

# 《模型对象设计与构造》

## Lec04第一单元总结

OO课程组

2022

北京航空航天大学计算机学院

# 提纲

- 单元教学目标的设置
- 单元训练的设计分析
- 作业和实验的效果分析
- 代码风格
- 博客作业

# 本单元教学目标设置

- 建立面向对象程序的认识
  - 面向对象程序的框架
  - 三个关键类：输入处理、主控、核心数据管理
- 认识对象的结构特征
  - 数据和行为两个维度
  - 序列、组合层次、递归层次、抽象层次
  - 迭代情况下的结构变化
- 理解和掌握层次化设计
  - 按照数据/行为建立抽象层次
  - 对多层次对象进行归一化管理
- 训练设计
  - 递进式的三次作业
  - 两次在线实验

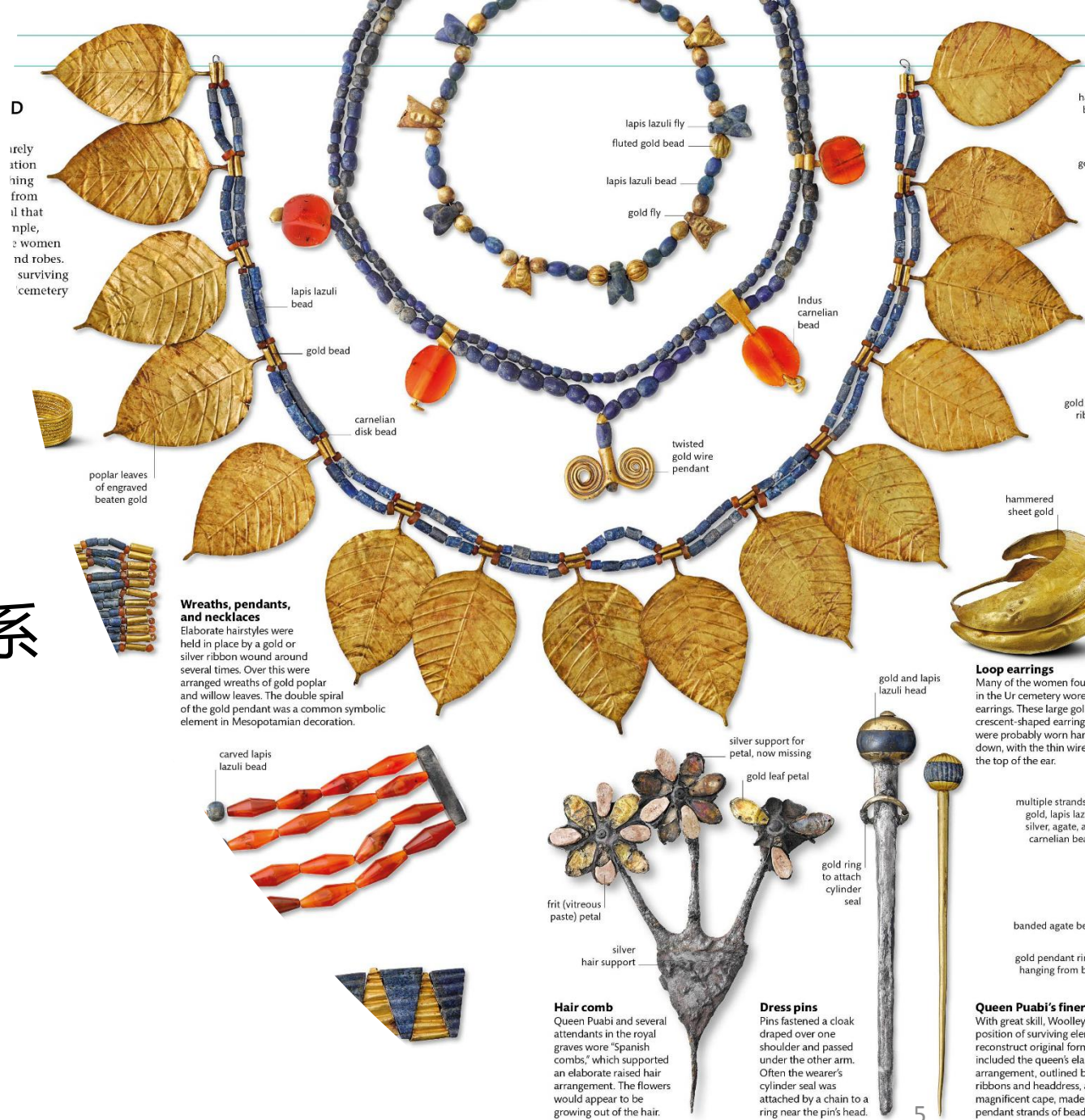
# 面向对象是一种思维方法

- 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
  - 如果对象可以在不同的主机上执行，且还能互相交互如何？
- 同一个类所实例化对象，状态管理能力相同，但却是不同个体
  - 独立存在，状态差异
  - 一个对象出错，不意味其他对象必然出错
- 所谓“面向”，即把对象(类)作为基本单位来规划程序的行为
  - 数据管理行为
  - 用户交互行为
  - 计算控制行为

对象协同行为

# 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
  - 如何管理和利用层次关系



# 面向对象是一种思维方法

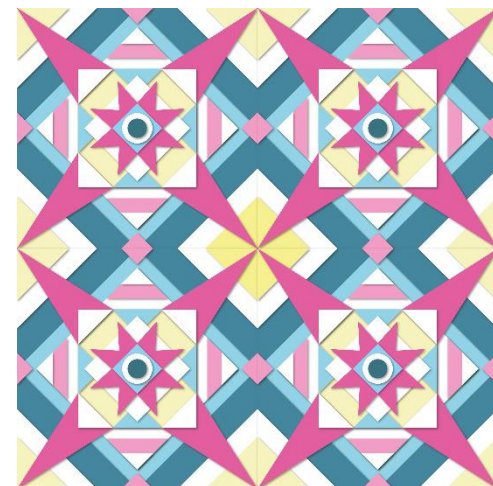
- 对象管理
  - 谁来管
    - 内部：自己构造，自己管理
    - 外部：传递工作内容(外→内，内→外)
  - 如何管
    - 个体枚举
    - 静态数组
    - 可伸缩容器
- 如何用
  - 存储数据
  - 层次代理



# 对象的创建与管理

- 本单元训练涉及多种类型的对象
  - 表达式、项、因子
  - 三角函数、求和函数、自定义函数、函数调用
- 组合规则更是复杂，且可以递归
- 解析输入并构造对象是个重要问题
  - 字符串解析：分散 vs 统一
  - 对象构造：分散 vs 统一
- 可以使用设计模式来重构

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



## Design Patterns

Elements of Reusable  
Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



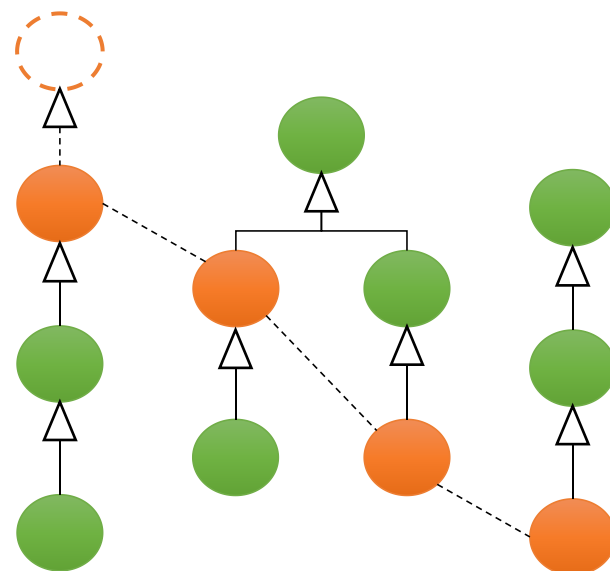
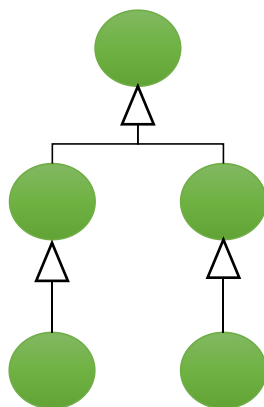
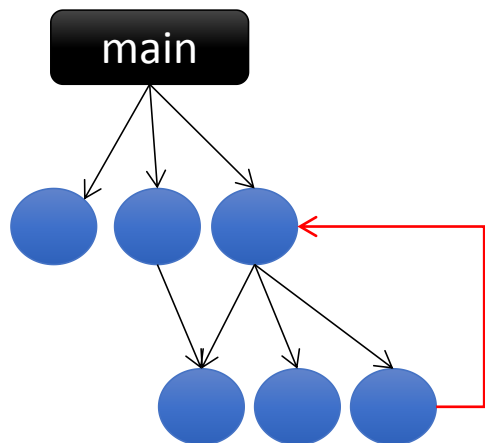
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

对象构造模式  
Singleton Pattern  
Factory Pattern  
Prototype Pattern  
Builder Pattern  
Object Pool Pattern



# 面向对象是一种思维方法

- 建立层次关系
  - 按照part-of关系形成的层次（组合层次）
    - 存在递归组合层次
  - 按照kind-of关系形成的层次（继承层次）
  - 按照can-do关系形成的层次（接口实现层次）
  - 三种层次关系可以叠加





# 面向对象是一种思维方法

- 层次关系管理
  - “意中人”在哪里
    - 从容器中找到
    - 对象引用带着面纱
  - 归一化管理
    - 通过上层抽象来无差别的引用和访问
    - 良好设计的结果
  - 协作是硬道理
    - 水平协同、跨层协同
    - 模式化(赋之以角色)：Factory、Observer(publish/subscribe)、Visitor...



