

《面向对象设计与构造》

Lec12-第三单元总结

-JML规格回顾及其设计模式-

OO2024课程组
计算机学院

规格化设计回顾

- 规格是什么?
 - 类方法规格
 - 类数据规格
 - 抽象层次下的类规格（方法规格+数据规格）
- 规格有什么作用?
 - 对设计
 - 对实现
 - 对测试
- 如何写和用规格?
 - JML
 - 规格模式

规格化设计回顾

- 类是一个编程单位，用户一般把类作为一个整体来使用或重用
- 类规格定义了与用户的契约
 - 数据规格：类所管理的数据内容，及其有效性条件(invariant, constraint)
 - 方法规格：类所提供的操作，权利+义务+注意事项
- 类规格定义了开发人员必须实现的规约
 - 实现数据内容并确保始终有效
 - 使用抽象函数来确认有效性检查是否充分
 - 任意一个方法的实现都不能破坏对象的有效性
 - 不能破坏invariant和constraint
 - 任意一个方法的实现都要满足方法本身定义的规约

规格化设计回顾

- 方法规格是一个方法与其用户交互的契约
 - 契约：权利+义务+注意事项
 - 义务：用户要保证提供有效的输入以及有效的对象状态
 - 权利：用户能够获得满足确定条件的输出结果
 - 注意事项：方法执行过程中可能会修改用户对象的状态
- 方法规格是一个方法对实现者做出的规约要求
 - 规约：前置条件+后置条件+副作用
 - 前置条件：实现者可依赖的初始条件
 - 后置条件：实现者如何提供满足要求的结果
 - 副作用：不去做多余的事情

规格化设计回顾

- 类数据内容的表示
 - 只需在**内容层次**给出类所管理的数据 (model, spec_public)
 - 命名应和其所表达语义有效关联，对用户可见
- 类有效性条件
 - Invariant: 任何时刻数据内容都必须满足的约束条件
 - constraint: 任何时刻对数据内容的修改都必须满足的约束条件
- 从用户角度来看，一个设计优良的类应有“五性”
 - 完整性：应提供四类操作（构造、更新、查询和生成）
 - 紧凑性：不提供与所管理数据无关的操作行为
 - 便捷性：能够以简单的方式访问和更新所管理的数据
 - 安全性：阻止不受控访问，确保线程安全
 - 正确性：实现与契约一致

规格化设计回顾

- 抽象是OO的一个基本机制，可以形成类型层次
- 用户在使用层次化的类时倾向于进行一般化处理
 - 高层类型必须对低层类型具有概括性
 - 数据抽象的概括性
 - 子类数据抽象与父类数据抽象之间的关系（抽象函数）
 - 子类对象有效性与父类对象有效性之间的关系
 - 行为的概括性
 - 父类方法规格与子类方法规格之间的关系
 - 凡使用高层类型引用的对象都可以替换成低层类型所构建的对象，且保证程序行为不会变化（LSP）

规格化设计回顾

- 子类与父类独立，但也继承了父类的规格
- 从规格的语义角度，层次之间具有严格的逻辑关系
 - 子类的不变式与父类的不变式
 - $I_Sub(c) \implies I_Super(c)$
 - $I_Super(c) \not\implies I_Sub(c)$
 - 子类重写方法的前置条件与父类方法的前置条件
 - $Requires(Super::f) \implies Requires(sub::f)$
 - $Requires(Sub::f) \not\implies Requires(Super::f)$
 - 子类重写方法的后置条件与父类方法的后置条件
 - $Ensures(Sub::f) \implies Ensures(Super::f)$
 - $Ensures(Super::f) \not\implies Ensures(Sub::f)$

$Requires(Super::f) \iff Requires(sub::f)$
 $Ensures(Super::f) \iff Ensures(sub::f)$

规格化设计回顾



- **WARRANTY**方法
 - Step1(**W**hy): 为什么需要这个方法?
 - Step2(**A**cceptance criteria): 这个方法所提供结果正确的判定条件是什么?
 - Step3(clea**R R**equirement): 这个方法是否需要调用者做出一些要求, 从而确保能够产生正确结果?
 - Step4(**A**Nticipated changes): 这个方法执行期间是否需要修改输入数据或者所在对象数据?
 - Step5(**T**rust**Y**): 无需代码即可确认其语义
- 这是北航OO课总结的方法!

规格化设计回顾

- 契约式设计是一种基于信任机制+权利义务均衡机制的设计方法学，JML源自于契约式设计的需要
- 信任机制
 - **类提供者**信任使用者能够确保所有方法的前置条件都能被满足
 - **类使用者**信任设计者能够有效管理相应的数据和访问安全
- 权利义务均衡机制
 - 任何一个类都要能够存储和管理规格所定义的数据（义务）
 - 任何一个类都要保证对象始终保持有效（义务）
 - 任何一个类都可以拒绝为不满足前置条件的输入提供服务（权利）
 - 或者通过异常来提醒使用者
 - 任何一个类都可以选择自己的表示对象而不受外部约束（权利）

规格化设计回顾

- 规格化设计是一种致力于保证程序正确性的方法
 - 正确性：在规定的输入范畴内给出满足规定要求的输出
 - “你”如果能够保证前置条件成立，“我”就能保证后置条件成立
 - 正确性基础
 - 实现满足规格 && 规格满足需求
- 规格化设计的核心
 - 方法的前置条件
 - 方法的后置条件
 - 对象的不变式

基于JML的规格模式

- 规格模式：一种构造规格表达结构的模式，design pattern面向软件功能，规格模式面向规格表达。
 - 规格是关于软件模块的设计
 - 规格模式是关于规格的设计
- 不同于维基百科上的specification pattern，这是一种design pattern，讨论如何简化一个模块的接口设计
- 规约逻辑与代码逻辑（构造逻辑）有显著差异
 - 规约逻辑不关心如何得到相关数据（中间数据或最终数据），只关心相关数据必须满足的约束条件
 - 构造逻辑关心如何通过数据结构和计算步骤来获得相关数据

基于JML的规格模式

- 约束类别
 - 绝对约束
 - 相对约束
- \result数据内容约束
 - 绝对约束
 - 基于**推导数据**的约束
 - 基于**构造性数据**的约束
- 约束组合与共性提取

