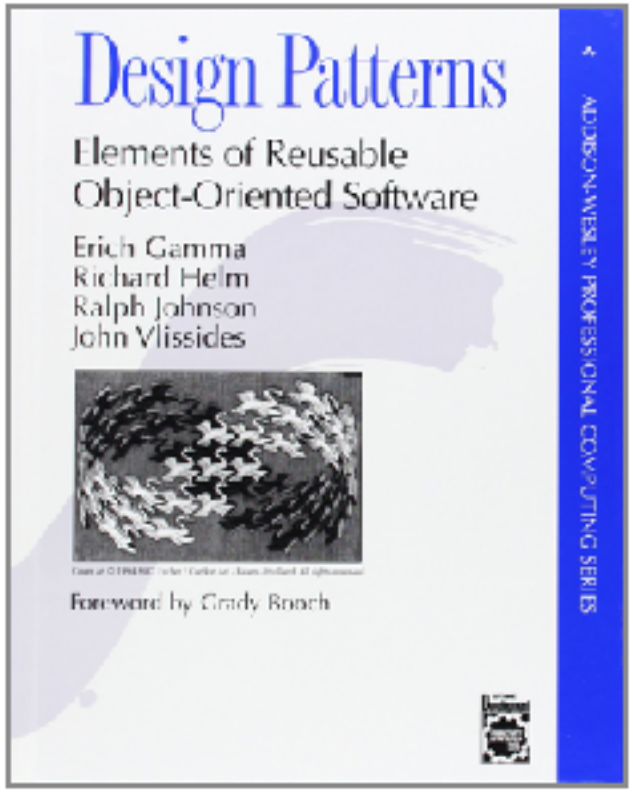
面向对象是一种思维方法

- 对象管理
  - 谁来管
    - 内部要求，自己构造，自己管理
    - 外部要求，外部构造的对象对自己有帮助
  - 如何管
    - 个体枚举
    - 静态数组
    - 可伸缩容器
  - 如何用
    - 存储数据
    - 层次代理

表达式项的创建与管理问题

- 本单元训练中的表达式项涉及多项式和三角函数式
- 组合规则更是复杂，且可以递归组合
- 扫描输入并构造相应的表达式项是一个重要的问题
- 应使用设计模式来重构

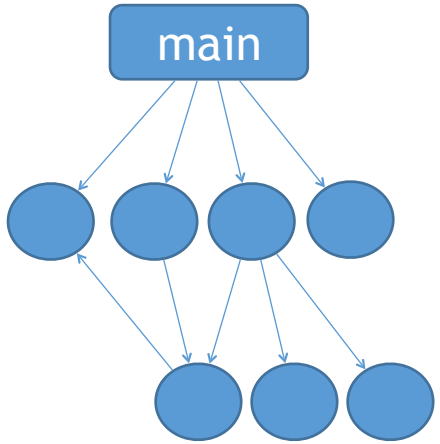
对象构造模式(creational pattern)
Singleton Pattern
Factory Pattern
Prototype Pattern
Builder Pattern
Object Pool Pattern

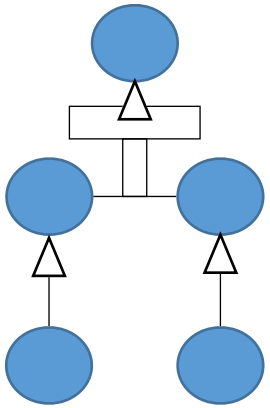


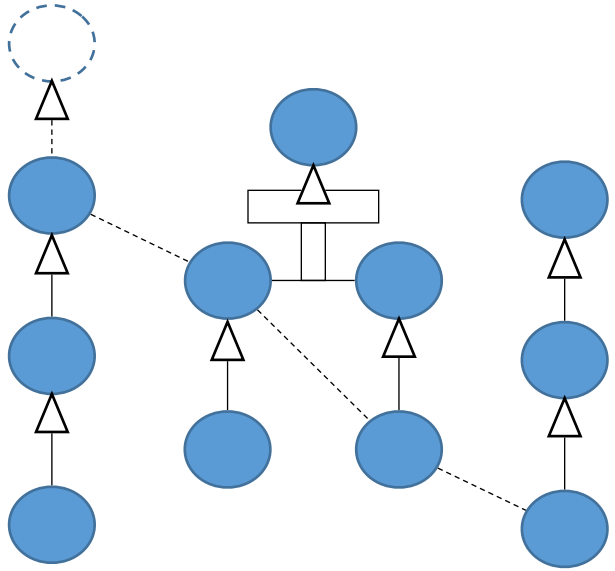
A pattern is a ***regularity*** in the world, man-made design, or abstract ideas. As such, the elements of a pattern ***repeat*** in a ***predictable*** manner. --wikipedia

面向对象是一种思维方法

- 建立层次关系
  - 按照所属关系形成的层次（管理层次），常常会有冗余管理（引用共享）
  - 按照数据抽象形成的层次（继承层次）
  - 按照行为抽象形成的层次（继承层次或接口层次）
  - 三种层次关系可以叠加







面向对象是一种思维方法

- （叠加的）层次关系管理
  - 对象定位
    - 在容器中找到所关注的对象
    - 通过传递的共享引用
  - 归一化管理
    - 通过上层抽象类或接口来无差别引用和使用相关的对象
    - 区别是什么?
  - 协作结构/模式
    - 赋之以角色：Factory、Observer(publish/subscribe)、Visitor...

几乎大部分design pattern都涉及到综合使用两种层次关系：继承层次和接口实现层次

三次作业的设计意图

- 第一次作业的设计目标
  - 核心数据
    - 幂函数
    - 幂函数的线性组合（多项式）
    - ➔对象管理层次结构
  - 核心操作
    - 幂函数求导：独立行为
    - 多项式化简：层次行为（主控、被控）
  - 建立程序鲁棒性概念
    - 输入处理的鲁棒性：输入结构异常、超长输入
    - 数据管理的鲁棒性：超大数的管理、未知对象数量
    - 计算的鲁棒性：超大数的计算、异常捕捉

三次作业的设计意图

- 第一次作业的训练目标
  - 掌握gitlab的使用
    - 什么是一次commit?
  - 熟悉代码风格检查
    - 为什么强调代码风格?
  - 重视和逐步掌握测试方法与技巧
    - 按照输入结构的测试设计
    - 中测的合理玩法
  - 熟悉和参与技术交流

三次作业的设计意图

- 第二次作业的设计目标
  - 通过迭代促使认识到代码结构的重要性
    - 好的设计可以更好适应需求的变化
  - 引入新的项（三角函数）： sin, cos
  - 跨幂函数和三角函数的组合
    - 线性组合
    - 乘积组合
  - 求导操作
    - 带来结构的变化
    - 线组合求导→线性组合
    - 乘积组合求导→线性组合（乘积组合）
  - 优化操作
    - 分解、合并
    - 搜索问题?

三次作业的设计意图

- 第二次作业的训练目标
  - 对第一次作业代码的bug修复
  - 代码重构
    - 如何建立管理层次关系（线性组合、乘积组合），第一次作业中相当多同学并未意识到层次关系
    - 基于正则的输入处理，避免大正则
    - 有一小部分高手已经提前建立抽象层次！
  - 强化了中测的鲁棒性测试用例，引导大家更加重视功能性测试和功能性bug
  - 性能提供了足够的“优化”空间，供有余力同学施展
    - 在架构与性能之间进行平衡

三次作业的设计意图

- 第三次作业的设计目标
  - 要求使用抽象层次（继承或接口实现）建立不同类型项之间的层次关系
    - 同时能够进行归一化处理
  - 增加新的组合规则
    - 线性组合：f(x)+h(x)
    - 乘积组合： f(x)*h(x)
    - 嵌套组合： f(h(x))
    - 组合规则可递归应用
  - 抽象层次建立线索
    - 求导
    - 化简

三次作业的设计意图

- 第三次作业的训练目标
  - 掌握和应用继承、接口和多态机制
  - 以统一的架构来整合三次作业的功能
  - 理解和体会新的架构对于需求变化的应对灵活性
  - 理解和掌握新的架构下的测试

