

**2019年春季学期  
计算机学院《软件构造》课程**

**Lab 2实验报告**

|  |  |
| --- | --- |
| 姓名 | 江建东 |
| 学号 | 1163450201 |
| 班号 | 1737102 |
| 电子邮件 | 1820985520@qq.com |
| 手机号码 | 18846196989 |

**目录**

[1 实验目标概述 1](#_Toc1988214)

[2 实验环境配置 1](#_Toc1988215)

[3 实验过程 2](#_Toc1988216)

[3.1 Poetic Walks 2](#_Toc1988217)

[3.1.1 Get the code and prepare Git repository 2](#_Toc1988218)

[3.1.2 Problem 1: Test Graph <String> 2](#_Toc1988219)

[3.1.3 Problem 2: Implement Graph <String> 4](#_Toc1988220)

[3.1.3.1 Implement ConcreteEdgesGraph 4](#_Toc1988221)

[3.1.3.2 Implement ConcreteVerticesGraph 6](#_Toc1988222)

[3.1.4 Problem 3: Implement generic Graph<L> 8](#_Toc1988223)

[3.1.4.1 Make the implementations generic 8](#_Toc1988224)

[3.1.4.2 Implement Graph.empty() 9](#_Toc1988225)

[3.1.5 Problem 4: Poetic walks 10](#_Toc1988226)

[3.1.5.1 Test GraphPoet 10](#_Toc1988227)

[3.1.5.2 Implement GraphPoet 10](#_Toc1988228)

[3.1.5.3 Graph poetry slam 12](#_Toc1988229)

[3.1.6 Before you’re done 12](#_Toc1988230)

[3.2 Re-implement the Social Network in Lab1 13](#_Toc1988231)

[3.2.1 FriendshipGraph类 13](#_Toc1988232)

[3.2.2 Person类 15](#_Toc1988233)

[3.2.3 客户端main() 15](#_Toc1988234)

[3.2.4 测试用例 16](#_Toc1988235)

[3.2.5 提交至Git仓库 17](#_Toc1988236)

[3.3 Playing Chess 18](#_Toc1988237)

[3.3.1 ADT设计/实现方案 18](#_Toc1988238)

[3.3.2 主程序ChessGame设计/实现方案 30](#_Toc1988239)

[3.3.3 ADT和主程序的测试方案 36](#_Toc1988240)

[3.4 Multi-Startup Set (MIT) 41](#_Toc1988241)

[4 实验进度记录 41](#_Toc1988242)

[5 实验过程中遇到的困难与解决途径 41](#_Toc1988243)

[6 实验过程中收获的经验、教训、感想 42](#_Toc1988244)

[6.1 实验过程中收获的经验和教训 42](#_Toc1988245)

[6.2 针对以下方面的感受 42](#_Toc1988246)

# 实验目标概述

本次实验训练抽象数据类型（ADT）的设计、规约、测试，并使用面向对象 编程（OOP）技术实现 ADT。具体来说：

针对给定的应用问题，从问题描述中识别所需的 ADT；

设计 ADT 规约（pre-condition、post-condition）并评估规约的质量；

根据 ADT 的规约设计测试用例；

ADT 的泛型化；

根据规约设计 ADT 的多种不同的实现；针对每种实现，设计其表示 （representation）、表示不变性（rep invariant）、抽象过程（abstraction function） 使用 OOP 实现 ADT，并判定表示不变性是否违反、各实现是否存在表 示泄露（rep exposure）；

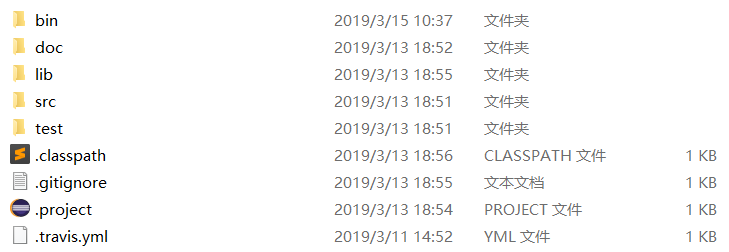
测试 ADT 的实现并评估测试的覆盖度；

使用 ADT 及其实现，为应用问题开发程序；

在测试代码中，能够写出 testing strategy 并据此设计测试用例。

# 实验环境配置

从GitHub的仓库pull到本地，然后使用eclipse作为项目打开，基本环境部署完毕。



在这里给出你的GitHub Lab2仓库的URL地址（Lab2-学号）:

https://github.com/ComputerScienceHIT/Lab2-1163450201

# 实验过程

请仔细对照实验手册，针对三个问题中的每一项任务，在下面各节中记录你的实验过程、阐述你的设计思路和问题求解思路，可辅之以示意图或关键源代码加以说明（但千万不要把你的源代码全部粘贴过来！）。