基于JML的规格模式

- 约束类别
 - 绝对约束 ~ 相对约束
- 绝对约束
 - 可以直接对\result的取值范围进行约束
 - \result > 0, \result == x;
- 相对约束
 - \result与输入参数和this之间应该满足的约束

```
/*@ requires index >= 0 && index < size();  
   @ assignable \nothing;  
   @ ensures \result == nodes[index];   @*/  
public /*@pure@*/ int getNode(int index);
```

基于JML的规格模式

- 往往需要检查方法输入的不同情况，对应着不同的\result规约
 - (fromNodeId != toNodeId)
 - (fromNodeId == toNodeId)
- 使用蕴含关系来进行规约
 - (fromNodeId == toNodeId) ==> \result == 0;

基于JML的规格模式

- \result所引用的可能是一个数据容器
 - 全称型约束：每个元素都必须满足的约束
 - 存在性约束：某些元素所满足的约束
- `int[] select(int max)`: 从IntSet中选出所有不大于max的元素
 - 该选出多少个?
 - 运行时才知道
 - 每个选出来的元素都必须满足的条件?
 - 对于\result中每个元素 x , $x \leq \max \ \&\& \ \text{isIn}(x)$
 - 对于IntSet中每个 $\leq \max$ 的元素, 都在\result中
- 本质上是规约**大小容器**之间的关系
 - IntSet是大容器, select方法返回的是小容器

`Path[] getPathwithNode(int node)`: 返回
PathContainer中所有经过node的path

```
/*@ requires containsNode(node);  
  @ ensures (\forall Path p; this.contains(p)&& p.containsNode(node); (\exists int j;  
0<=j && j< \result.length; \result[j] == p)) &&  
  @      (\forall int j; 0<=j && j< \result.length; \result[j].containsNode(node));  
  @*/  
Path[] getPathwithNode(int node);
```

```
/*@ ensures (\forall int j; 0<=j && j< \result.length; \result[j] <= max &&  
isIn(\result[j])) &&  
  @      (\forall int x; x<=max && isIn(x); (\exists int j; 0<=j && j< \result.length;  
\result[j] ==x));  
  @*/  
int[] select(int max);
```


基于JML的规格模式

- 有时需要对\result中数据的推导特性进行约束，而不是简单对每个数据进行约束
 - 推导特性：\max, \min, \sum, average...
- 推导特性是在\result的数据中进行推导计算获得的一个特性
- Course[] getCoursewithAvgMark(double avg, double eps)
 - CourseManager, 查询那些平均成绩在[avg-eps, avg + eps]范围内的课程
 - Course: CourseSelection[];
 - CourseSelection: courseID, stuID, double mark;

基于JML的规格模式

- 很多时候\result是某个中间数据的统计结果，而这个中间数据往往必须使用构造性算法才能产生，这时就难以直接对\result进行约束
 - `int getLinkedPersons(int pid): Network`, 返回从pid可以认识的不同人数
 - **关键是构造可达节点集**

```
/*@ ensures (!containsNode(pid)) ==> \result == 0;  
  @ ensures (containsNode(pid)) ==> (\exists Person[] ps; (\forall int j; 0<=j &&  
j< ps.length; contains(ps[j].pid) && ps[j].isCircle (pid));\result == ps.length);  
  @*/  
int getLinkedPersons(int pid);
```

基于JML的规格模式

- 最优化求解方法的规格
 - 最优化求解方法：按照给定的目标函数(objective function)从复杂容器中提取相关对象的方法(满足目标函数)
 - 常见例子：求最短路径、求最大连通子图、求最长公共子串等
- getShortestPathLength

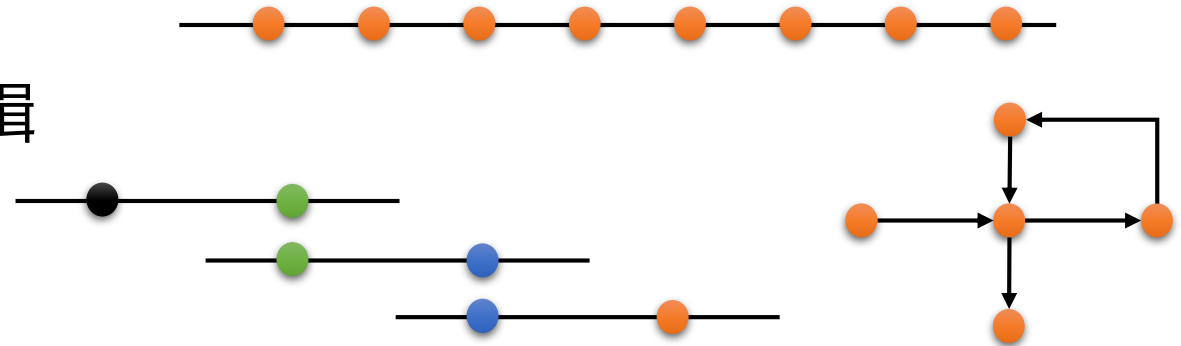
Objective function:

$\exists p(f, t) \in this, \forall p'(f, t) \in this, p(f, t).length \leq p'(f, t).length$

```
/*@ normal_behavior
@ requires ...;
@ ensures (fromNodeId != toNodeId) ==> (\exists int[] snodes; snodes[0] == fromNodeId && snodes[snodes.length - 1] == toNodeId &&
@   (\forall int i; 0 <= i && i < snodes.length - 1; containsEdge(snodes[i], snodes[i + 1])));
@   (\forall int[] cnodes; cnodes[0] == fromNodeId && cnodes[cnodes.length - 1] == toNodeId &&
@   (\forall int j; 0 <= j && j < cnodes.length - 1; containsEdge(cnodes[j], cnodes[j + 1])));
@   snodes.length <= cnodes.length) && \result == snodes.length);
@ ensures (fromNodeId == toNodeId) ==> \result == 0;
@ ...
@*/
public /*@pure@*/ int getShortestPathLength(int fromNodeId, int toNodeId) ...;
```