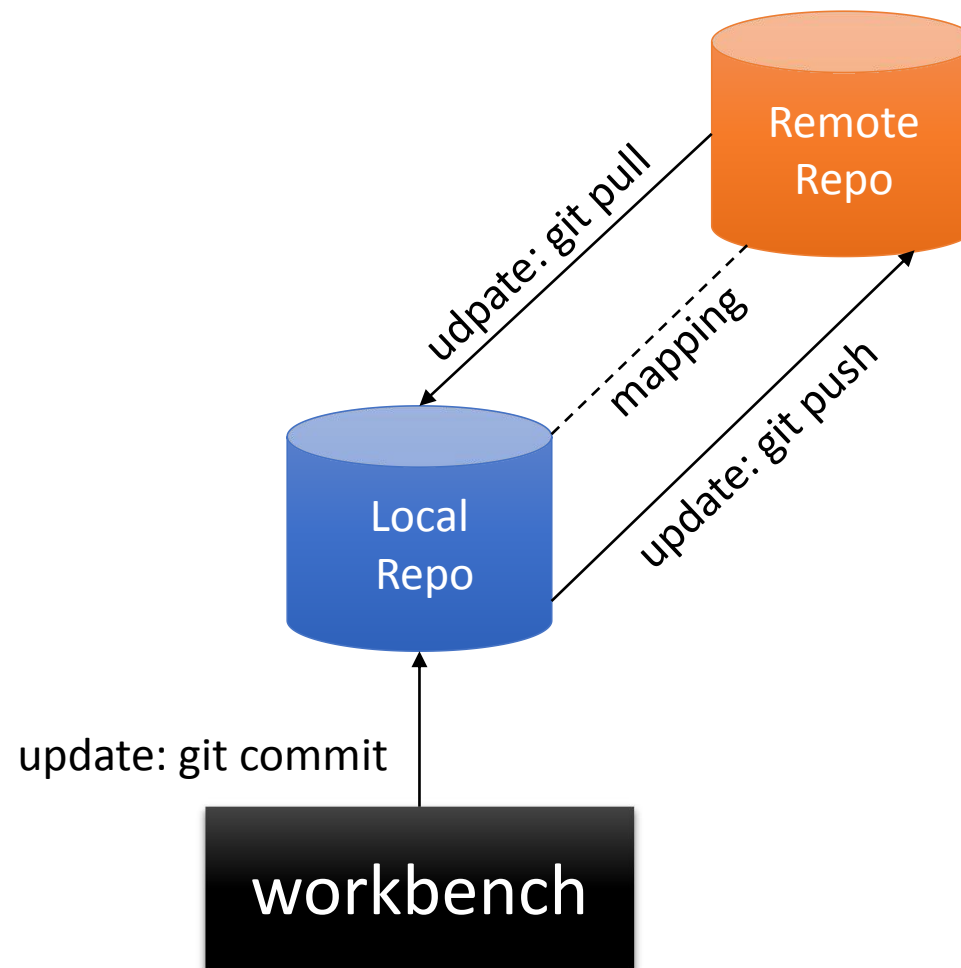
几乎大部分design pattern都涉及到综合使用  
继承层次和接口实现层次

# 本单元作业的设计意图

- 第一次作业的设计目标
  - 使用类来管理数据
    - 多种类型的对象：表达式、项、因子（常量因子、变量因子、表达式因子）
    - 使用容器来管理诸多对象，带来合并时的遍历
  - 分工协作的行为设计
    - 展开：乘法问题
    - 化简：合并同类项
  - 数据的两种结构
    - 表示结构~逻辑结构
  - 建立程序鲁棒性概念
    - 输入处理的鲁棒性：正则表达式
    - 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：超大数的计算处理

# 本单元作业的设计意图

- 第一次作业的训练目标
  - 掌握gitlab的使用
    - 什么是仓库？
    - git commit , git push , git pull在干啥？
  - 熟悉代码风格检查
    - 为什么强调代码风格？
    - 是独门暗器吗？
  - 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



# 本单元作业的设计意图

- 第二次作业的设计目标
  - 迭代设计
    - 增加函数与函数调用
    - 函数因子
  - 预定义函数与自定义函数
    - 求和函数、三角函数
    - 函数定义与函数调用
  - 递归结构更加复杂
    - 表达式因子→表达式
    - 函数因子→函数表达式
  - 增加行为类别
    - 展开：乘法、去括号
    - 合并：同类项
    - 替换：形式替换（在逻辑结构上替换）

# 本单元作业的设计意图

- 第二次作业的训练目标

- bug修复：(pass the failed test case(s)) && !(fail any passed test case(s))

- 代码重构

- 建立组合层次关系（递归结构的表示）

- 避免大正则

- 有余力的同学可以尝试递归下降，但不是必须

- 字符串替换是一个痛点

- 本质上是变换规则问题

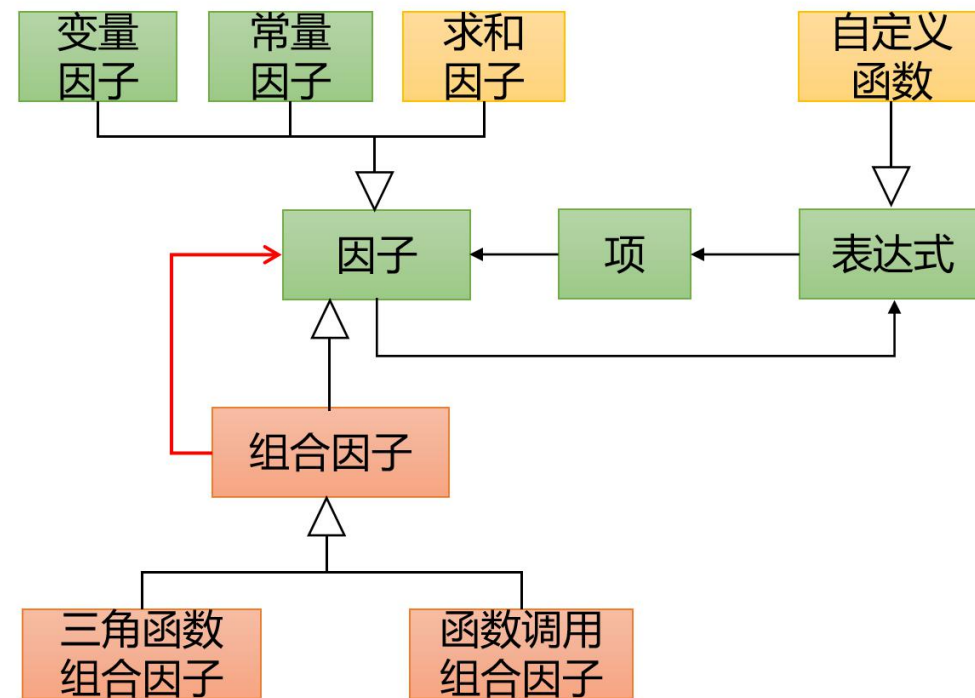
- 如何定义？

# 本单元作业的设计意图

- 第三次作业的设计目标
  - 在多种类型之间建立抽象层次
    - 有几种表达式/项/因子？
    - 进行归一化处理（引用和操作）
  - 增加新的组合规则
    - **因子嵌套**：  $f(h(x))$
    - 组合规则可递归应用
- 因子的行为抽象
  - 乘展开
  - 代入展开
  - 合并同类项
  - 解析构造或生成
- 因子的数据抽象(函数化)
  - 系数、幂
  - 实参列表
  - 函数表达式
- 多种因子函数
  - 常量函数
  - 幂函数
  - 三角函数
  - 自定义函数
  - 求和函数