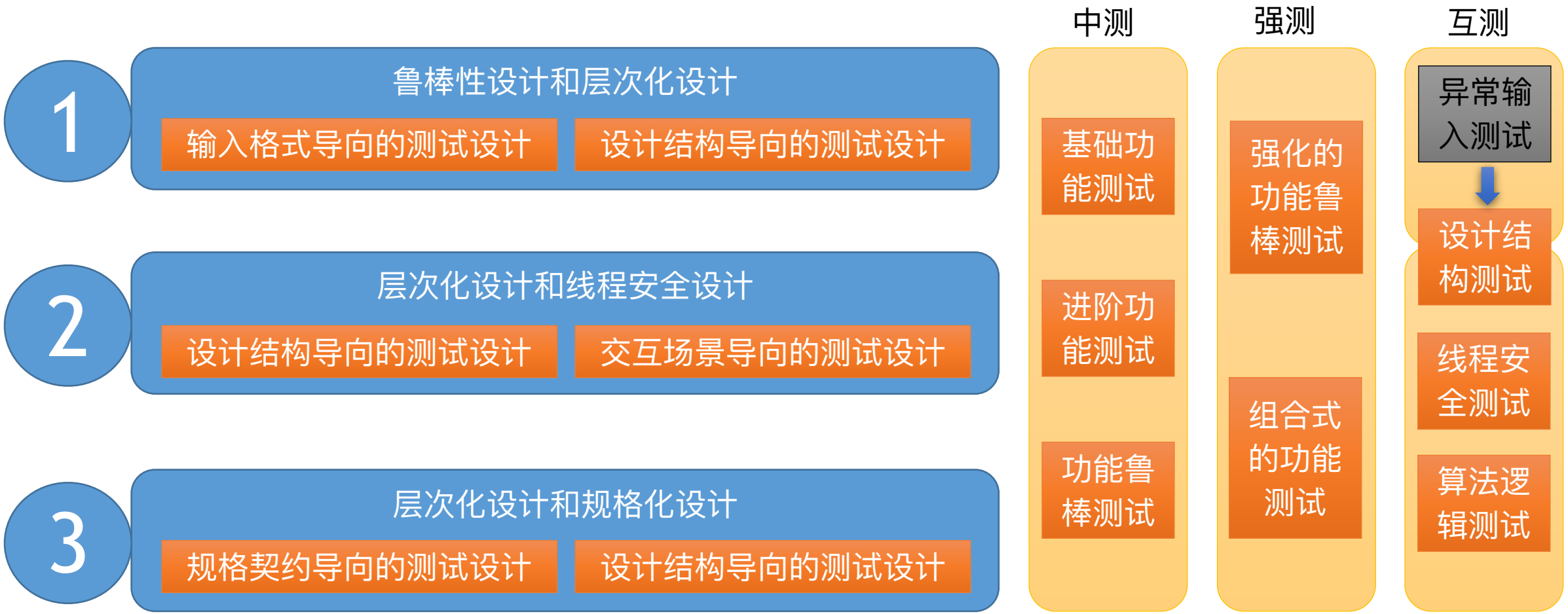
三次作业的测试实践

- 黑盒测试关注功能性和鲁棒性
  - 依据需求来设计测试用例
  - 正常测试用例~异常测试用例
- 输入的组合
  - 虽然是单输入，但仍具有明确的pattern和组合性
    - 基本项、组合规则
  - 每个基本项都要出现
  - 每种组合规则都要出现
  - 每种符号模式都要出现
  - 每种系数模式都要出现
  - ...

三次作业的测试实践

- 诚如有同学所言
  - 课程所使用的测试规则，与其是发现别人的bug，不如说是让每个同学提前发现自己的bug，在测试中提升对程序编码质量和设计质量的理解，进而获得提升
- 三次作业之间具有很强的递进性
  - 测试也是如此
  - 后一次作业兼容前一次作业的功能和测试用例
  - 便于开展回归测试
  - 同学们应思考在这种迭代开发模式下，如何管理好测试的迭代?