基于JML的规格模式

- 中间数据的构造甚至需要几层才能规约清楚\result
- 比如地铁系统涉及路线换乘概念：最少换乘路径长度
 - 首先必须回答什么是换乘
 - 构造相应的中间数据来表达换乘逻辑
 - 提取出换乘站点
 - Note：一条路径中可能存在环路



Path中也存在shortestPath问题: `getShortestPathLength(from, to)`

```
@ ensures \result == (\exists int[] tn_idx; tn_idx.length == pseq.length * 2; pseq[0].getNode(tn_idx[0]) == fromNodeId &&  
@   pseq[pseq.length - 1].getNode(tn_idx[tn_idx.length - 1]) == toNodeId &&  
@   (\forall int i; 0 <= i && i < pseq.length - 1; pseq[i].getNode(tn_idx[2 * i + 1]) == pseq[i + 1].getNode(tn_idx[2 * (i + 1)]))));
```

```
@ ensures (\exists Path[] tpath; isConnectedInPathSequence(tpath, fromNodeId, toNodeId);  
@   (\forall Path[] spath; isConnectedInPathSequence(spath, fromNodeId, toNodeId);  
@   tpath.length <= spath.length) && \result == tpath.length - 1);
```

基于JML的规格模式

- 对于复杂的构造式规格，如果不引入组合机制，将使得规格不具有可读性
 - 规格的可读性越差，读错的可能性就越大
 - 形式化方法工程应用受限的一个重要原因
- 规格的组合机制
 - 根据构造需要，逐层构造并组合
 - 本质上和层次化设计类似
 - 构造path序列来实现(from \rightarrow to),从而可以讨论这个序列的长度等特性
 - 然后讨论这个序列必须满足的性质
 - path内连通，path间连接

基于JML的规格模式

- 类中多个方法的规格往往需要构造相同的中间数据
- 类中多个方法的规格往往需要对相同的数据内容进行限制
- 应用**共性提取机制**，增加相应的方法规格，并应用组合机制来简化规格的设计结果
- 如果多个方法都需要构造相同的中间数据，说明需要为这个类构造相应的规格数据内容
 - **设计**：数据内容提取简化规格设计
 - **实现**：用冗余存储换来性能提升

规约设计思维在递归设计、函数式编程、约束求解等都有应用。其核心是首先提出并规定预期结果所满足的约束，然后才会有过程式算法实现。

基于JML规格的实现考虑

- 规格定义了数据内容和方法规约，是代码实现的依据
- 规格实现
 - 选择合适的数据结构和数据表示对象
 - 选择合适算法来实现方法
- 需要重点考虑的几个问题
 - 数据容器的选择
 - 中间数据的存储和检索
 - 规格的层次化
 - 算法的选择

基于JML规格的实现考虑

- 规格不会限制模块内部细节设计，只对模块接口进行规约
- **绝不是对照规格来机械的实现代码**
 - 核心还是架构设计，在此基础上来实现规定的规格
- 基于数据抽象的实现设计思路
 - 规格规定了要管理哪些数据内容？要对数据进行什么处理？
 - RailwaySystem要求提供票价、换乘次数、乘客满意度和连通图结构的查询操作，以及从Graph继承来的连通性、最短路径等查询操作
 - 核心是铁路路网(本质上是Graph)这个数据结构

基于JML规格的实现考虑

- 该使用什么样的数据结构来表示抽象数据
 - 规格只说明数据内容
 - 经典的类设计问题
- 该选择什么样的数据容器
 - 数据访问特征
 - 是否频繁访问某些数据(提前并不可知)
 - ArrayList, LinkedList
 - 数据约束特征
 - 是否允许重复元素
 - 是否有pair型数据 (person与personId)

基于JML规格的实现考虑

- 构造性规格定义了中间数据，这表明要实现这样的规格通常也需要相应的中间数据
- 中间数据如何表示和存储
 - 数据结构设计和容器选择
- 中间数据往往是根据特定功能而存在的数据，容易发生变化
 - 必要时可展开层次化抽象，剥离变化部分，把不变成分独立出来
 - Network的节点与边结构保持稳定
 - Network的边权则有多种形态
 - ➔可以定义多种WeightedEdge，Network层次使用Edge来管理结构

基于JML规格的实现考虑

- 规格中广泛采用层次化设计和组合机制
 - 是不是必须实现这种层次化结构?
- 不是必须, 但大部分情况下应该
 - 规格的层次化往往反映了设计者的架构思考
- 如果实现者有更好的架构考虑, 则可以不按照规格中的层次来设计实现
- 算法考虑应该与架构考虑配合起来
 - 数据抽象层次
 - 数据管理结构

基于JML规格的实现考虑

- 任何一个方法都需要主动去规划有哪些异常
 - 即导致自己的计算无法按照“正常流程”进行，从而不能满足后置条件的状况
- 通过显式的方式在规格中声明相应的异常
 - 标题声明（编译器要求）
 - 后置条件归纳（把不能满足后置条件的情况也纳入到“能够满足”）
- 捕捉异常状况的发生
 - 对输入条件和对象状态的判断
 - 通过try机制来捕捉被调用方法抛出的异常(Exception)
 - 通过try机制来捕捉虚拟机或运行平台抛出的异常(Runtime Exception)
- 异常的处理
 - 记录异常信息
 - 屏蔽局部异常
 - 反射局部异常