# 本单元作业的设计意图

- 第三次作业的训练目标
  - 掌握和应用继承、接口和多态机制
    - 面向应用的业务需求
  - 以统一的架构来整合三次作业的功能
    - 要以迭代演化的视角来看待三次作业
  - 强调架构的意义
    - 保持程序结构的简洁
    - 灵活应对需求变化
  - 如何基于黑盒测试来定位程序Bug
    - 复杂的程序内部状态导致失败的输入和bug位置之间的逻辑联系也复杂了
    - 确保仍然能够重现失败，对输入进行化简
    - 程序中增加内部状态的输出观察





# 数据结构设计还是层次化设计？

- 不少同学通过使用List和Map容器来管理输入的表达式、项和因子
  - 使用全局的数据容器：List<List<Factor>>
  - 忽略了这些对象之间的内在抽象层次关系
  - 导致在遍历对象来进行代入替换或合并时非常复杂
- 首先建立数据抽象层次
  - 基础：函数（系数、幂）
  - 层次依据：多种不同的函数
- 然后建立基于函数的组合管理层次
  - 线性组合、乘积组合、嵌套组合
  - 组合结构依赖于**抽象**，而不是具体！
  - **每一种组合仍然是一种函数！**
- 关于优化的繁琐
  - 优化涉及到搜索和归并，可能需要使用一些**全局性的容器**
    - 针对优化的局部处理，不能处理为全局性的设计

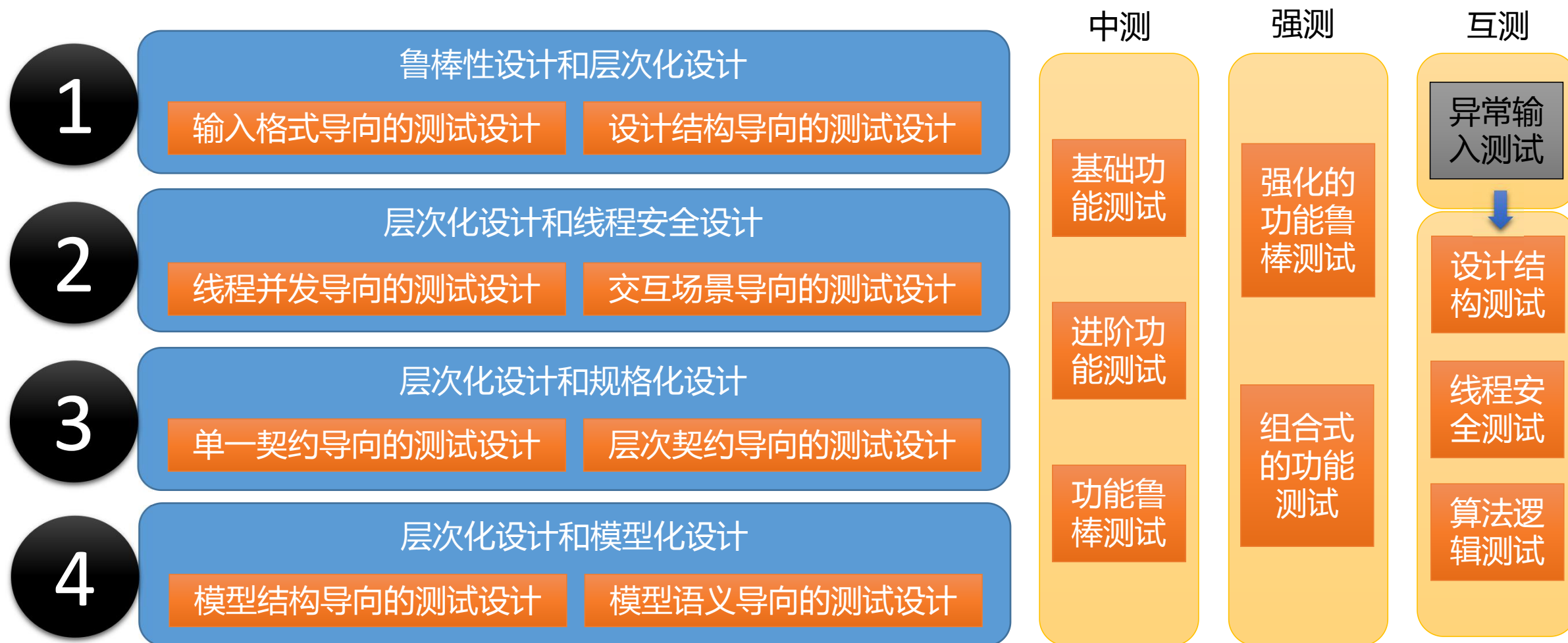
# 本单元作业测试实践

- 黑盒测试关注功能性和鲁棒性
  - 依据需求/指导书来设计测试用例
  - 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
    - 表示结构：线性序列结构
    - 逻辑结构：带递归的层次结构
- 按照指导书的文法递归向下来生成数据实例
  - 每个**对象**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
  - ...

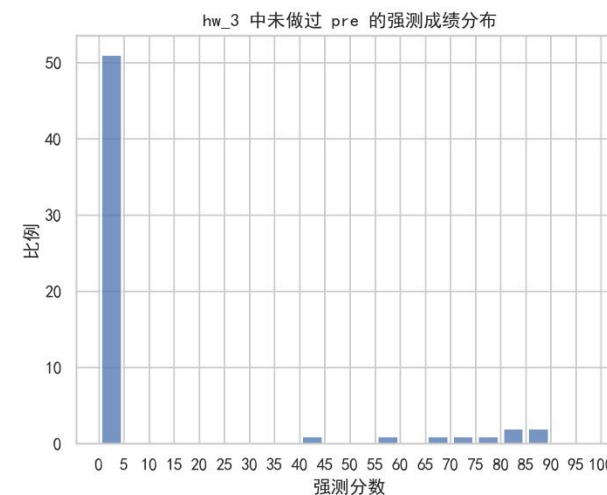
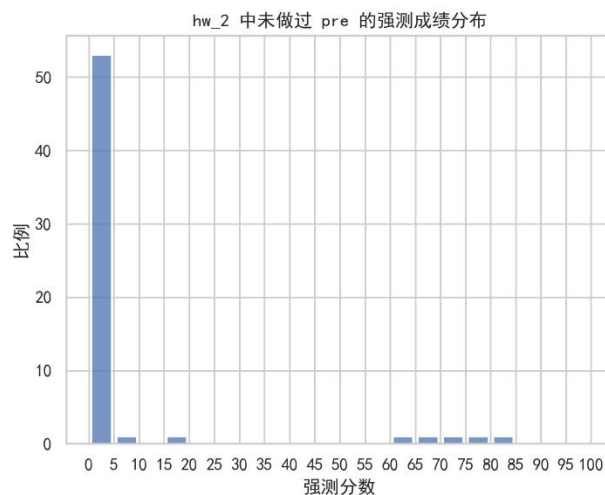
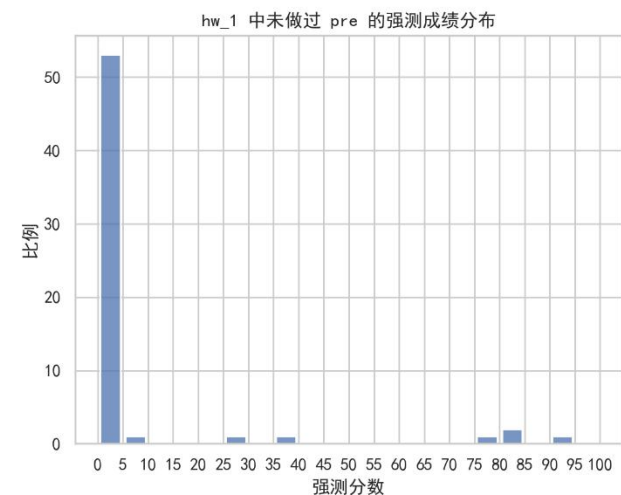
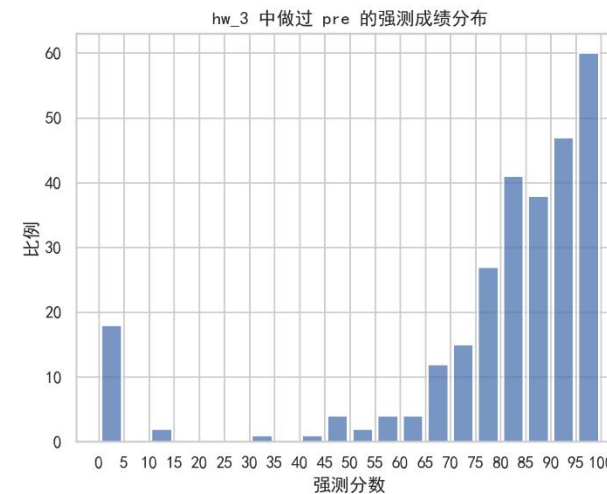
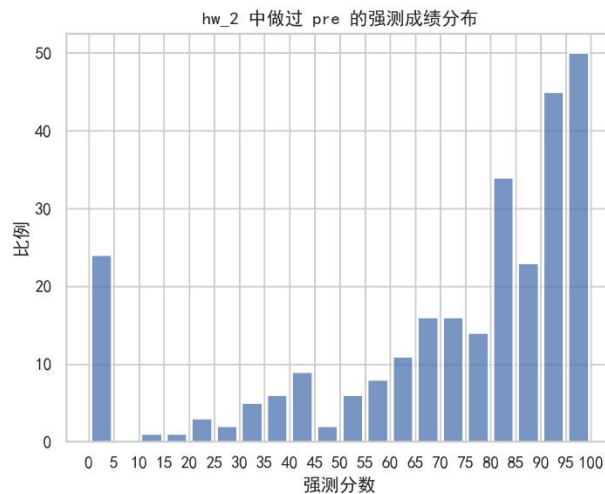
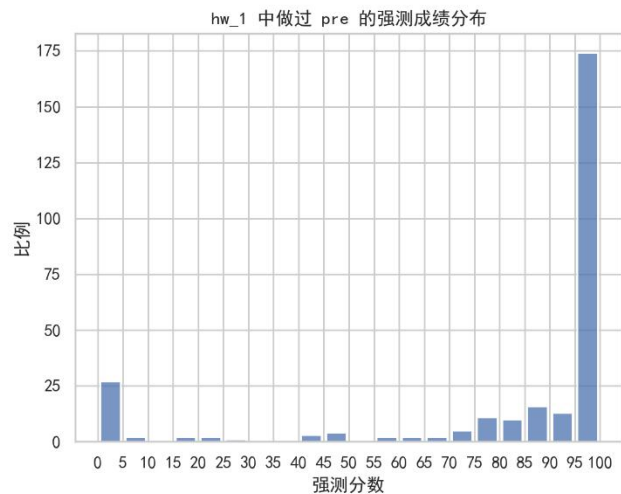
# 本单元作业的测试实践