## Poetic Walks

该问题是用来练习ADT设计，泛型和单元测试设计。spec会提供一些明确的设计要求，据此来实现功能。先设计接口Graph，随后实现两个具体实现接口的类，在实现上述内容的基础上，最后实现一个根据诗歌自行填充文本的工具。

### Get the code and prepare Git repository

该任务的仓库代码地址为

<https://github.com/rainywang/Spring2019_HITCS_SC_Lab2/tree/master/P1>

在gitBash中使用*clone*指令将代码下载到本地，然后使用*git init*指令初始化本地仓库，在更新代码后使用*git commit*推送本地，*git push*推送至远程仓库。



### Problem 1: Test Graph <String>

在此项要求中，只需要实现针对*String*类型的图的单元测试。实验代码中提供了两个测试文件，一个是*GraphStaticTest*，用来测试*Graph*的静态方法*empty*，由于只需要执行一次，所以这里就不对它做任何修改。还有一个文件为*GraphInstanceTest*，用来测试通过*Graph*接口实现的实例对象，这个实例对象用*emptyInstance*函数生成实例对象。

实现思路：

根据*Graph*的spec要求，*add，set，remove，vertices，sources，targets*函数的输入可以划分为以下等价类

*add()*的参数*vertex*可以划分为：

单个顶点，重复顶点

*set()*的参数*source/target*可以划分为：

顶点在*vertices*中，顶点不在*vertices*中，而*weight*可以划分为：大于0和等于0

*remove()*的参数*vertex*可以划分为：

顶点有边相连，无边相连。顶点在图中存在，顶点在图中不存在。

*sources()*的参数*target*可以划分为：

*target*有以*target*作为终点的边相连，没有边相连。

*targets()*的参数*source*可以划分为：

*source*有以*source*作为起点的边相连，没有边相连。

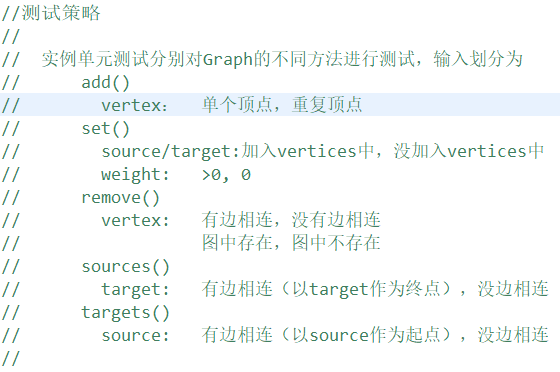
由以上的划分，就可以根据输入的等价类编写单元测试的例子。

实现过程：

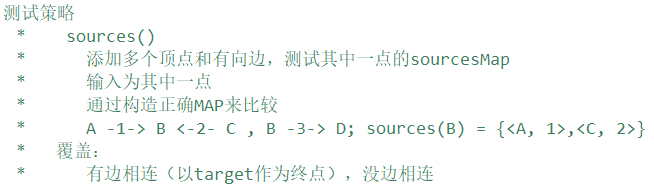
对于返回值是布尔值的函数，通过*assertTrue*或*assertFalse*判断，对于返回值是数据结构的函数，通过构造相对应的正确结构，使用*assertEquals*进行比较。

结果：

总的测试策略



部分函数测试策略



### Problem 2: Implement Graph <String>

P2是实现具体的*Graph*接口中的函数，也就是需要实现实验代码中的*ConcreteEdgesGraph*和*ConcreteVerticesGraph*，对于每一个类，都需要实现*checkRep*函数和*toString*函数，并且需要编写*AF*和*RI*。

#### Implement ConcreteEdgesGraph

**设计：**

*ConcreteEdgesGraph*提供了两个*rep*，*Set*类型的*vertices*和*List*类型的*edges*，分别用来保存图中的顶点和有向边。并且在spec中还定义了一种不可变类*Edge*用来表示一条有向边。