基于JML规格的实现考虑

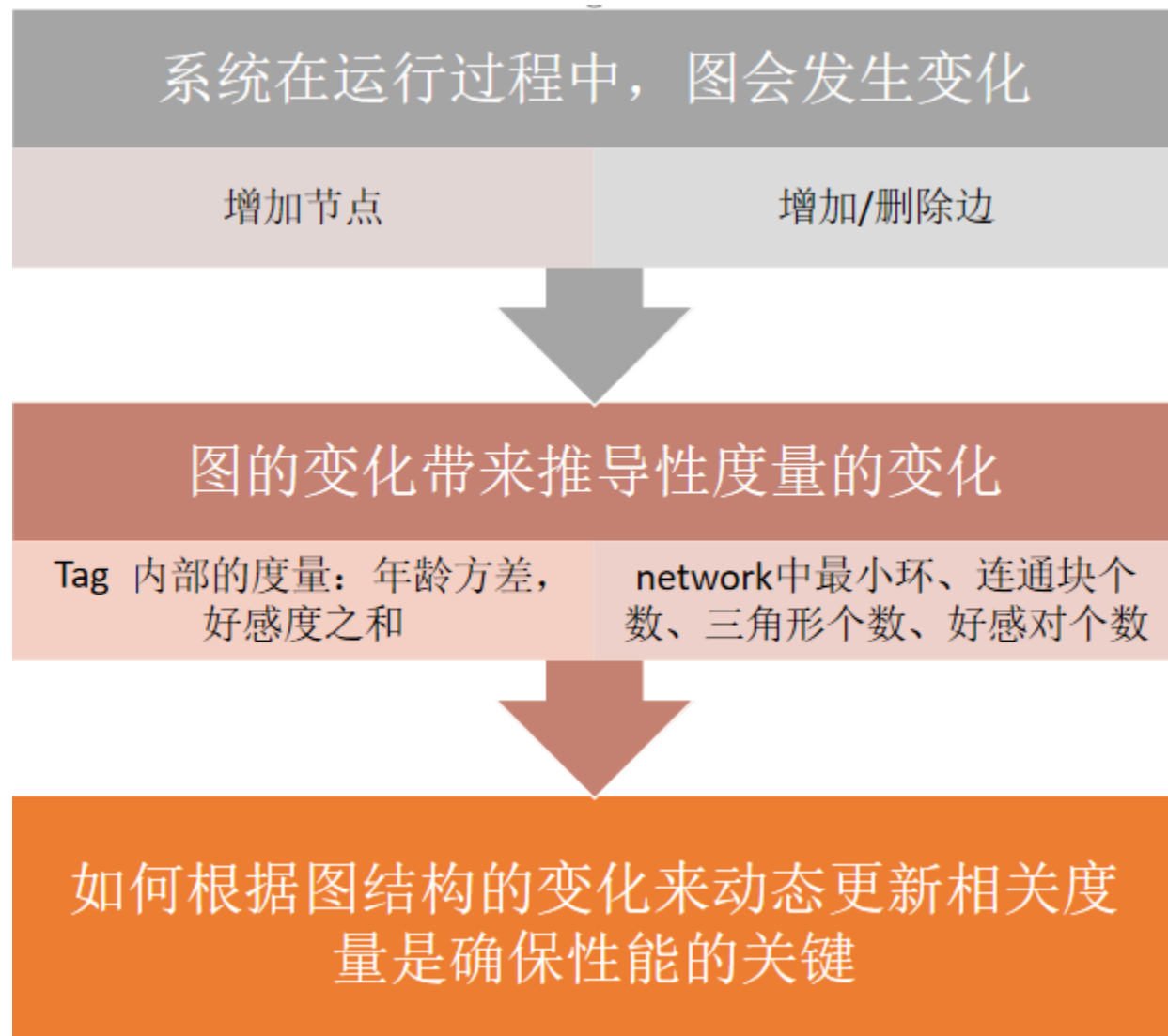
- 契约设计：何时抛出异常
- 契约使用：被调用方法抛出异常的场景和条件
 - 从而获得对状态的准确推理
 - Exception一般并不能携带足够的状态信息
- 从契约使用者角度，需要评估被调用方法所可能抛出异常对自己处理流程的影响
 - 统一处理：无实质性影响
 - 分类处理：影响需要区分
 - 反射处理：对更上层的用户也可能产生影响

本单元设计案例解析

- 本单元作业共三个层次
 - Person：个人，含有多种属性
 - 基本属性查询
 - 人际关系的添加与查询
 - Tag：社交网络中人的好友分类标签
 - 标签中人的加入与退出
 - 标签中人的属性分析
 - Network：社交网络
 - 个人和标签的增删改查、管理
 - 简单人际关系、属性查询
 - 复杂人际关系分析
 - 人与人之间的互动
 - Message：消息
 - 不同消息的继承
 - 消息对于人和标签的影响
 - 如何发送消息
- 通过JML给定了整个社交网络的基本功能规格，如何设计层次之间的交互方法甚至额外层次是本单元作业的关键
- **架构考虑**
 - 抽象为图论问题，增加图层次
 - 考虑各个层次中方法依赖关系，取舍各个方法，解耦
- **性能考虑**
 - 容器的选用
 - 维护中间变量使得查询更快捷
 - 对数据范围的合理考虑(是否需要更好的性能)
 - 算法