- 测试是一面镜子
  - 反射镜：测试别人，发现自己的问题
  - 放大镜：通过测试放大程序中的局部问题，直至架构问题
  - 望远镜：结合需求迭代预测可能的脆弱点，提前进行重构或固化
- 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？

# 课程作业的设计及其测试路线图



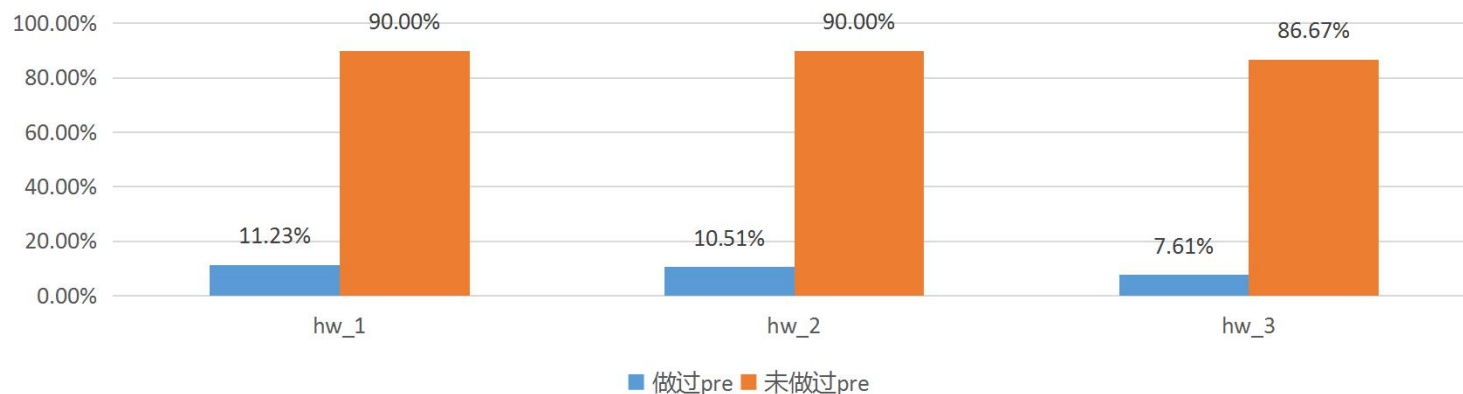
# 不能忽视的Pre价值



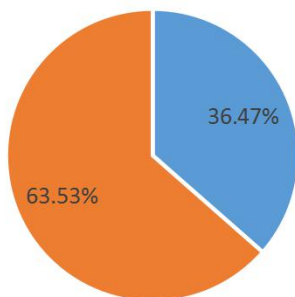
三次作业中做过pre的学生强测成绩**明显高于**未做过pre的学生

# 不能忽视的Pre价值

三次作业两类学生无效作业占比

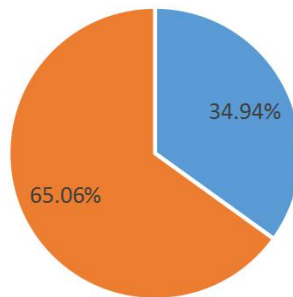


hw\_1无效作业中两类学生占比



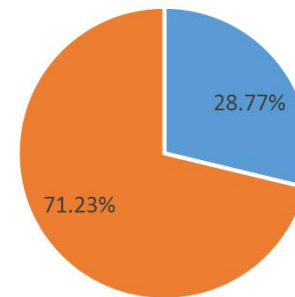
做过 pre 未做过 pre

hw\_2无效作业中两类学生占比



做过 pre 未做过 pre