作业设计及其测试路线图



你的程序永远可能有Bug

- 中、强测全部通过的情况下，互测中被Hack的Bug数量分布：

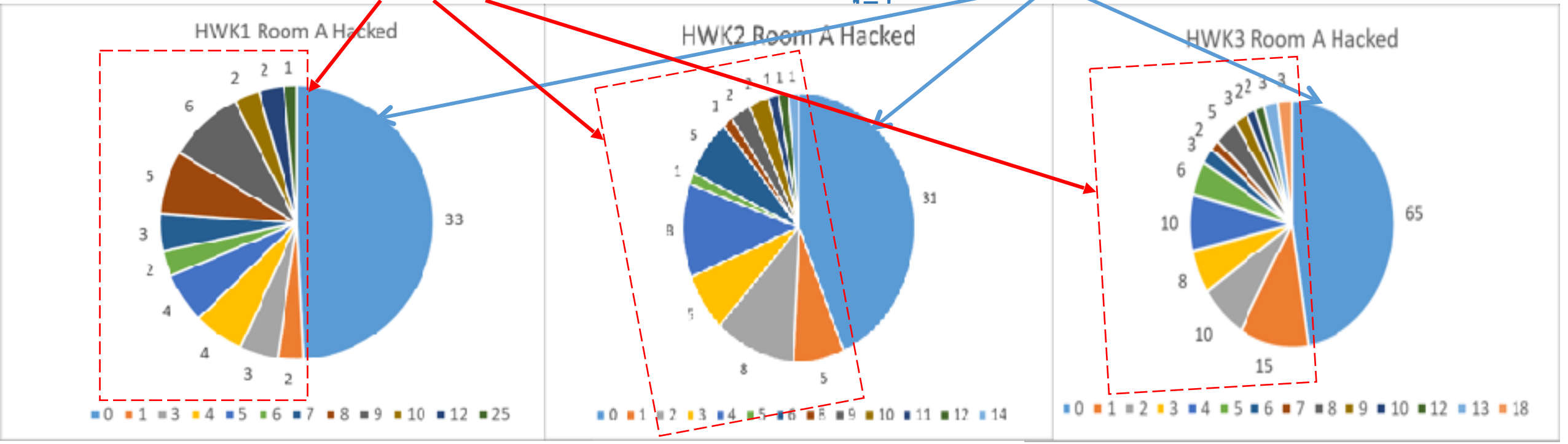
第一次作业
- ➢ 50.7%仍然存在Bug
- ➢ 有人被找出25个Bug！

第二次作业：
- ➢ 56.3%仍然被找出Bug
- ➢ 有人被找出14个Bug

第三次作业：
- ➢ 52.6%仍然被找出Bug
- ➢ 有人被找出18个Bug

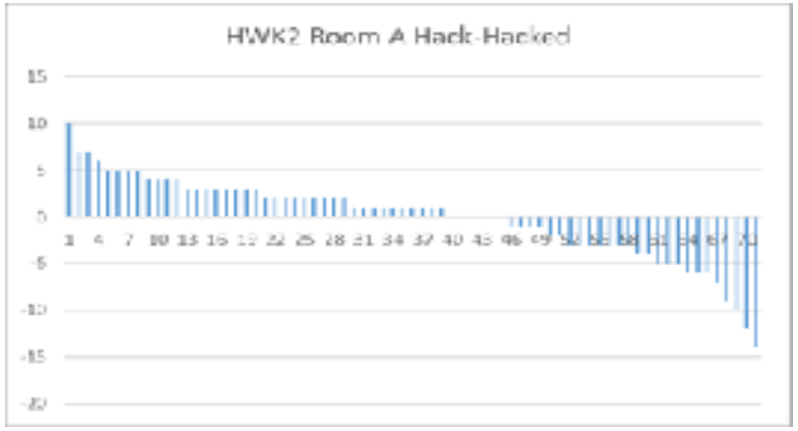
强测没问题，照样有Bug！

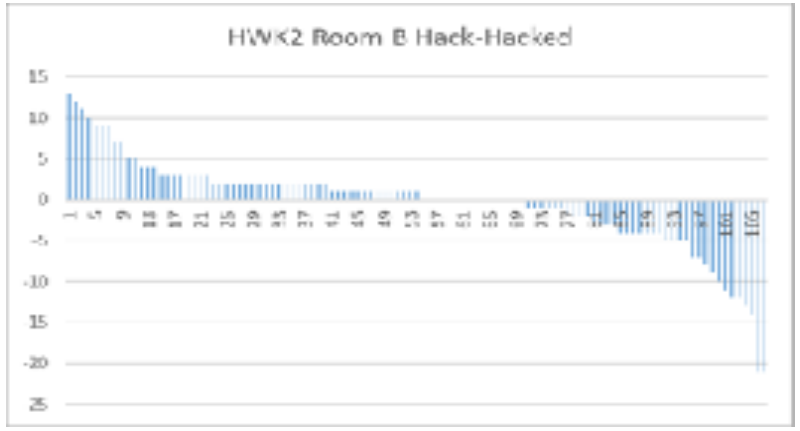
居然没被互测出Bug，我不信！

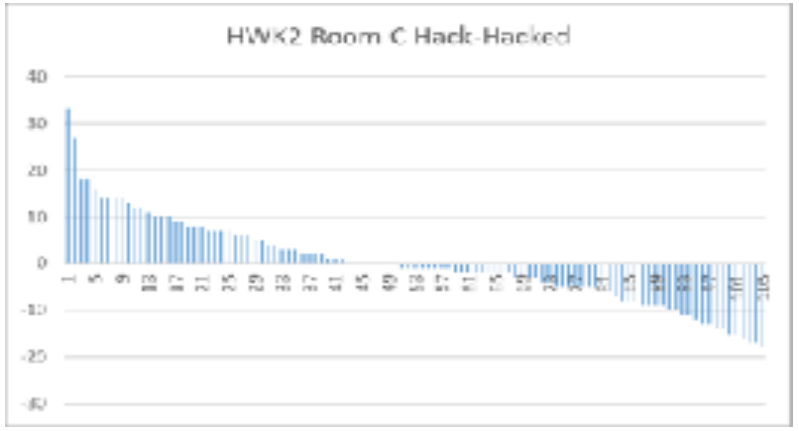


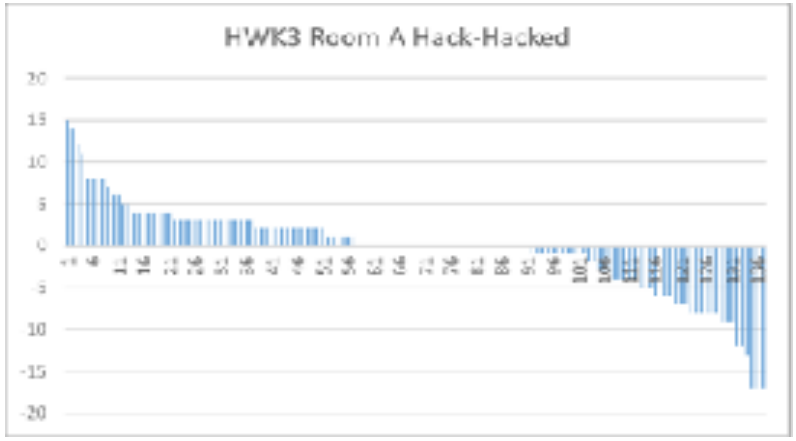
无论分在哪个Room，能否得分取决于你自己

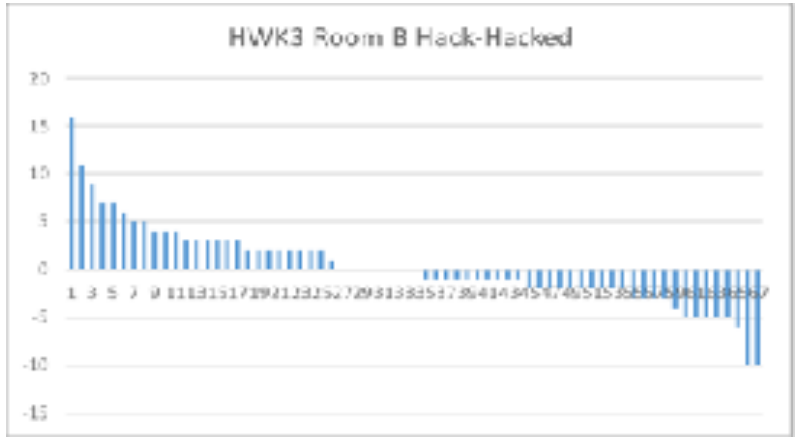
- 任何一个Level的room，得分者/失分者和得失分处于相对的平衡状态
- Level越高，互测得分越难
  - A：最高得分为10；B：最高得分为13；C：最高得分为34；
  - A：最高得分为15；B：最高得分为16；C：最高得分为19（WF不再是测试主流）；

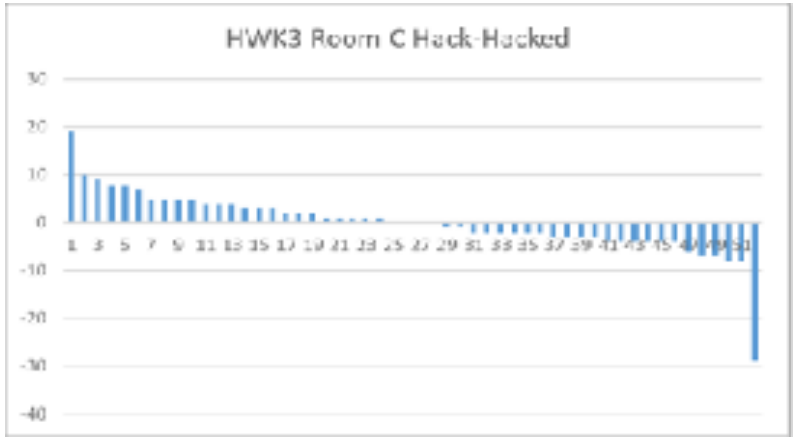










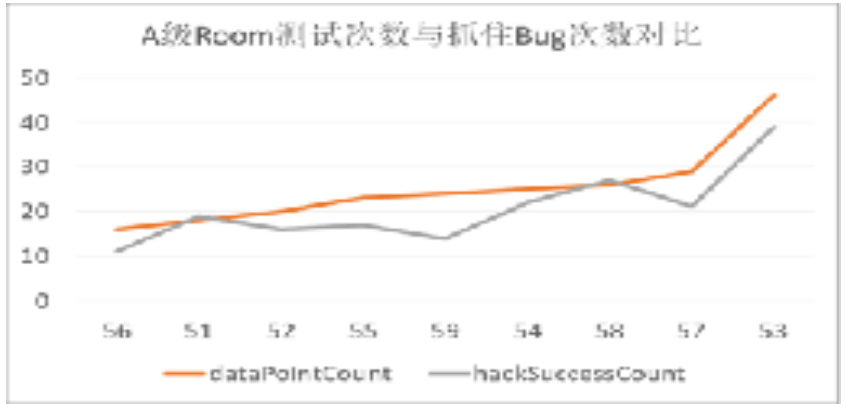


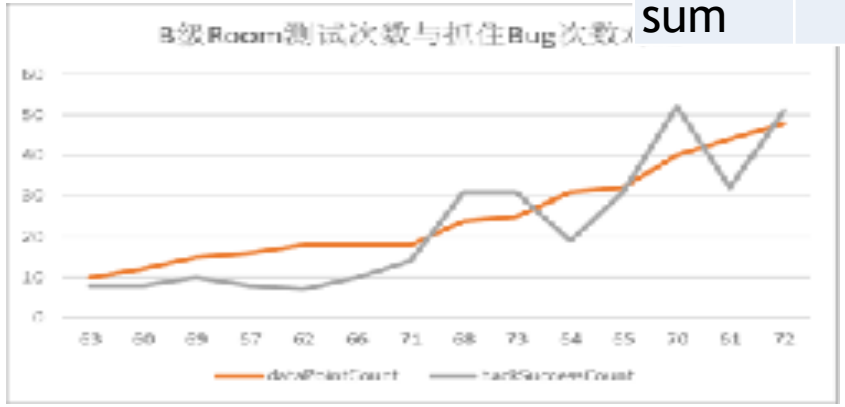
勤劳致富

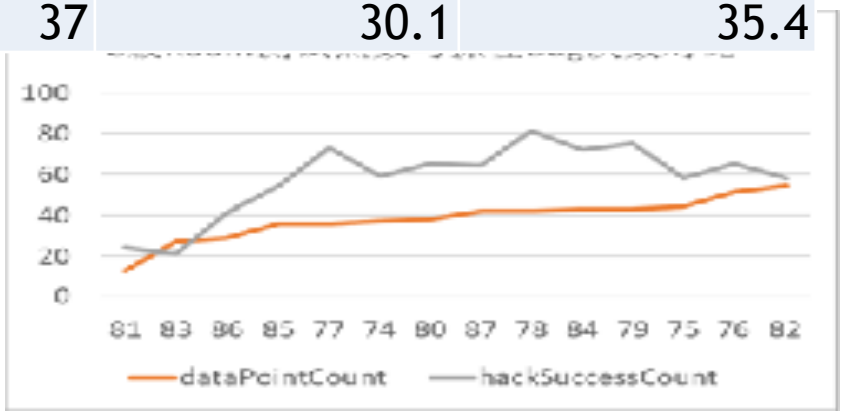
- 多做测试，才有可能得高分
  - 各级别能够发现Bug的次数与多开展测试皆有同向趋势。