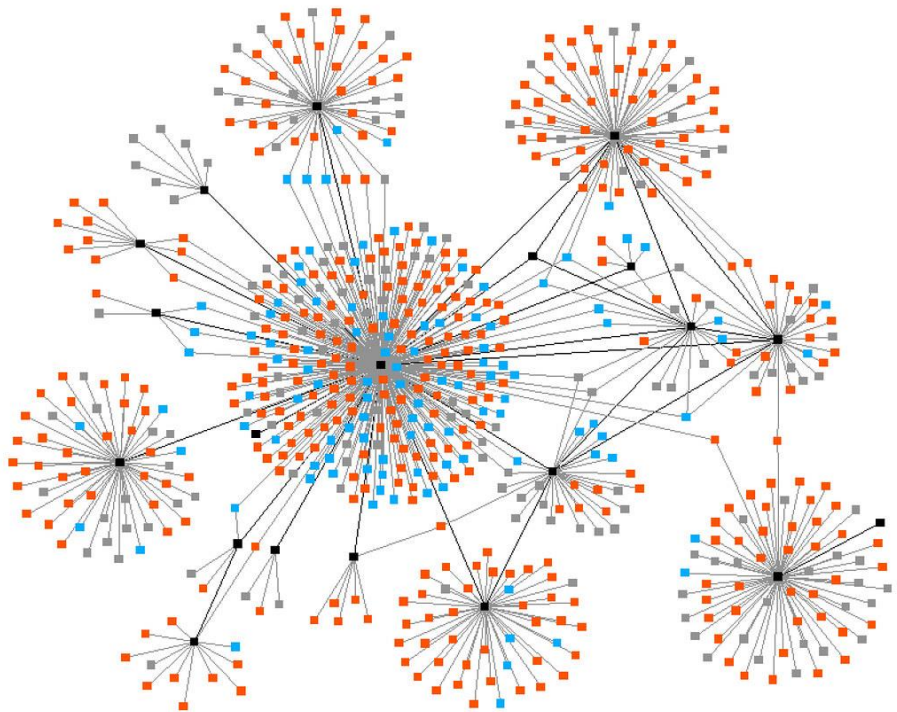
图结构对性能的影响因素：

- 规模:节点及边
- 图的动态变化



关系网络如何建模

- 关系网络的基本特征是一个graph
- 项目给出了group结构，其实是建立了一些cluster
 - 关系网络具有“小世界”特征
- 如果所有的查询和度量分析都要遍历整个大图，性能会如何？
- 按照节点的连接关系特征分成多个cluster--edge container
 - cluster内连接紧密，cluster间连接稀疏



设计案例解析

- 识别易变和不易变结构，独立表示和处理
 - 图结构是基础结构，不会变
 - 节点和边的结构不会变
 - 变化的是图实例，即图模型（特别是cluster）
 - 即节点和边实例会变
 - 由此建图手段需要适应性变化
- 设计时应考虑未来的扩展性
 - 比如边权有无增加新的模式，甚至可能动态变化？
 - 比如图模型结构的变化会带来多重计算，是否支持动态更新图模型？
 - 可使用**Event-chain机制，预留出事件捕捉接口**（addNodeEvent, addEdgeEvent, addTagEvent, structureChangeEvent等）和相应的触发处理机制

基于JML规格的测试与验证

- JML为测试提供了划分依据和判定依据
 - 依据前置条件的测试划分
 - 依据规格数据内容的状态设置→测试场景
 - 依据后置条件和不变式的测试检查
- 应使用JUnit来实现测试执行的批处理和自动化

基于JML规格的测试与验证

- 除了测试，还可以进行形式验证
 - 检查代码实现是否满足规格
- 使用抽象函数在规格数据和实现数据之间建立连接/映射
- 单一层次的验证
 - 方法执行结果是否与方法后置条件相符
 - 方法执行结果是否违背不变式或修改约束
- 跨层次验证
 - 基于LSP原则
 - 关键是重写方法

基于JML规格的测试与验证

- 也可以通过可测试性设计来进行验证
 - 在代码中内置检查手段
 - 测试和验证相融合
- 构造不变式检查方法：repOK
- 根据方法的前置条件，设置对象的状态
 - 调用repOK
 - 调用相应方法
 - **检查方法返回结果是否满足后置条件**
 - 调用repOK
- 方法后置条件也可以自动检查
 - 为类的每个方法独立实现一个专用于测试的***repOK检查方法
 - 一般化方法
 - 专门建立一个(输入,期望输出)配对表，拿到实际输出时与期望输出进行对比
 - 枚举式方法



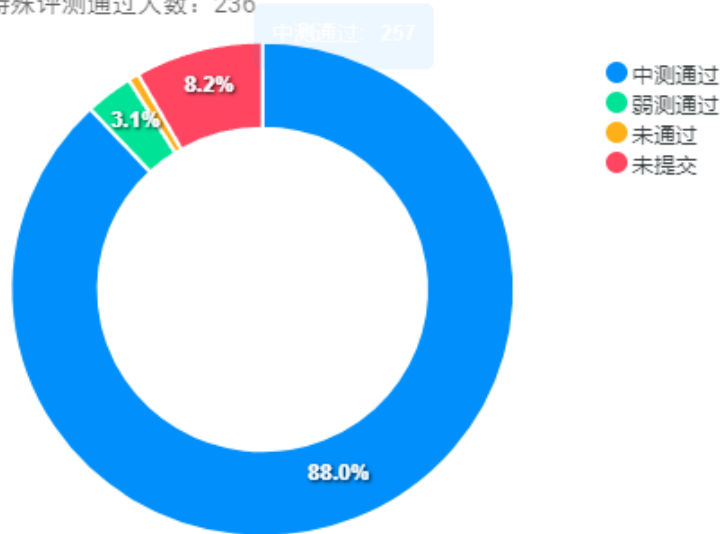
基于JML规格的测试与验证

- 最终目标：检测需求和实现的一致性
- **需求 – 规格**一致性：由规格设计人员保证
- **规格 – Junit 测试**一致性：
 - 根据规格逐条写出Junit测试方法
 - 作业：根据需求说明构造前置数据和Junit测试
 - 实验 5：提供Junit测试数据和批处理测试方法的构造样例
- **Junit测试 – 代码实现**一致性：
 - 作业：使用前置数据检验代码实现，观察断言是否为假
 - 实验 5：单元测试，数据生成器生成前置数据，调用被测方法后判定断言结果