hw\_3无效作业中两类学生占比

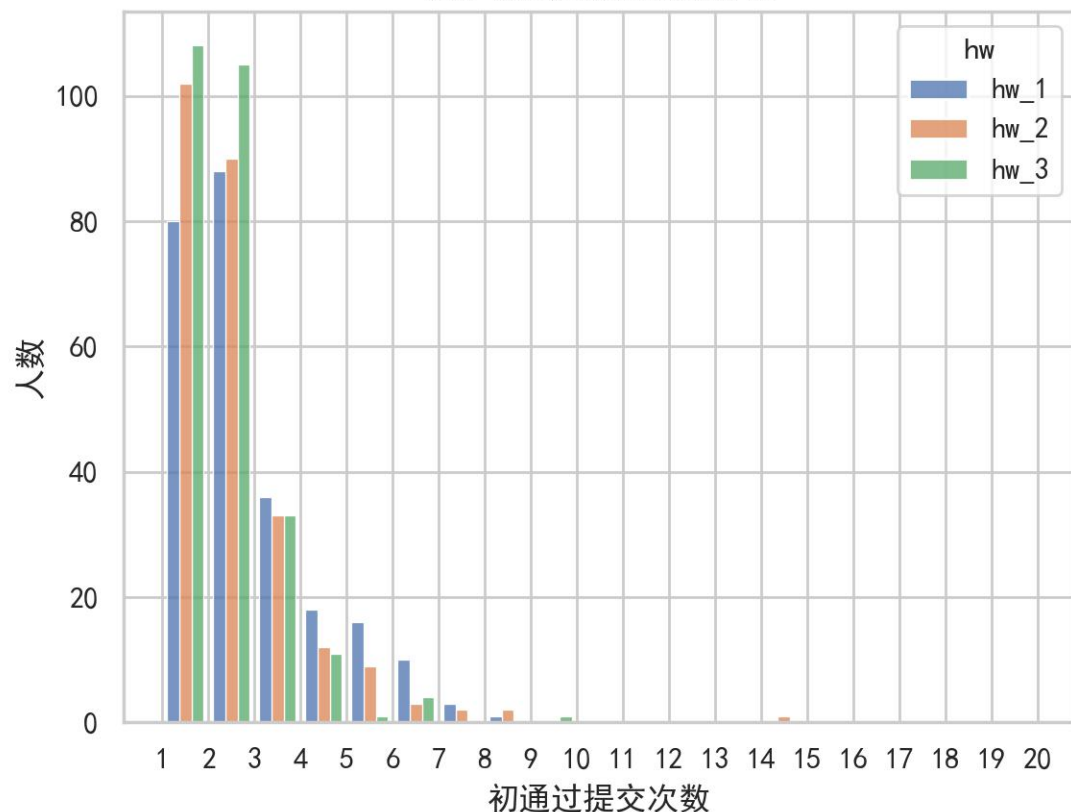


做过 pre 未做过 pre

三次作业中做过pre的学生无效作业比例**明显低于**未做过pre的学生

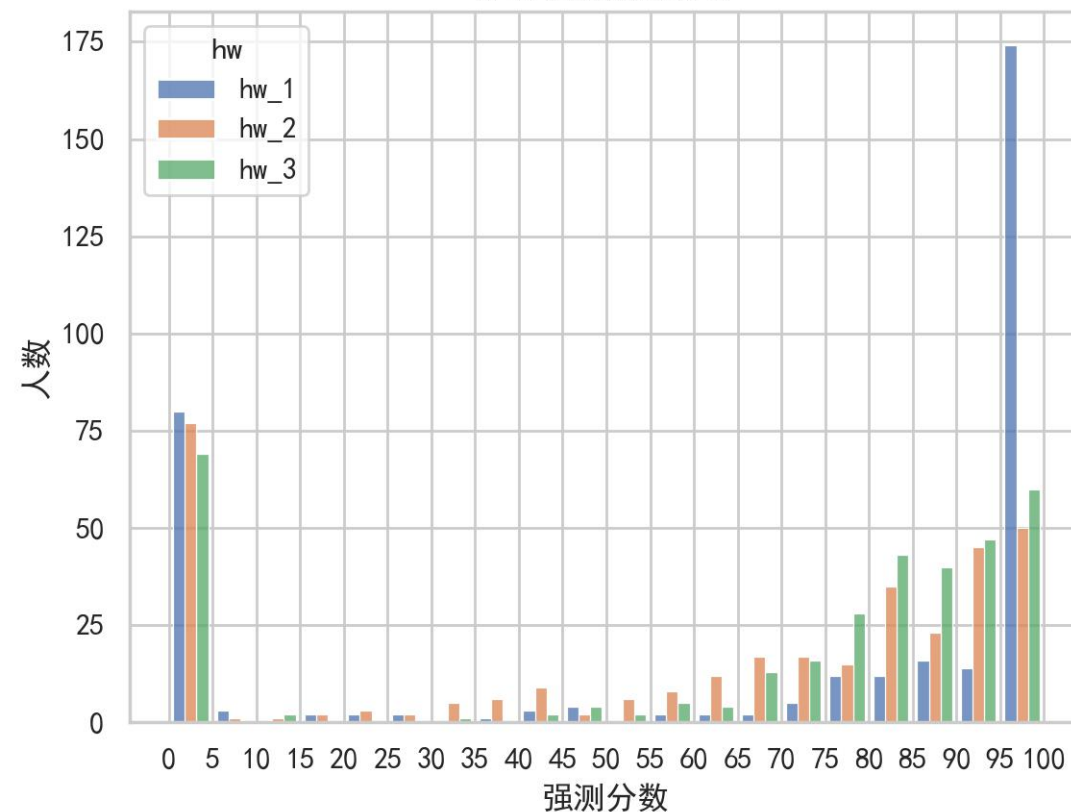
# 三次作业数据分析：公测及强测情况

整体初通过提交次数分布



公测通过提交次数逐渐减少  
逐渐掌握知识点，进步迅速

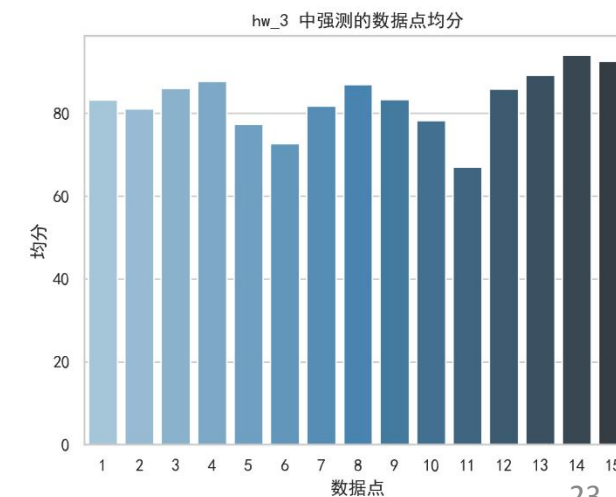
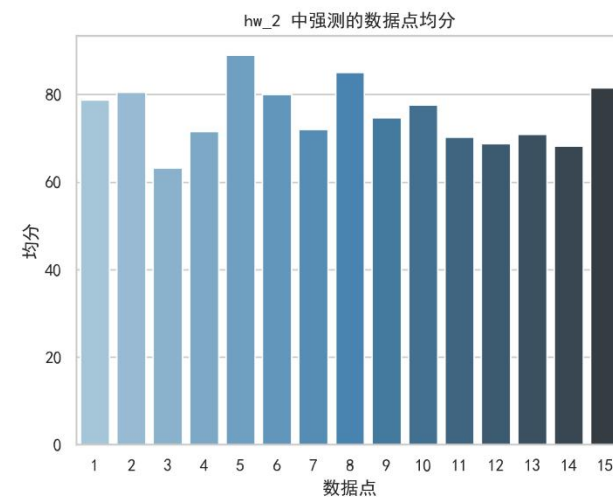
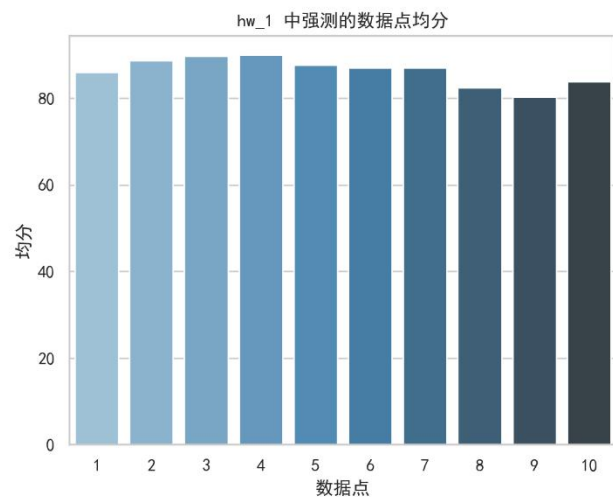
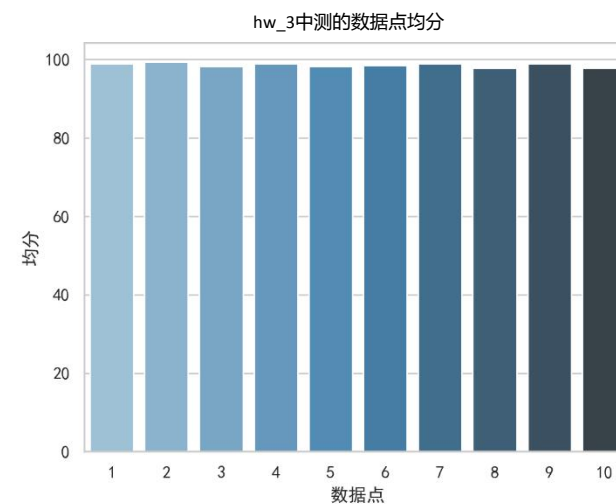
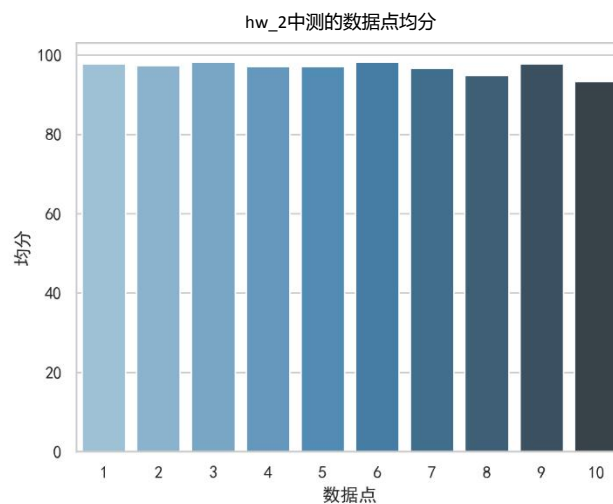
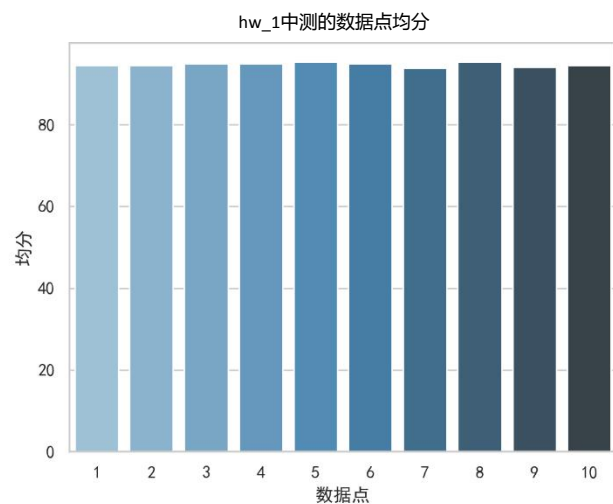
整体强测成绩分布