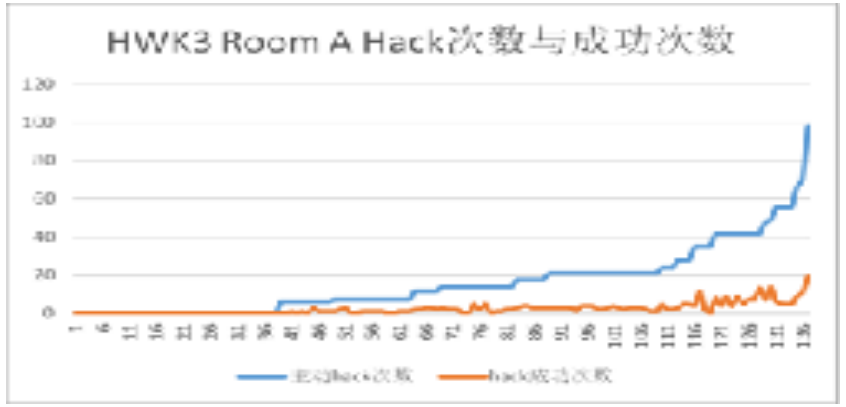
各级别房间所提交的测试点，
与Hack成功测试点的平均值

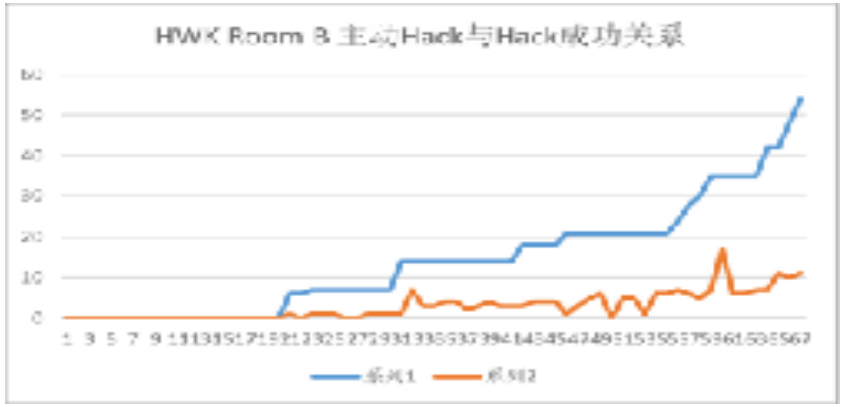
	num	ave(提交测试点)	ave(hack成功)
a房间	9	25.2	20.7
b房间	14	25.1	22.3
c房间	14	38.2	57.9
sum	37	30.1	35.4