在*ConcreteEdgesGraph*中，*vertices*中保存的必须是*String*类型的顶点，edges中保存的是*Edge*类型的边，*edges*中边所涉及到了顶点需要保存在*vertices*中。

*Graph*中的方法通过对上面两个*rep*操作来实现。

对于*Edge*类，其表示一个具体的有向边，因此需要包含起点，终点和权重。Edge还包括一些获得相关信息的方法和*toString*以及比较相等的*isEqual*方法。

**实现思路：**

有关*ConcreteEdgesGraph*：

*add*方法实现添加顶点的功能，首先需要做合法性检查，随后判断点集中是否包含相同顶点，最后向*vertices*中添加顶点。

*set*方法实现对有向边的添加，更新，删除，特别的，对于没有添加到点集中的点，*set*方法会自动将其添加。首先还是做合法性检查，之后操作逻辑为：

首先判断这个边在边集中是否存在，如果存在则对其进行更新或删除操作，如果不存在，则作为新边将其添加到edges中且添加顶点到*vertices*中。

*remove*方法实现删除顶点和其相连边的功能，为了实现删除相连边的功能，需要对*edges*做一次遍历，将所有以该点作为*source*和*target*的边移除。

*vertices*方法实现返回点集。为了防止表示泄露，需要做防御性拷贝。

*sources*和*targets*方法本质上是相同的，不过一个是查找源点，一个是查找终点，其实现需要对edges做一次遍历，将所有满足条件的顶点和权重存入*map*结构中。

有关*Edge*：

*Edge*的基本方法正常写，但是spec中要求不能使用*equals*方法比较，所以写一个新方法叫*isEqual*，比较的是源点，终点和权重是否相等。

有关*ConcreteEdgesGraphTest*：

该类是继承自*GraphInstanceTest*的子类。有关实例的测试方法自动继承。这里只需要覆盖*emptyInstance*方法获得*EdgesGraph*实例，添加*toString*方法的测试即可。

*toString*方法的输入可以划分为：

空图，无输入，返回相应字符串

只包含点的图，无输入，返回相应字符串

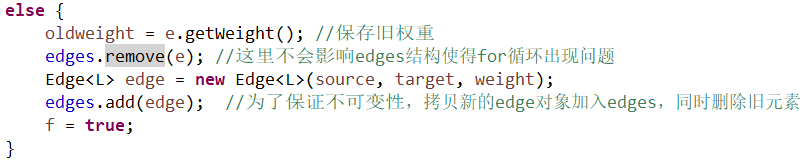
包含点和边的图，无输入，返回相应字符串

同时这个测试类还需要对*Edge*进行测试，由于*Edge*是一个不可变类，进行构造，调用相应方法观察结果即可。

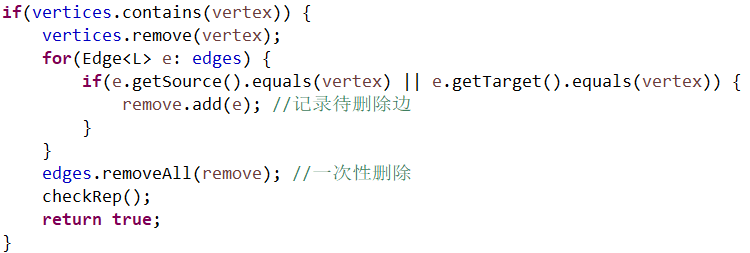
**实现过程：**

*Graph*中其余几个函数实现过程见上述的实现思路。这里说一下有关*防御性拷贝*和*保持数据结构*的问题。

1. *set*方法在更新权重时，涉及到修改*edge*对象内部的*weight*，由于*Edge*是不可变对象，更新时就需要重新生成一个新的*edge*对象，将旧的从*edges*中移除，将新的添加进去。

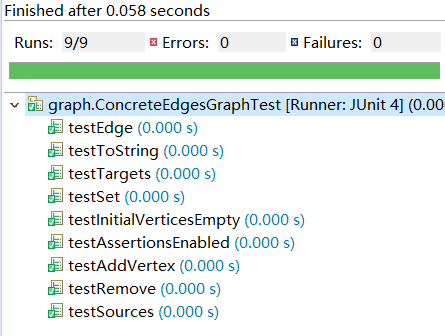


1. *remove*方法中涉及到*list*的*remove*操作，此时如果直接对其内容remove，则会引起数据结构的破坏。Java文档中建议使用迭代器*remove*。这里我选择使用一个*list*临时保存要删除的边，最后使用*removeAll*一次性删除。