强测成绩占比稳定，整体难度平滑上升  
在架构设计加强引导，作业质量有所提升



# 三次作业数据分析：数据点得分情况



# 三次作业数据分析：数据点得分情况

- hw\_1：强测数据点8-10成绩偏低
  - 指数带符号
  - 合并同类项优化（按照指数合并系数，具有较长的数据与系数）， $(-1)**0$
- hw\_2：强测数据点3、4、7、11、12、14成绩偏低
  - 实参因子的多种情况覆盖测试
  - 自定义函数调用带指数
  - 求和函数表达式是表达式因子
  - 自定义函数定义形参顺序不是  $x, y, z$
  - 多次调用同一个自定义函数
  - 对求和函数的上下限以及求和表达式的覆盖性测试
  - 求和函数上下限边界情况测试
  - 自定义函数调用的乘方
- hw\_3：强测数据点5、6、11成绩偏低
  - 三角函数嵌套调用自定义函数
  - 自定义函数函数嵌套调用
  - 求和函数上下限边界情况测试

# 互测分析：测试无法证明没有Bug

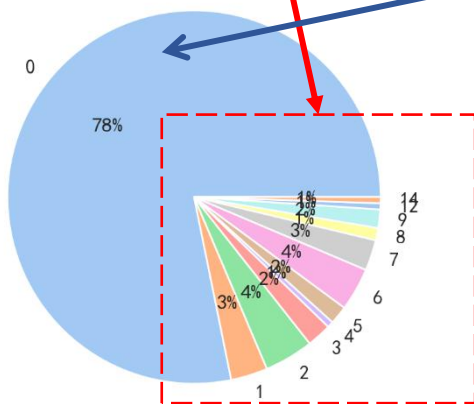
- 中、强测全部通过的情况下，互测中被Hack的Bug数量分布：

第一次作业(A Level)

- 22%仍然存在Bug
- 有人被找出14个Bug！

**强测没问题，照样有Bug！**

hw\_1 通过所有公测的 bug 数 (A 屋)

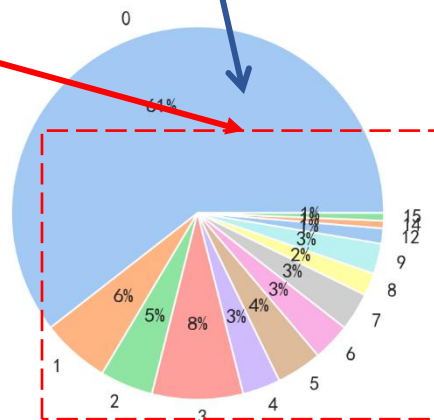


第二次作业(A Level)

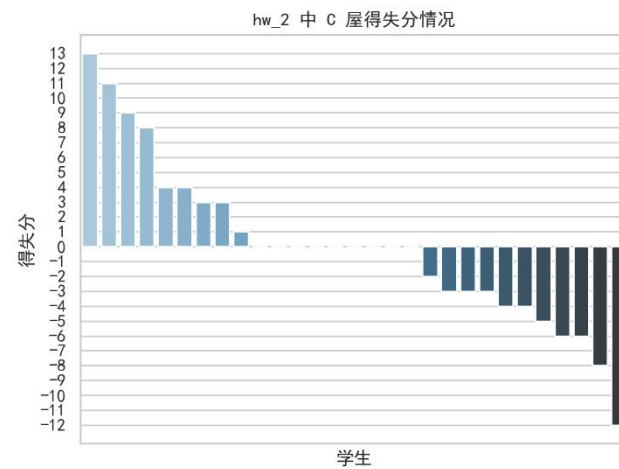
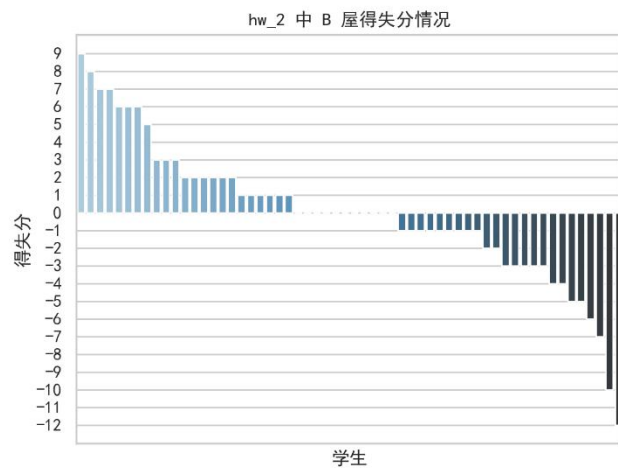
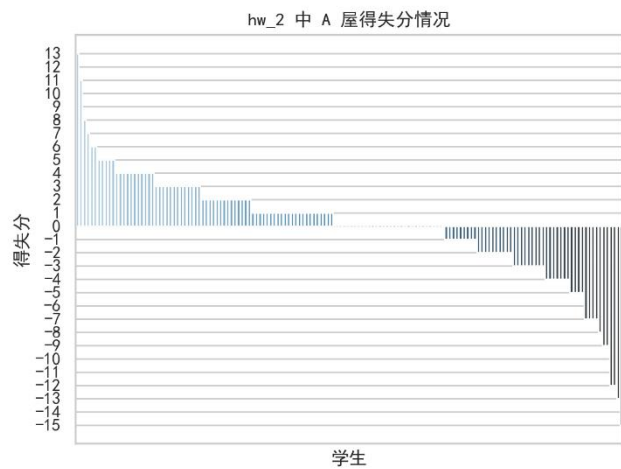
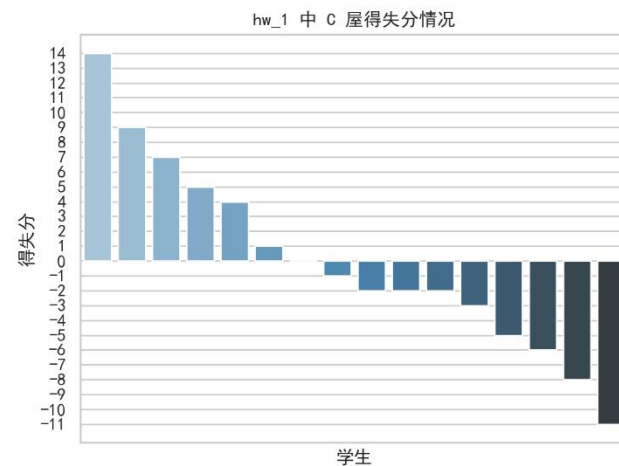
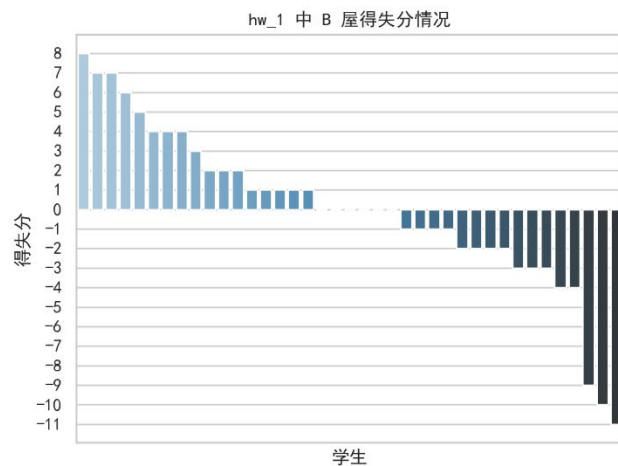
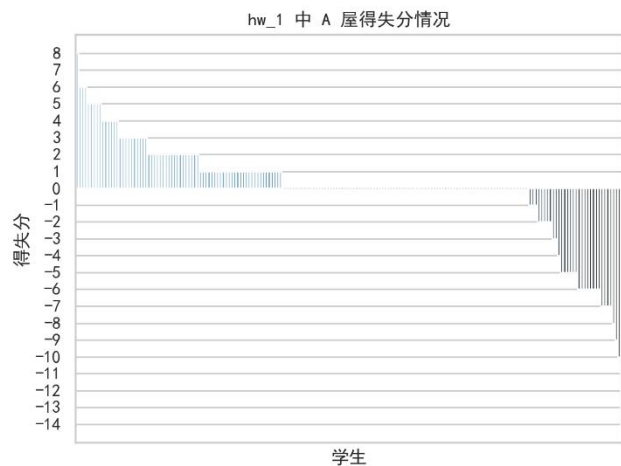
- 39%仍然被找出Bug
- 有人被找出15个Bug

**居然没被互测出Bug，我不信！**

hw\_2 通过所有公测的 bug 数 (A 屋)