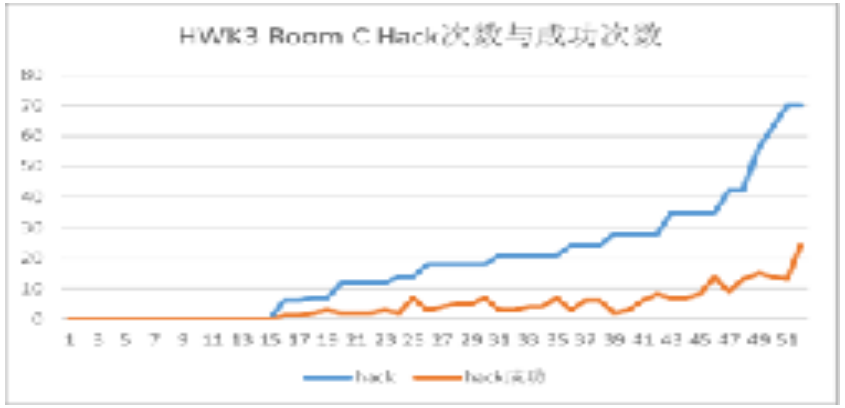








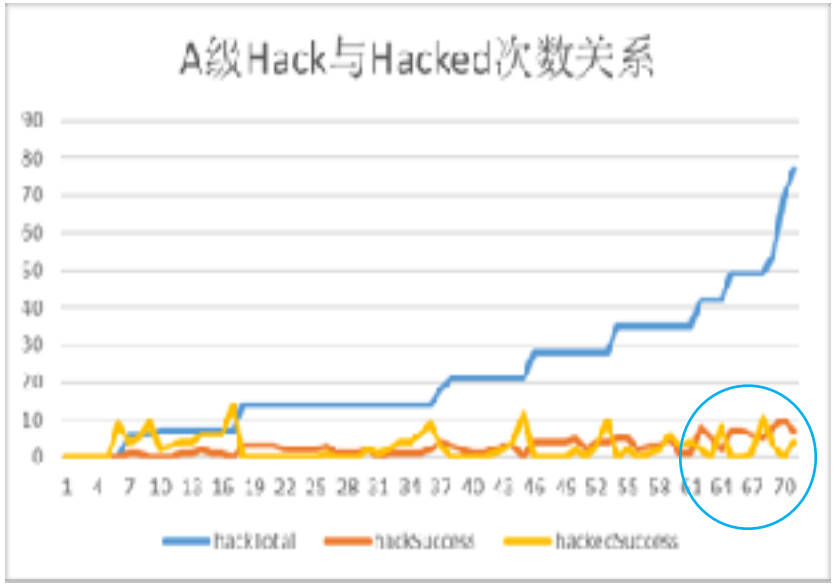


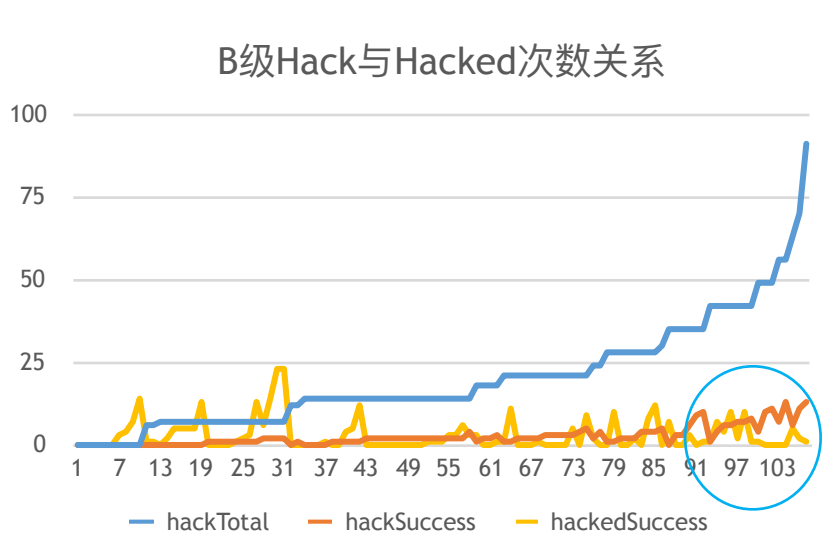


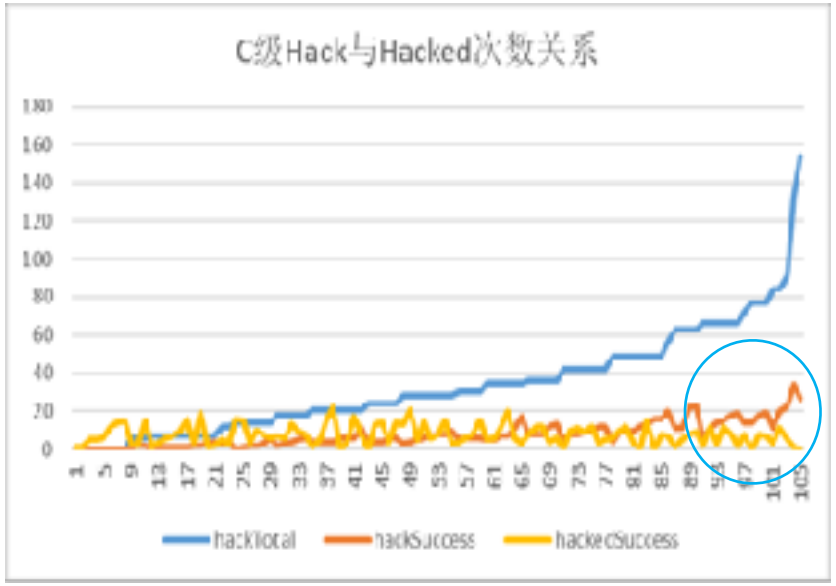
更有效地发现Bug

- 梳理测试内容，构建分类树
- 审视自己，大家都面临同样的疑惑和问题
- 越复杂的设计越容易出错
- 越巧妙的设计，适应场景的能力越弱
- 从设计与实现的逻辑分析入手
- ……

- 你发现Bug的能力，
与你出错的机率成反向。
- **知道如何出错，**
就能知道如何避免出错！







测试的基本原则

- 独立视角，不能把自己限定成设计和开发人员，怎么用就怎么测
- 尽早介入，在需求确定时就开始，设计之初就思考怎么测试每个类
  - TDD：Test Driven Development
  - 规划输入分类树
- 可靠追溯，一旦测试发现错误，必须能定位到相应的代码和数据
- 能够复现，必须用规范的方式复现
  - 最好能够以最精简的方式复现错误
- 有序开展，从小规模-〉大规模
- 理解测不全，再好的测试也不能穷举所有路径，所以要控制软件模块的规模，软件模块（类）越大，测不全带来的问题越严重

测试的重要性和局限性

- 是质量的重要保证手段之一。但是内在质量决定外在质量，软件的质量是设计出来的。
- 能够帮助提高设计者的思维。从用户的角度去思考软件的设计和编写，提高防错性意识，从根本上来说，程序设计要有怀疑一切的思想准备，任何函数的调用（甚至系统函数）都有失效的可能。
  - 通过主动防御来处理可能的失效
- 测不全问题始终存在，就会有潜在缺陷，核心程序需要进行严格的证明（如航空航天弹载、星载和箭载系统等）
  - 基于规格来证明
- 测试和使用环境紧密相关，必须通过严格的系统联试，用户的行为习惯也不能做过多的预期限制（是用户怎么用，而不是软件产品能做什么）

虽然软件产品质量在细节上可以由代码，评审和测试进行改进
但总体质量基本性质不会改变，除非进行重新设计

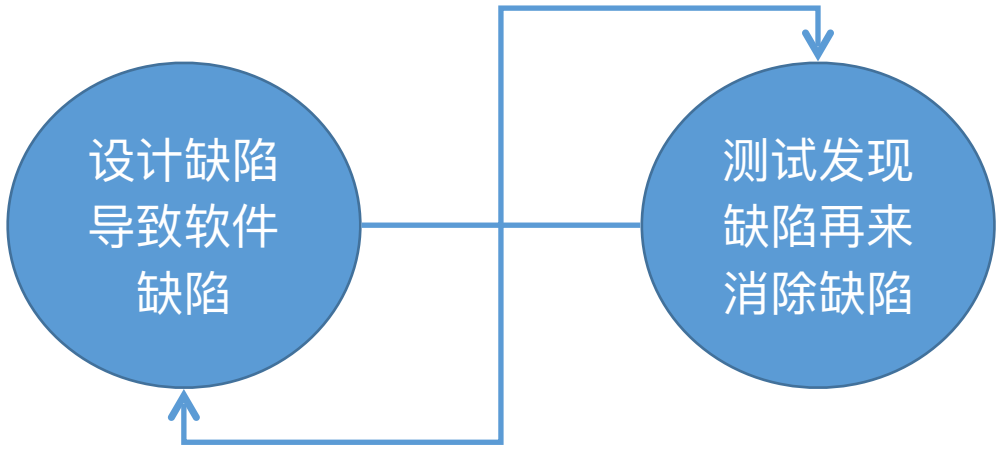
静态检查

- 静态检查以不执行代码的方式来检查和发现代码中的缺陷，依赖于人对于代码的理解和逻辑推理
  - 类规模大小适中，方法实现的代码量适中（比如：在50-100行之间）
  - 循环变量不允许在循环体内进行处理和修改
  - 与循环无关的计算放在循环体外
  - 不在循环体内对常数求值
  - 清楚无效的可执行代码
  - 明确数值的上/下溢
  - 任何调用都测试被调用者返回的状态

不因为程序的效率而牺牲可读性，除非对效率有明确要求

单元测试

- 对课程作业级别的代码，单元测试是非常有效的办法
  - 语句覆盖度100%
  - 分支覆盖度100%
  - 错误处理路径覆盖100%
  - 不只是功能，也要关注性能
  - 使用额定数据、异常数据、边界数据
    - 了解等价类的概念
- 每一个模块（类）是独立的可理解的
- 系统需求的修改应该只涉及单个模块
- 模块内的修改尽可能不影响其他模块



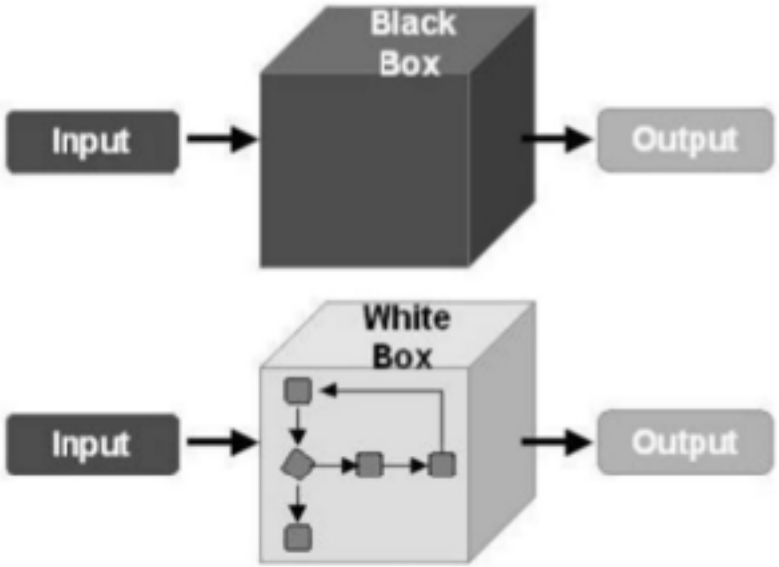
类的独立性是否有保障

测试重点

- 封装性
  - 类的数据成员尽可能不被外部类直接调用
  - 当改变数据成员的内部存储结构时，不应影响现有的外部接口
- 函数接口调用准确
  - 实参和形参数目相等
  - 实参和形参属性匹配
  - 实参和形参单位一致（1999年，美国火星气候轨道器失败，因为实参和形参使用了不同的英制和公制单位）
  - 实参和形参次序是否一致
  - 不同模块的全局变量定义是否一致
- 继承的测试
  - 子类修改了成员函数，涉及到的相关的函数调用部分都需要重点测试