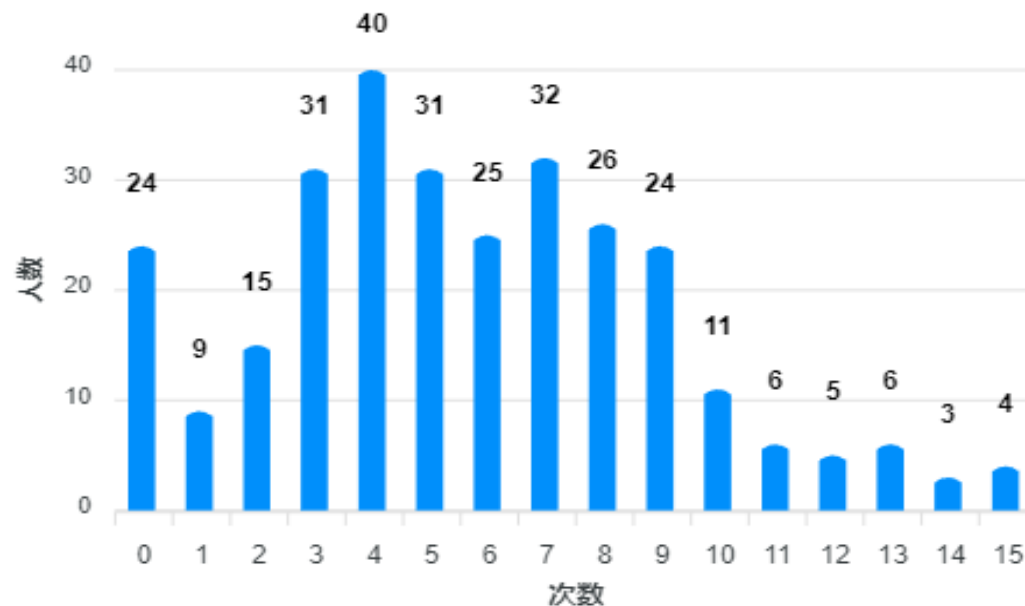


第九次作业分析-公测提交情况

- 该课程的学生人数：292
- 本次作业提交了代码的总人数：268
- 弱测通过的人数：266
- 中测和弱测都通过人数：257
- 特殊评测通过人数：236



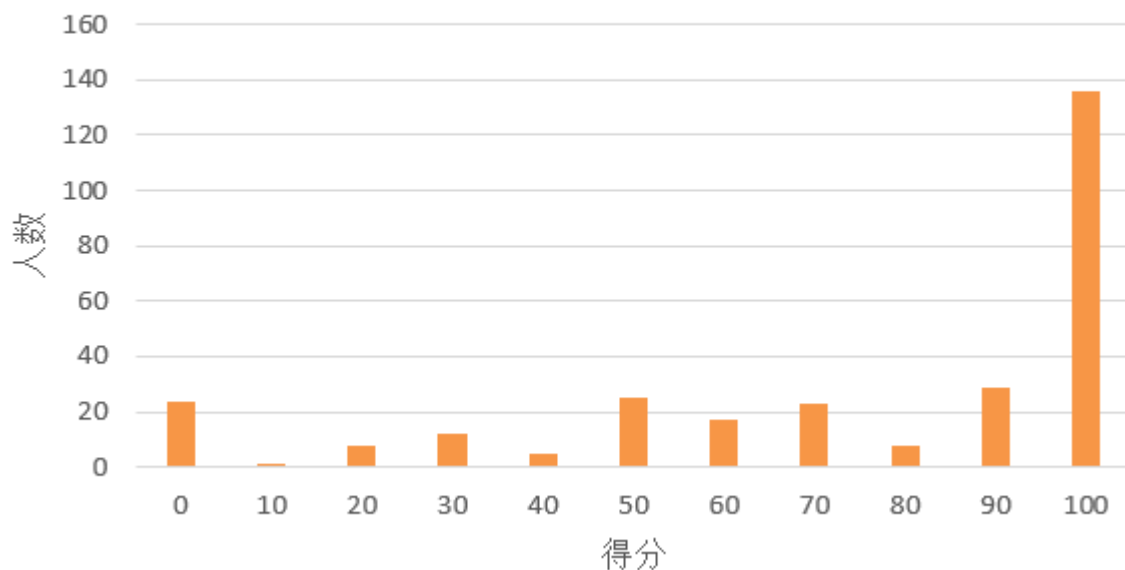
提交的所有作业中
中弱测通过率为 **95.9%**



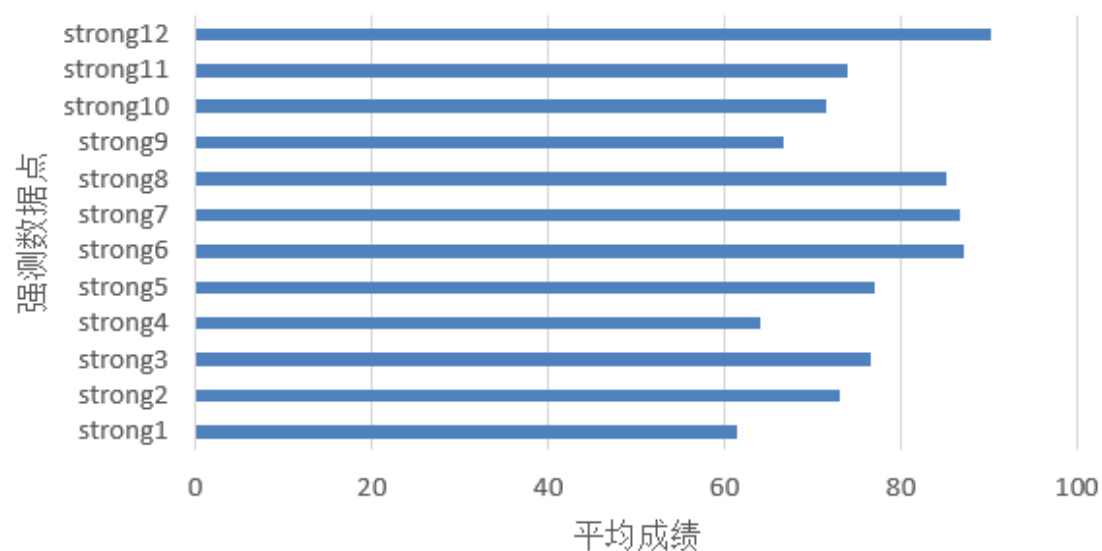
平均提交次数为 **5.57次**

第九次作业分析-公测得分情况

hw9强测成绩分布

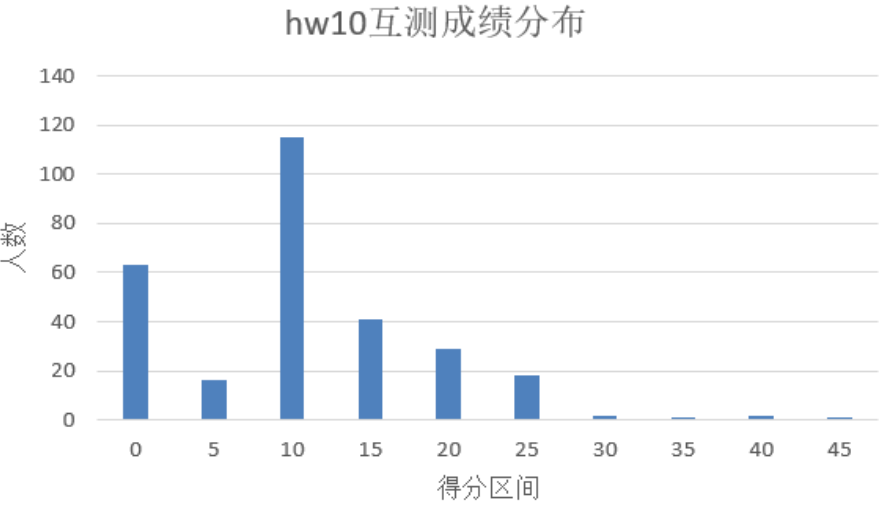


hw9强测数据点平均成绩