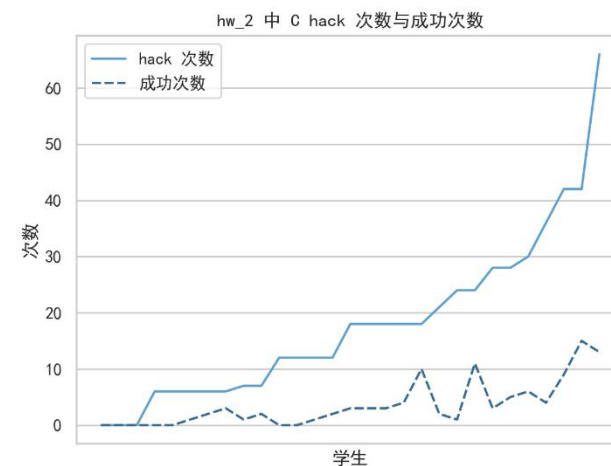
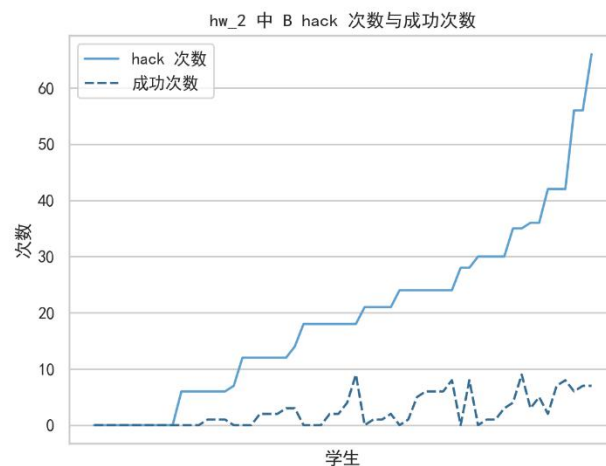
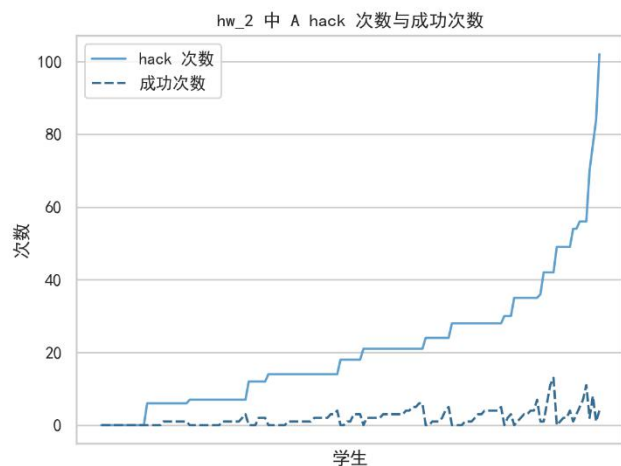
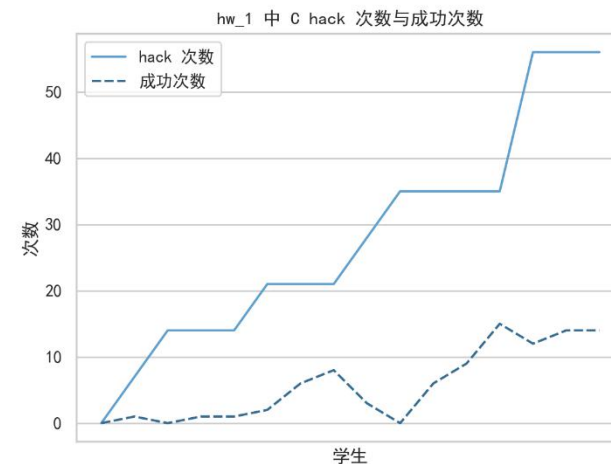
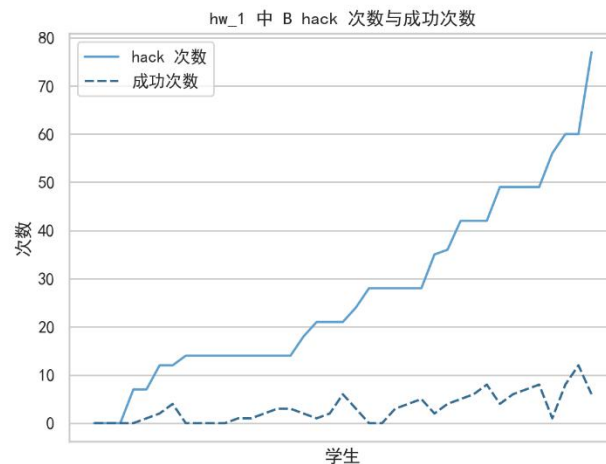
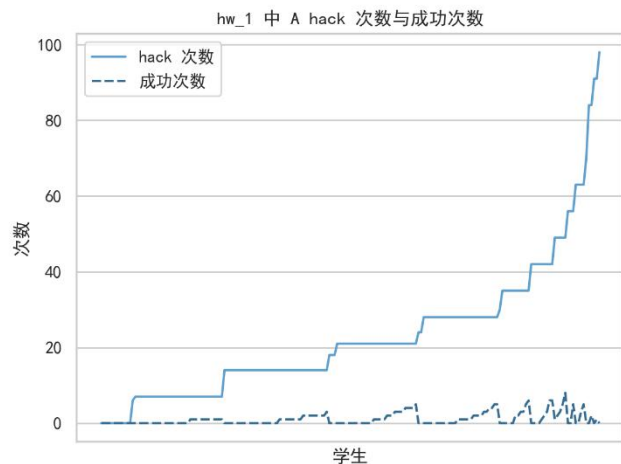


# 互测得失分分析



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**

# 互测分析：机会总是会给坚持尝试的人



互测尝试越多，发现bug机会就越大  
各级别能够发现Bug的次数与测试提交数皆有同向趋势

# 互测分析：hack成功率

作业	房间类型	总人数	hack 次数	平均 hack 次数	成功 hack 次数	hack 成功率
hw_1	A	191	4345	22.75	199	4.58%
hw_1	B	39	1055	27.05	120	11.37%
hw_1	C	16	448	28	92	20.54%
hw_2	A	153	3177	20.76	287	9.03%
hw_2	B	58	1129	19.47	139	12.31%
hw_2	C	29	523	18.03	104	19.89%

各级别互测意愿几乎相近，级别越高成功率越低，级别越低**互测补偿价值越高**

# 实验1效果分析

- 按时提交比例77.98%
- 任务一：git使用
  - 训练目标：熟悉 OO 课程平台与代码仓库的基本使用方法
  - 训练要点：git 的配置，git 基本命令的使用，git 冲突处理
- 任务二：表达式解析与计算
  - 训练目标：提供一个将表达式解析为二叉树的主干代码，理解代码逻辑并填充代码剩余部分——**可转化为第一次作业表达式的解析器**
  - 训练要点：Java 基础语法，对已给代码的解析逻辑的理解
  - Mul.java：需要根据Add.java和Sub.java进行填空，完成计算方法
  - Parser.java：
    - 需要类比完整给出的findAddOrSub方法去填findMul的缺失部分，理解建立二叉树递归的边界线划分
    - 需要会写无符号数的正则表达式
    - 需要分类讨论叶子结点的类型并完成相应的代码逻辑

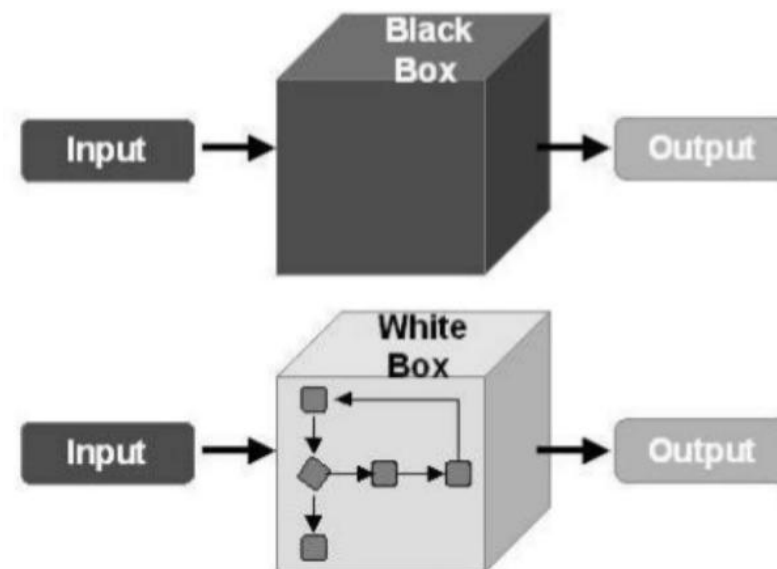


# 实验2效果分析

- 按时提交比例86.90%
- 训练目标：实现避免输出时出现不必要括号包裹因子情形的方法
- 训练要点：
  - 理解已给代码核心计算部分的层次化的架构
  - 通过代码中给出的注释提示编写 TODO 部分
  - 把实验中的技巧应用在作业中
- Power.java、Number.java，根据注释中的提示，调用自身组分的化简方法或者直接返回
- Term.java，递归调用自身组分Factor的化简方法以组成新的Term
- Expr.java
  - 需要编写处理只有一项时的逻辑，以防止产生形如  $((x))$  的多余括号
  - 需要编写处理多项时的逻辑，以防止出现形如  $(a + b) + c$  的多余括号