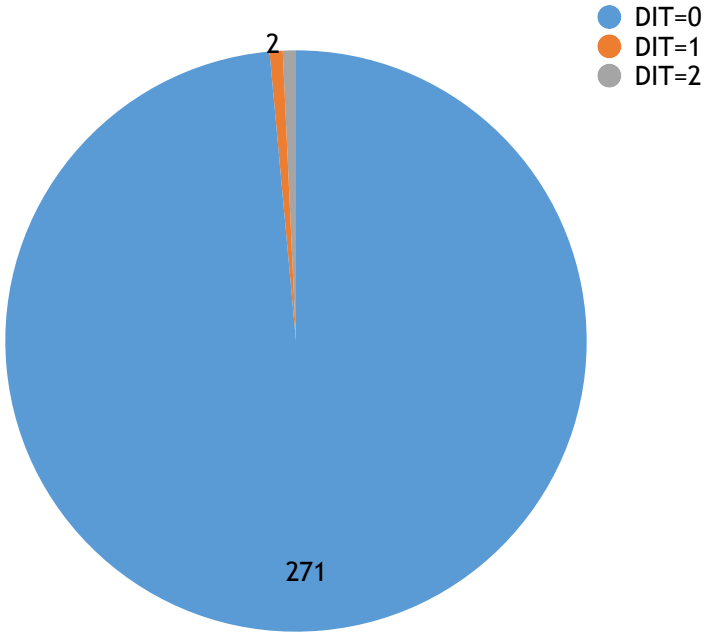
如何分析和度量代码质量

- 黑盒分析
  - 功能测试 ✔ （弱测、中测、强测和互测）
  - 性能测试 ✔ （性能分）
- 白盒分析
  - 人工阅读、代码审查(Code Review) ✔ （互测）
  - 代码静态分析 ✔ （风格分）
    - 代码风格检查
    - 代码静态特征
  - 代码动态分析
    - 代码行为特征

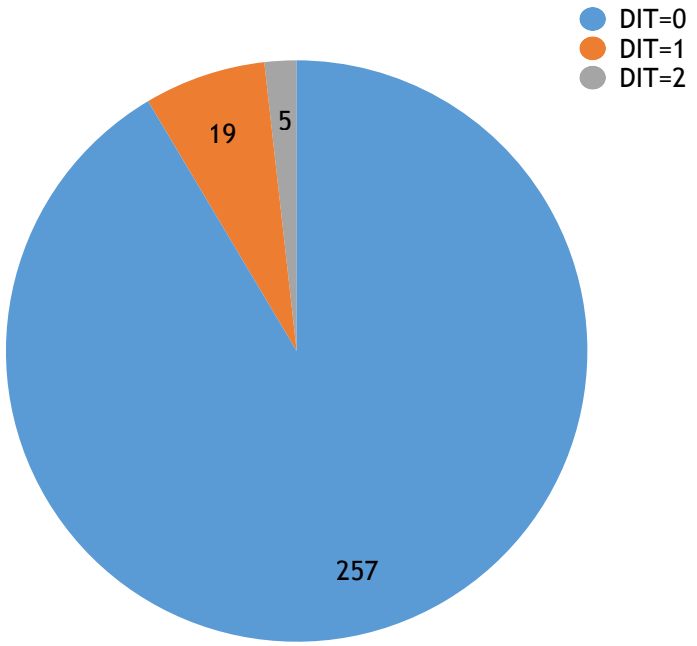


代码静态特征：面向对象设计相关

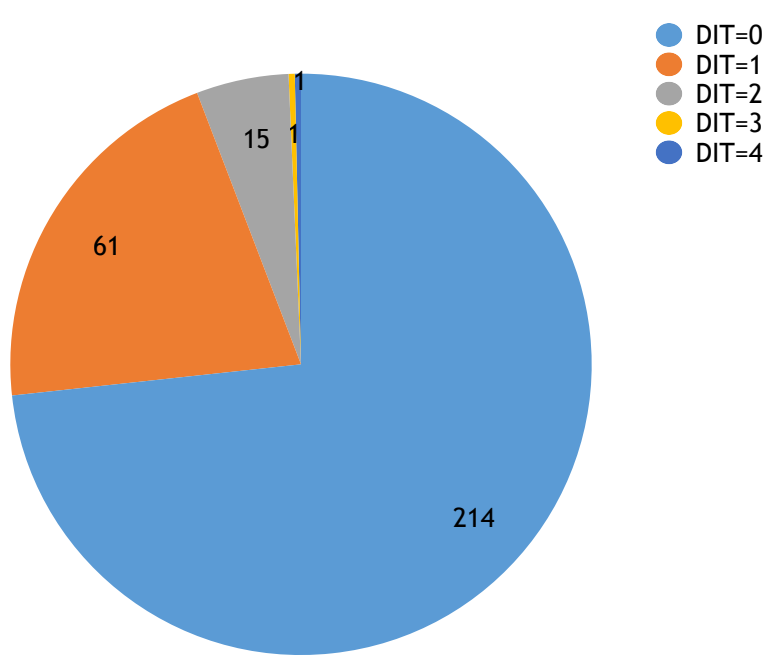
- 继承和多态
  - 特征1：类继承和接口实现的深度(Depth of Inheritance Tree, DIT)。度量是否有效使用类继承和接口实现。



作业1



作业2



作业3

代码静态特征：面向对象设计相关

- 不必要的类
  - 特征2： 只有极少数属性、且没有方法的类 (Unnecessary Abstraction, UA)。缺少数据上的行为抽象。

- 命令式的类
  - 特征3：只含有一个Public方法的类(Imperative Abstraction, IA)。一般只包含一个构造函数或一个静态方法。

- 循环依赖的类
  - 特征4：两个或以上的类之间相互依赖(Cyclic-Dependency, CD)。一般在逻辑划分不清晰的情况下，多各类之间功能交叉依赖。

```
class A {
    long  a;
    long  b;
}
```

Unnecessary Abstraction

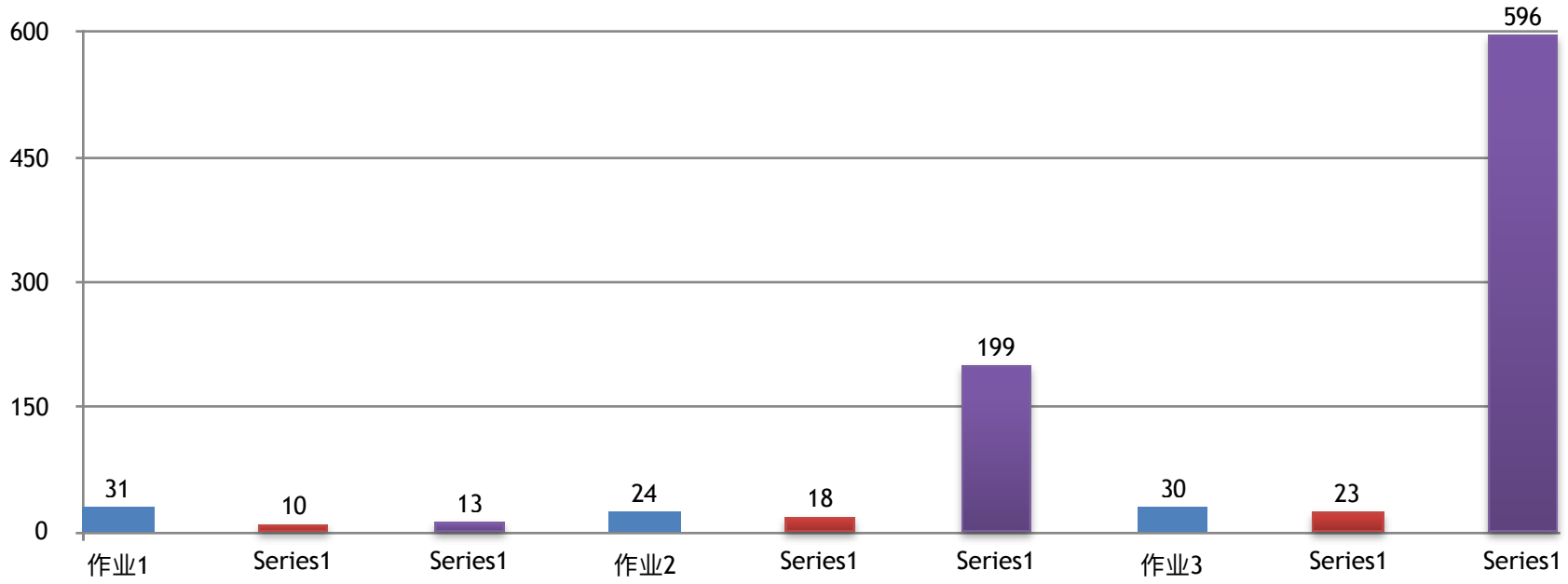
```
public class Poly {
    private BigInteger …;
    private BigInteger …;

    public Poly (String str) {
            …
    }
}
```