- 强测测试点4、7、10、11: qci、qbs、qts指令的**压力测试**，部分同学时间复杂度过高或处理过程有误，导致不能通过这四个测试点
- 强测测试点1: 考察了本次作业的几乎全部指令，随着**指令数量级增加**，正确率也明显降低
- Junit测试点case6: 考察了对于qts方法错误地**addRelation**导致破坏pure规则的检查

第九次作业分析-互测



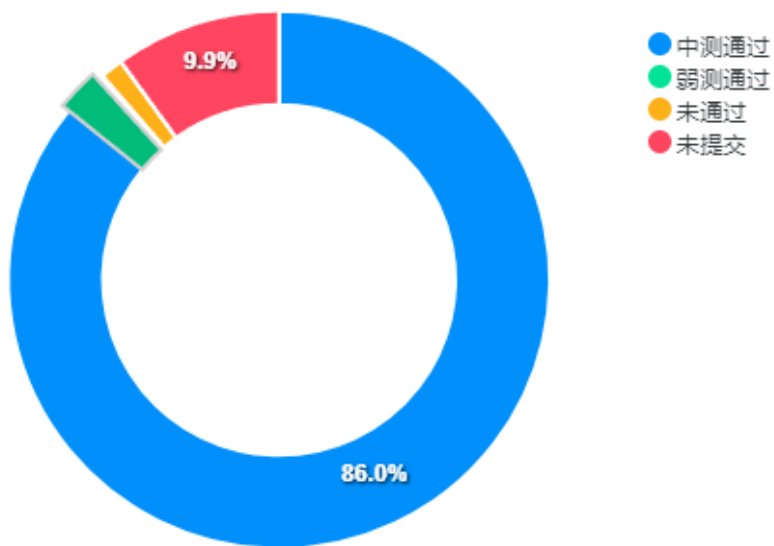
互测得分最高为47分

房间类型	总人数	hack 次数	平均 hack 次数	成功 hack 次数	hack 成功率
A	187	6539	35	150	2.30%
B	44	1535	35	195	12.70%
C	16	460	30	98	21.30%