# 如何分析和度量代码质量

- 黑盒分析
  - 功能测试 ✓ ( 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
    - 代码风格检查 ✓ ( 风格分 )
    - 代码静态特征
  - 代码动态分析
    - 代码行为特征



# 测试的基本原则

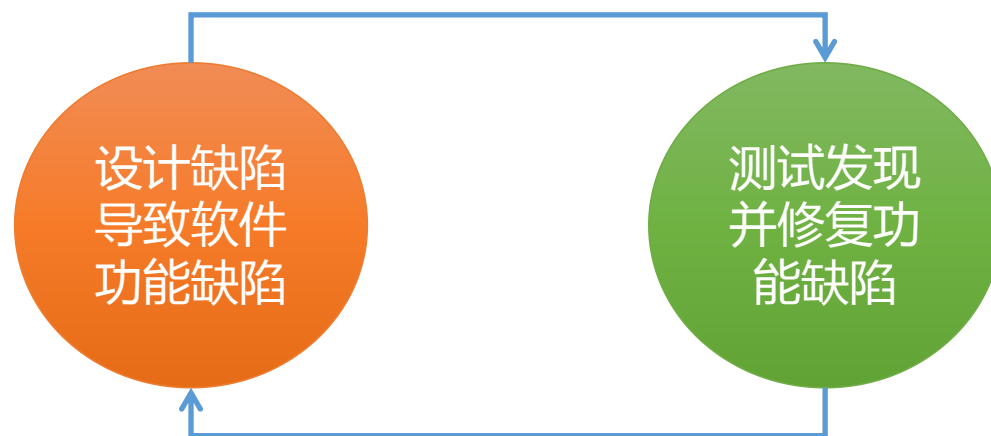
- 独立视角，不能把自己限定成设计和开发人员，怎么用就怎么测
- 尽早介入，在需求确定时就开始，设计之初就思考怎么测试每个类
  - TDD：Test Driven Development
  - 规划输入的划分与组合
- 可靠追溯，一旦测试发现错误，需要追溯相应的代码和数据
- 确保复现，以规范方式复现问题
  - 逐步化简测试输入，在化简过程中获得了bug位置的理解
- 有序开展，测试输入的复杂度和规模由小及大
- 测不全是方法学上的缺陷，无法穷举所有可能
  - 要控制复杂度。软件模块越复杂，测不全带来的问题越严重

# 测试的重要性的和局限性

- 是重要的质量保证手段
  - 内部质量决定外部质量
  - 软件质量是设计出来的
- 提高设计的可测性
  - 降低逻辑的复杂度
  - 提供中间状态的可观察性
  - 提供主动防御能力，提前发现数据问题
- 测不全问题始终存在
  - 存在潜在缺陷，核心程序需要进行严格验证（如机载、弹载和箭载系统等）
  - 验证手段：规格化+模型化
- 测试需要考虑用户的使用逻辑
  - 基于用户行为习惯的测试，互联网软件测试广泛采用，beta测试

# 单元测试

- 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
- 可以尝试使用“Junit+执行脚本”手段来部署自动化的单元测试能力
- 单元测试是TDD的核心手段



# 静态检查

- 静态检查(不让计算机执行代码)也是一种重要的验证手段
  - 人们积累了不少代码错误模式
  - 通过检查代码是否出现某些错误模式来进行验证
  - 通过检查代码的逻辑来解决测试难以覆盖的问题
- 常见的检查要点
  - 调用是否对返回值进行了接收和检查
  - 类库及方法的使用是否符合要求
  - 容器访问的越界保护
  - 对象拷贝是否彻底
  - 数值计算可能溢出
  - 是否在循环体内对循环变量进行修改

# 代码静态特征：面向对象设计相关

- 抽象层次结构
  - 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。
- 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，多个类之间容易出现功能交叉依赖。



# 代码静态特征：控制类和方法的长度

- 设计原则：高内聚、低耦合
- 单个类功能集中
  - 特征5：类的代码行数(Lines of code per class, LOCC)
- 单个方法功能集中
  - 特征6：方法的代码行数(Lines of code per method, LOCM)

# 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 (pos < text.length()) {
    ...
    if (text.startsWith("+", pos) || text.startsWith("-", pos)) {
        ...
    } else if (text.startsWith(...)) {
        ...
    } else if (text.startsWith(...)) {
        ...
    } else if (text.startsWith(...)) {
        ...
    } else if (text.startsWith(...)) {
        ...
    } else if (text.startsWith(...) || text.startsWith(...)) {
        ...
    } else if () {
        while (... &&...) {
            ...
        }
    }
    else {
        ...
    }
}

```

## 复杂方法

```

if (m1.find() || m2.find()
    || m3.find() || m4.find()) {
    ...
}
else if (m5.find() || m6.find()
    || m7.find() || m8.find()) {
    ...
}
else if (m9.find() || m10.find()
    || m11.find() || m12.find()) {
    ...
}
else if (m13.find() || m14.find()
    || m15.find() || m16.find()) {
    ...
}

```

## 复杂条件

# 解决办法

- 将复杂问题进行层次分解和抽象
  - 类数据层次
  - 类行为层次
- 类之间：有效协同，职责分派
- 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>)

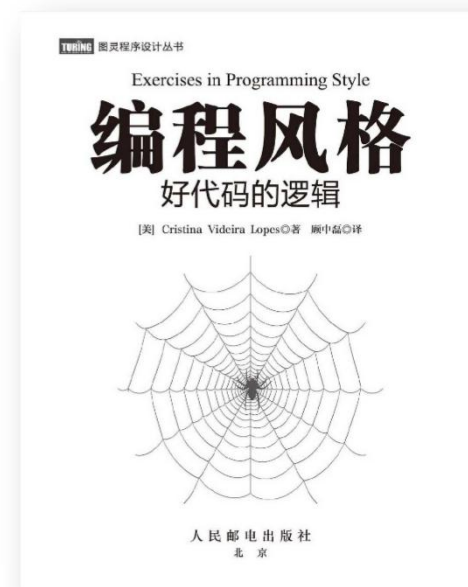
# 为什么要探讨代码风格

- 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街景地图里，没有路名，没有地图，只有街景。你要自己找到机场，飞回家
- 在互测过程中，读到别人的代码是什么感受？
  - 结构清晰、易于理解的代码：心情愉悦，学习心态→一般**不是hack目标**
  - 结构混乱、难以理解的代码：心情烦躁，发泄心态→通常**是hack目标**
- 你想成为哪个？

- 街景~局部代码
- 机场~预期想找的‘对象’
- 没有路名~看不懂代码中的命名
- 没有地图~看不懂代码中模块之间的逻辑关系

# 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



# 狭义的代码风格

- 程序员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：代码只和具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

# 面向对象思维的人生启发

- 识大局：做好上层抽象，形成整体架构
  - 学会站在更高的层次上分析问题、解决问题
- 拘小节：做出良好设计、认真处理细节
  - 具体问题具体分析，控制局部小节的影响范围，“多态”处理
- 立规矩：规范实现方式、无规矩不方圆
  - 规规矩矩做人，踏踏实实办事
- 强能力：守住功能下线，提升性能上线
  - 像优化程序一样优化自身，像修复bug一样不断自省

# 作业说明

- 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班级博客](#)上发布，一直伴随你
  - (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
    - 画出自己作业类图，并自我点评优点和缺点
      - 注意1：**不要**使用IDEA等工具“无脑”逆向生成类图
      - 注意2：**需要**配文字来解释每个类的设计考虑

# 作业说明

- (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
  - 对比分析**出现了bug的方法**和**未出现bug的方法**在**代码行**和**圈复杂度**上的差异
- (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)架构设计体验
  - 结合三次作业的迭代介绍自己的架构如何逐步成型（以及如何成为垃圾...）
- (5) 心得体会
  - 本单元学习的心得体会，真情实感

OO号充电桩ready！课程组会继续做好并强化外部资源支撑，提高同学们的能力

**作业指导、助教解答、相关技巧博客文章的推送**