Imperative Abstraction

```
public class A  extends B{
    …
}
        public class B {
            public B (String str) {
                    new A();
            }
        }
```

Cyclic-Dependency
(Tree is good, Graph is bad)

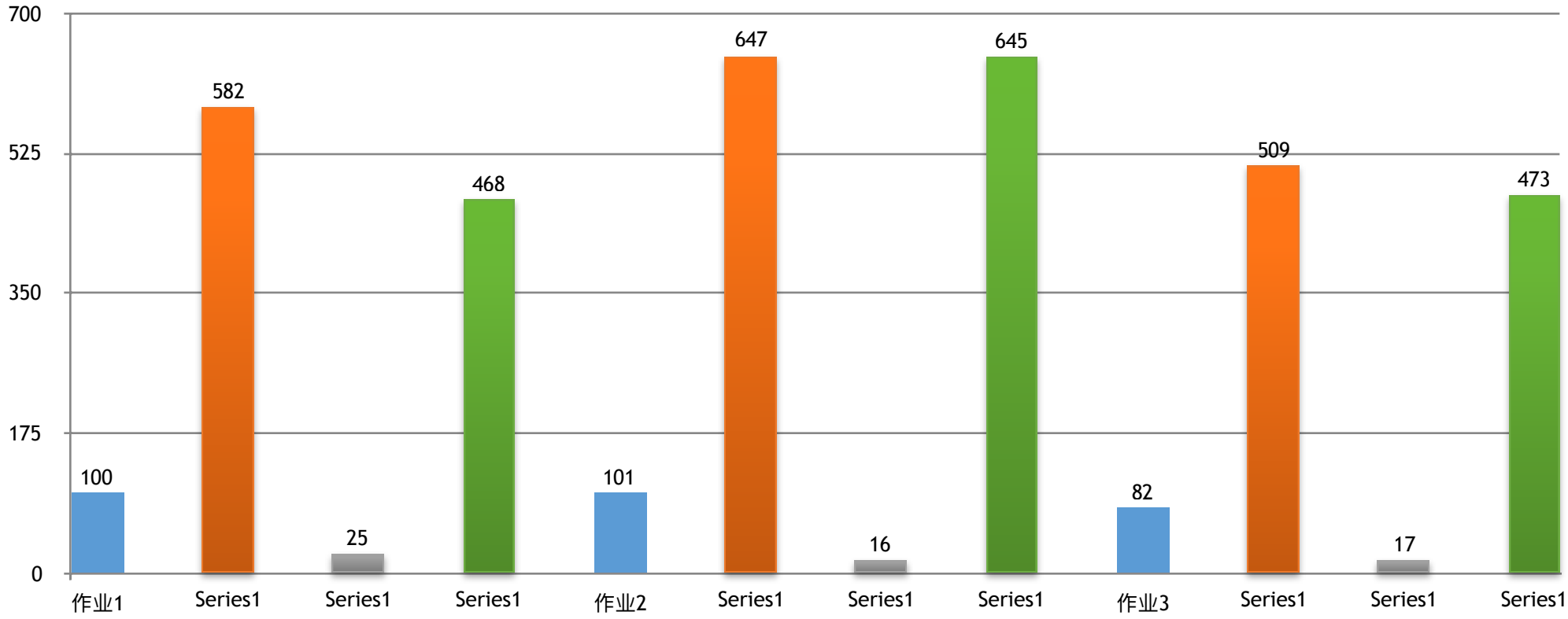


代码静态特征：控制类和方法的长度

- 抽象原则：高内聚、低耦合

- 单个类功能集中
  - 特征5： 类的行数(Lines of code per class, LOCC)。

- 单个方法功能集中
  - 特征6：方法的行数(Lines of code per method, LOCM) 。



在平均情况下，类和方法的行数都不大，分别是100左右和10~30之间
The simple is better ^^

代码静态特征：潜在的Bug区域

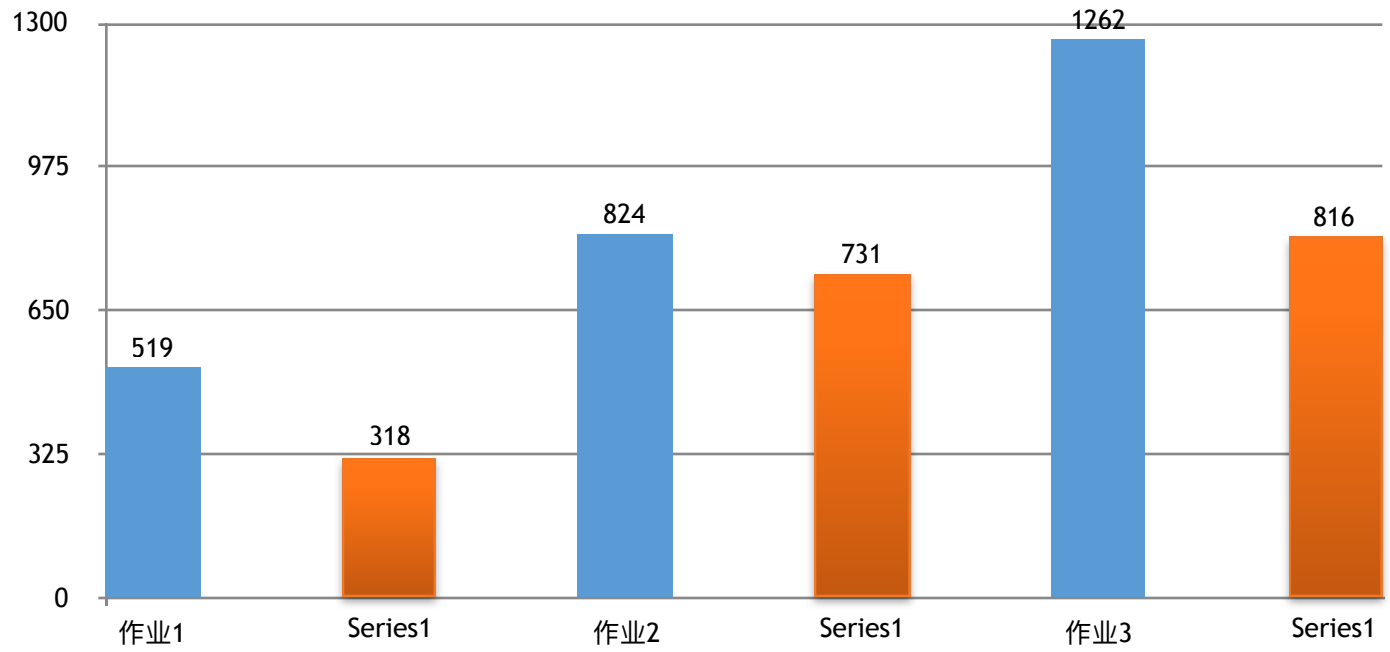
- 复杂方法：存在过多控制流结构以及嵌套使用
  - While, for, if, switch, do-while等多重嵌套使用
  - 控制流复杂，不易理解和调试
  - 指标7：某个方法含有控制流语句个数> M
- 复杂条件：条件嵌套使用
  - 与、或、非
  - 逻辑计算复杂
  - 指标8：某个条件表达式含有子表达式的个数> N

```
while (a) {
        find = false;
        for (int i = 0; i < poly.count && !a; i++) {
            for (int j = 0; j < poly.count && !a; j++) {
                if (i != j) {
                    a = …
                    if (!a) {
                        a = …
                        if (!a) {
                            a=…
                             if (!a) {
                                a=…
                            }
                        }
                    }
                }
            }
        }
    }
```

复杂方法

```
if (!val.equals(a) && (!key.getX ().equals(a)
                || (!key.getSin ().equals(a))
                || (!key.getCos ().equals(a)))) {
            …;
        }
```

复杂条件



M=8, N=3

Lessons

- 将问题的复杂语义进行二层分解
  - 类层次
  - 类行为层次（即方法）
- 类之间：有效进行类和继承/接口抽象
- 类内部：行为解耦
  - 控制每个方法的代码行数、有效解耦
  - 降低方法内控制结构的复杂度
  - 降低条件表达式的复杂度