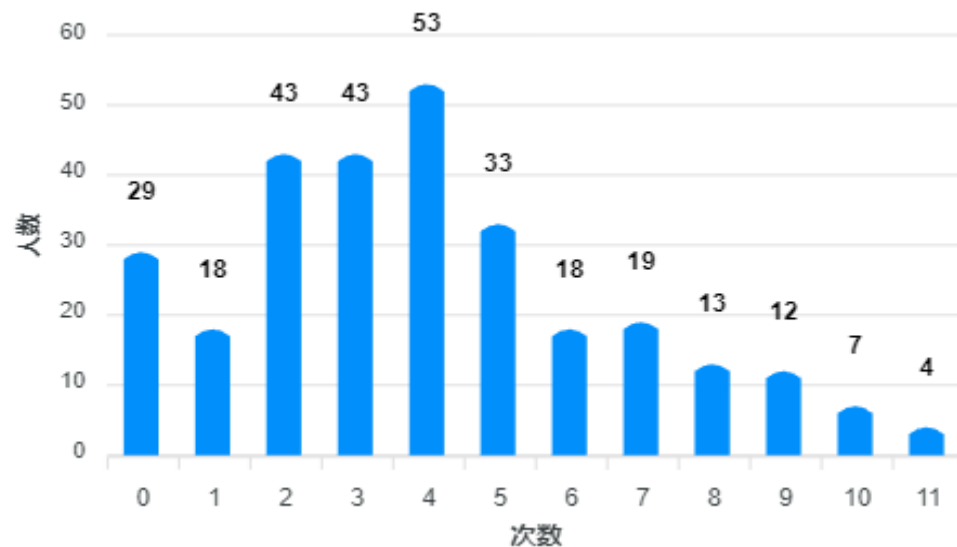
互测意愿强烈

第十次作业分析-公测提交情况

- 该课程的学生人数：292
- 本次作业提交了代码的总人数：263
- 弱测通过的人数：259
- 中测和弱测都通过人数：251
- 特殊评测通过人数：225



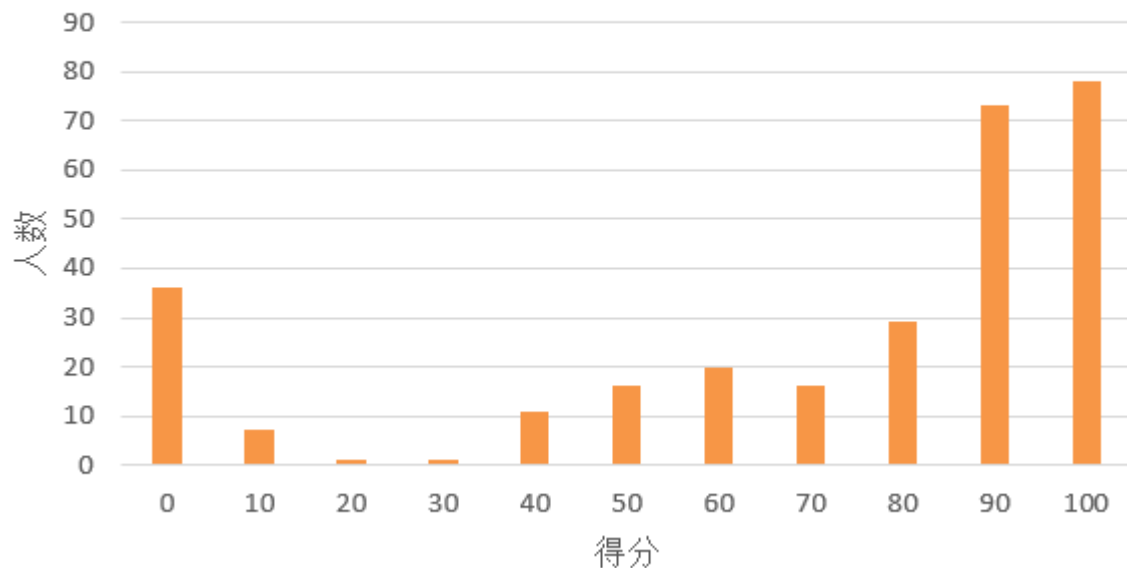
提交的所有作业中
中弱测通过率为**95.4%**



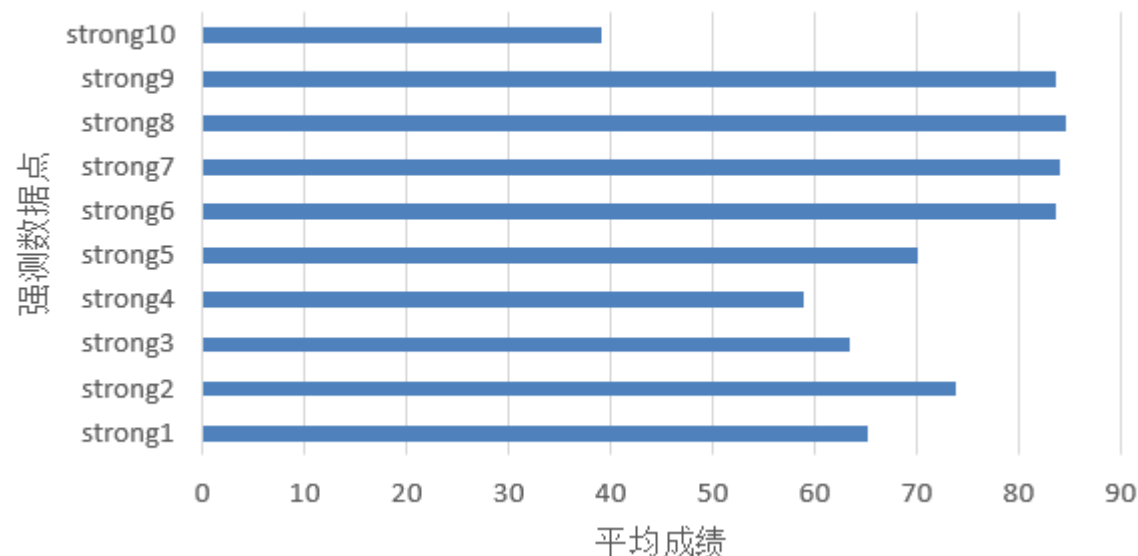
平均提交次数为**4.08次**，显著低于第9次作业

第十次作业分析-公测得分情况

hw10强测成绩分布

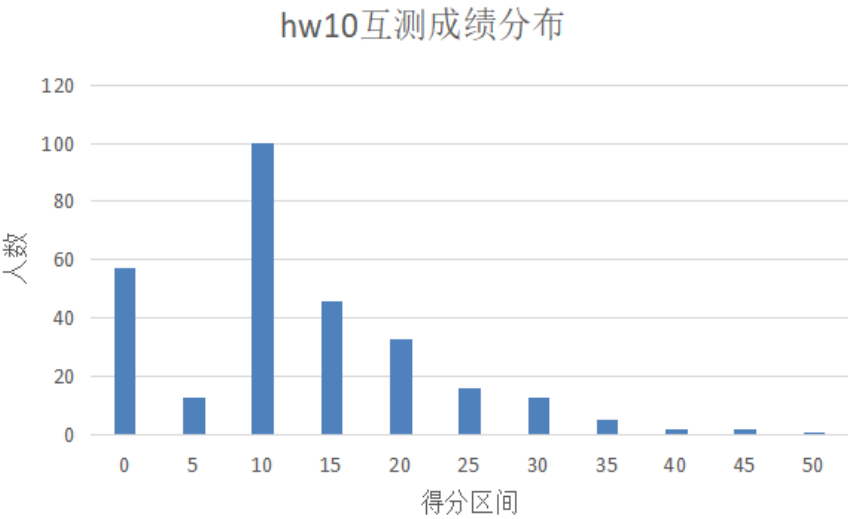


hw10强测数据点平均成绩



- 强测测试点1, 10: 对tag内属性的查询, 部分同学**没有缓存结果**, 每次都重新计算tag对应的属性**导致超时**
- 强测测试点3, 4: 对qba和qcs方法的压力测试, 部分同学细节处使用高级数据结构优化, 但最外层查询仍然使用二重循环导致超时。或者排序时候比较器直接使用减法比较导致溢出
- Junit测试点7: 对qcs方法的测试, 应考虑当测试对象没有acquaintance的情况, 即前置条件为假时不计数, 这一点可以通过qba抛出的异常进行检测

第十次作业分析-互测



互测得分最高为50分

房间类型	总人数	hack 次数	平均 hack 次数	成功 hack 次数	hack 成功率
A	185	7401	40	244	3.30%
B	45	1562	35	158	10.10%
C	14	419	30	101	24.00%

互测收益率较第9次作业有小幅提升

作业

- 针对第三单元的三次作业和课程内容，撰写技术博客
 - (1) 分析本单元的测试过程
 - ① 谈谈你对黑箱测试、白箱测试的理解
 - ② 对单元测试、功能测试、集成测试、压力测试、回归测试的理解
 - ③ 数据构造有何策略
 - (2) 梳理本单元的架构设计，分析自己的图模型构建和维护策略
 - (3) 分析作业中出现的性能问题及其修复情况，谈谈自己对规格与实现分离的理解
 - (4) 总结分析如何利用规格信息来更好的设计实现JUnit测试，以及JUnit测试检验代码实现与规格的一致性的效果
 - (5) 本单元学习体会