*Edge*类的实现也没有什么需要特别注意的东西。

**实现结果：**



#### Implement ConcreteVerticesGraph

**设计：**

*ConcreteVerticesGraph*提供了一个rep， *List*类型的*vertices*，用来保存图中的顶点和有向边关系。并且在spec中还定义了一种可变类*Vertex*用来表示图中的一个顶点和以其作为源点的有向边。

至于为什么要让*Vertex*类型既保存顶点又保存边，是因为这个类中只有一个rep，而图中的有向边可以看作两个顶点的一种关系，自然而然想到*Vertex*也可以实现边的保存。

*ConcreteVerticesGraph*中，*vertices*中保存的必须是*Vertex*类型的顶点。

*Graph*中的方法通过对*vertices*操作来实现。

对于*Vertex*类，其表示一个具体的点和有向边，因此需要包含*String*类型的顶点和一个*Map*用来保存与其相关的终点和权重的映射。*Vertex*还包括一些获得相关信息的方法和添加边的*addEdge*方法，*toString*以及比较相等的*isEqual*方法。

**实现思路：**

有关*ConcreteVerticesGraph*：

*add*方法实现添加顶点的功能，首先需要做合法性检查，随后判断点集中是否包含相同顶点，最后向*vertices*中添加顶点。

*set*方法实现对有向边的添加，更新，删除，特别的，对于没有添加到点集中的点，*set*方法会自动将其添加。首先还是做合法性检查，之后操作逻辑为：

首先判断这个点在点集中是否存在，如果存在则对其进行更新或删除操作，如果不存在，则作为新点将其添加到*vertices*中且添加对应边到*Vertex*的*Map*中。

*remove*方法实现删除顶点和其相连边的功能，为了实现删除相连边的功能，需要对*vertices*做一次遍历，找到该点并删除其包含的所有关系，然后找到其他*Vertex*中含有该点的有向边关系，删除。

*vertices*方法实现返回点集。将*list*转化为*String*类型的*set*返回。

*sources*和*targets*方法本质上是相同的，不过一个是查找源点，一个是查找终点，其实现需要对*vertices*做一次遍历，将所有满足条件的顶点和权重存入*map*结构中。

有关*Vertex*：

*Vertex*的基本方法正常写，但是spec中要求不能使用*equals*方法比较，所以写一个新方法叫*isEqual*，比较的是这个顶点的名称和其边关系*Map*是否相等。

有关*ConcreteVerticesGraphTest*：

该类是继承自*GraphInstanceTest*的子类。有关实例的测试方法自动继承。这里只需要覆盖*emptyInstance*方法获得*EdgesGraph*实例，添加*toString*方法的测试即可。

*toString*方法的输入可以划分为：

空图，无输入，返回相应字符串

只包含点的图，无输入，返回相应字符串

包含点和边的图，无输入，返回相应字符串

同时这个测试类还需要对Vertex进行测试，由于*Vertex*是一个可变类，需要对其*addEdge*，*toString*以及其他方法做测试。对于*addEdge*，输入可划分为

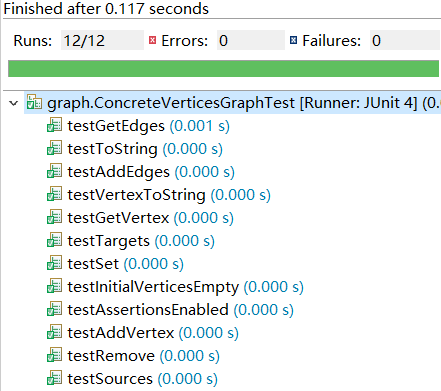
weight >0, 0, <0

**实现过程：**

这里讲一下*set( )*的实现，由于这个类的有向边是保存在*Vertex*中的，所以不能像上面直接遍历*edges*，而是要先找到对应的顶点，再在顶点中所保存的边关系中查找有向边是否存在，否则就创建新的顶点添加新的边。