开源静态分析工具
DesigniteJava
(https://github.com/tushartushar/DesigniteJava)

What is more

- 其他静态特征
  - 某个类引用其他类的次数(Fan-out)
  - 类被其他类引用的次数(Fan-In)
  - 单条语句过长(Long Statement)
  - 方法参数列表过长(Long Parameter List)
  - 默认值未设置(Missing default)

- 代码动态特征
  - 代码性能相关的特征
  - 并发行为相关的特征
  - 内存访问相关的特征

关于代码风格——先从一个传说讲起

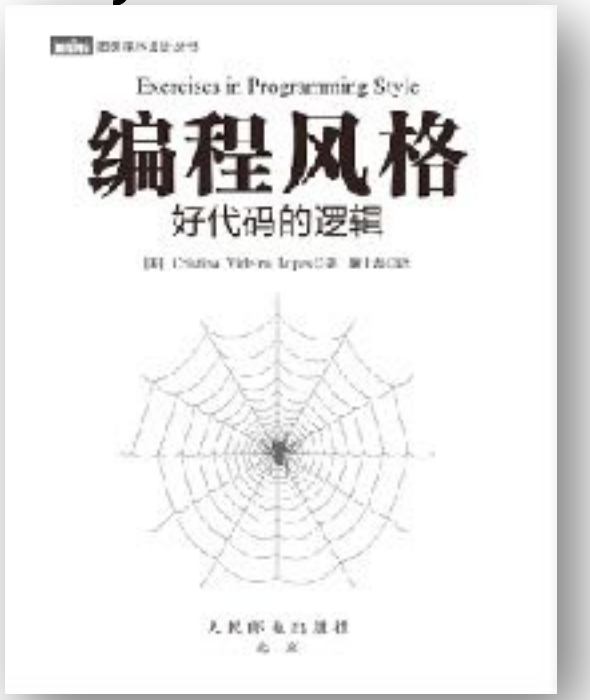
- 这是一个发生在美国一家名为WTS Paradigm的企业资源规划软件开发商办公楼里的故事
- 程序员Anthony Tong愤怒的拿出一把半自动手枪，枪杀4名同事
  - 事发非常突然
  - 没有任何迹象表明凶手动机
- 大家猜测最有可能的原因是同事——
  - 不写注释
  - 不遵循驼峰命名
  - 括号换行
  - 天天git push -f
  - ……

为什么要探讨代码风格

- 一个基本定律：写代码容易，读代码难
  - 有一个游戏：醒来你发现自己被随机丢在某个城市的google street view里，没有路名，没有地图，只有街景。你要自己找到机场，飞回家——读代码的体验，和玩儿这个游戏差不多
  - “如果一个维护者不再继续维护你的代码，很可能他有想杀了你的冲动”
- 在互测过程中，读到别人的代码是什么感受?
  - 这是什么……太厉害了 VS 这是什么……太弱渣了
  - 大牛和我做朋友吧 VS 杂碎怎么混进来的
  - 居然能这么写？ VS 居然能这么写！
  - 我勒个去这个怎么做到的？ VS 我勒个去这个烂到这样我也是服了！
- 你想成为哪个?

好的代码风格并不是唯一的

- 20世纪40年代，法国作家雷蒙格诺创作了《Exercises in Style》
  - 一个三句话的故事，用99种不同的风格讲出来
  - 反叙法、隐喻法、非人类视角……
- 代码风格：在特定的约束条件下完成编程的结果
  - 既可以来自于外部，也可以是自己强加的
  - 既可以是环境的挑战，也可以是人为的限制
  - 既可以源于经验和可测量数据，也可以仅是个人喜好
- 不同的约束，可以衍生层出不穷，风格迥异的代码
  - 代码和风格互相配合，不能一分为二
  - 基于约束条件的代码风格，是将编程实践中创造性工作统一在一起的完美模型



风格千万条，啥是第一条?

- 严谨的代码风格并无绝对好坏之分
- 不怕代码跑不动，就怕风格不一样
  - 4空格缩进遇到2空格缩进
  - 大括号换行遇到大括号不换行
  - 横杆命名遇到驼峰命名
  - 单行结构体加大括号遇到不加大括号
  - ……
- 选择易于理解并统一的代码风格
  - 建立良好的环境，减少阅读成本
  - 增强团队协作，需要点滴积累

狭义的代码风格

- 程序猿长期以来养成的一些编写代码的习惯
- 格式规范
  - 换行、缩进、长句断开……
- 命名约定
  - 常量命名、变量命名、包命名、类和接口命名、方法命名……
- 文档约定
  - 类和接口描述、方法描述……
- 其他约定

广义的代码风格

- 坚持美观，灵活对待，符合编程的一般原则
  - 避免重复原则：一旦重复某个语句或概念，可以进行抽象
  - 抽象原则：与“避免重复”相关，一个功能只出现在一个位置
  - 简单原则：简单的代码占用资源少，漏洞少，易于修改
  - 避免创建不必要的代码：除非需要，否则不创建新功能
  - 尽可能做最简单的事：简化每一个类的功能，保持简单的路径
  - 别让我思考：代码要易于理解，不要让别人敬而远之
  - 开闭原则：可以基于你的代码进行拓展，但不能修改你的代码
  - 代码维护：本人和他人都能够容易维护
  - 最小惊讶原则：尊崇约束，减少给别人的惊喜或惊吓

广义的代码风格

- 坚持美观，灵活对待，符合编程的一般原则
  - 单一责任原则：一段代码保证只有单一的明确的任务
  - 低耦合原则：一段代码应减少对其他区域代码的依赖关系
  - 最大限度凝聚原则：相近功能的代码尽量放在同一个部分
  - 隐藏实现细节：当功能发生变化时，尽可能降低对其他组件的影响
  - 迪米特法则：代码只和其有直接关系的部分相连
  - 避免过早优化：优化前必须设计好，并用数据证明性能确实优化了
  - 代码重用原则：能重用的代码不用额外开发
  - 关注点分离：不同领域功能，应由不同的代码和最小重叠模块组成
  - 拥抱改变：积极面对变化，使代码易于重构和扩展

代码风格的必要性

- 狭义的代码风格如同一身得体的打扮，能够给人留下第一印象
- 广义的代码风格体现能够写出“专业代码”的专业态度
- 腾讯、华为等各大企业都对代码风格进行了明文规定
  - 程序板式、注释、标识符命名等基本约定
  - 程序可读性、变量及结构体使用、可测性、可维护性等编程约定
  - 代码编辑、编译、审查等行为约定
  - 提供编码模板：可读性至上，遵循正确约定
- 专业能力的提高，伴随着代码风格的成熟
- 专业能力提升代码风格，代码风格体现专业能力

作业

- 总结性博客作业
  - 针对所讲授内容、自己发现别人的问题、3次作业被发现的bug(包括公测)、分析课所介绍的共性问题
    - 鼓励同学们互相阅读和学习，并积极点评
    - 在cnblogs上发布，一直伴随你，会有很多人看到你的博客，只要精彩，定有很多转发。
  - (1)基于度量来分析自己的程序结构
    - 度量类的属性个数、方法个数、每个方法规模、每个方法的控制分支数目、类总代码规模
    - 计算经典的OO度量(可使用工具)，分析类的内聚和相互间的耦合情况
    - 画出自己作业的类图，并自我点评优点和缺点
      - 使用UML工具，网上有很多免费资源

http://fileadmin.cs.lth.se/cs/education/edan70/CompilerProjects/2014/Reports/TervalampiOlssonLacerda-OOMetrics.pdf

作业

- (2)分析自己程序的bug
  - 分析未通过的公测用例和被互测发现的bug：特征、问题所在的类和方法
  - 关联分析bug位置与设计结构之间的相关性
  - 从分类树角度分析程序在设计上的问题
- (3)分析自己发现别人程序bug所采用的策略
  - 列出自己所采取的测试策略及有效性，并特别指出是否结合被测程序的代码设计结构来设计测试用例
- (4)Applying Creational Pattern
  - 分析自己的三次作业，识别应用对象创建模式的机会，并给出具体的重构说明