**实现结果：**



### Problem 3: Implement generic Graph<L>

**P3**要求将*String*参数修改为泛型*L*，以实现ADT更加通用的目的。

#### Make the implementations generic

**设计：**

将所有有关*String*的地方都修改为*L*，所以在需要调用*Vertex*和*Edge*的地方都要添加泛型参数<>。

**实现过程：**

将有关类型修改为以下格式

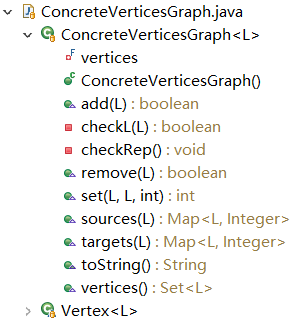
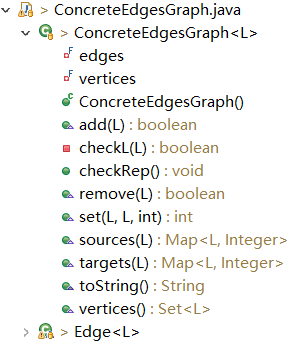
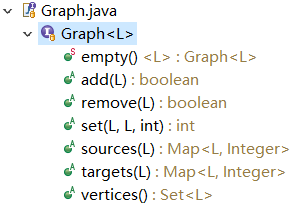
public class ConcreteEdgesGraph<L> implements Graph<L> { ... }

class Edge<L> { ... }

public class ConcreteVerticesGraph<L> implements Graph<L> { ... }

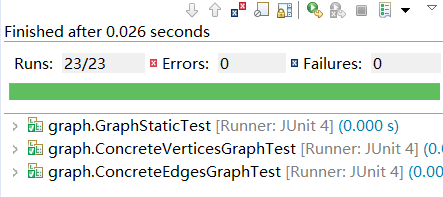
class Vertex<L> { ... }

有关代码展示



**实现结果：**

所有测试通过

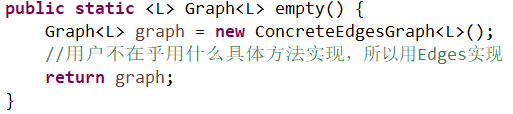


#### Implement Graph.empty()

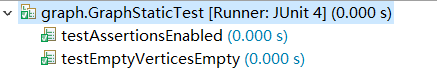
**设计：**

实现*Graph*中的*empty()*函数，以实现通过静态方法可以生成一个空的*Graph*对象。由于客户端不在意通过何种具体实现来实现这种功能，所以选择一个子类对象返回即可。

实现过程：



实现结果：



### Problem 4: Poetic walks

P4要求实现一个能够根据语料库自动补全文本的工具。

#### Test GraphPoet

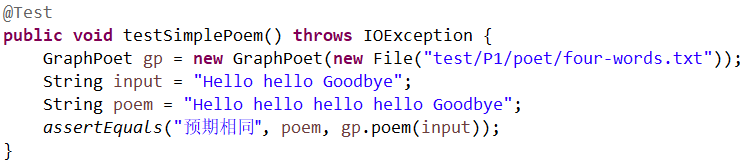
**设计：**

根据spec要求，输入和输出应满足下面的要求：

生成的poem间隔应为一个空格，原单词大小写不变，桥接词小写

**实现结果：**

部分代码



#### Implement GraphPoet

**设计：**

使用Graph的静态方法empty构造图graph。将文本中的每个词作为顶点，每个不同词之间连接的次数作为权重，构造出带权有向图。

对于poem方法，将输入的每个词拆分开，两两搜索，查找是否有匹配的连接词，选择出权重最高的连接词将其添加到两个词中。

**实现思路：**

对于GraphPoet构造器：

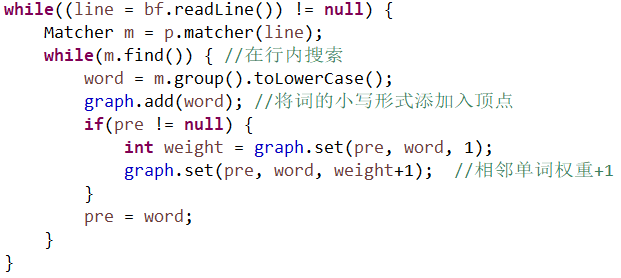
使用正则表达式对文本解析，按行读取，提取出单词存入vertices中，将该单词与下一个词的构成的有向边，权重记为1，存入edges中。

对于poem()：

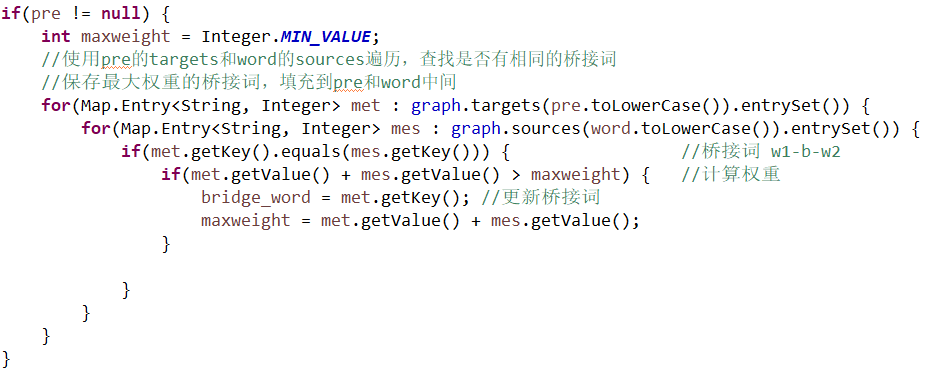
将输入的文本拆分为单词，对于每相邻的两个单词w1，w2，将其分别作为源点和终点进行搜索，如果查找到有一个单词b既是前一个单词w1的终点之一，又是后一个单词w2的源点之一，那么这个词就是桥接词。如果桥接词b不为空，则将其加入到w1和w2之间，否则不添加。其具体实现使用StringBuilder构造，最后输出为String类型。

**实现过程：**

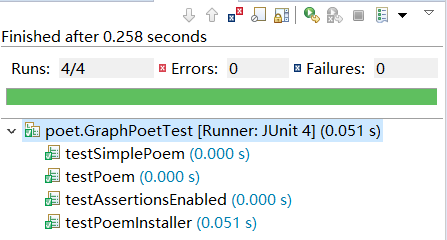
对于构造器，将其变为小写存入图中。



poem方法查找桥接词的源码



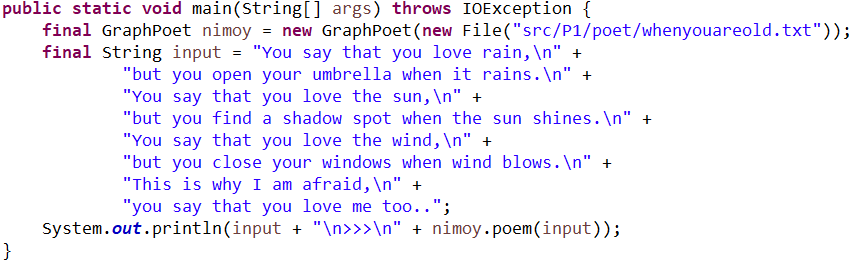
**实现结果：**



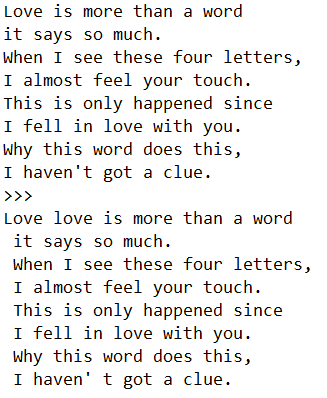
#### Graph poetry slam

**实现过程：**

添加文本作为语料库，将另一首诗作为输入。



**实现结果：**



### Before you’re done

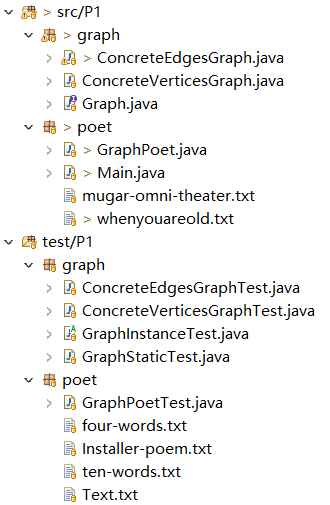
使用*git add .*将当前的修改添加到缓冲区，使用*git commit*添加到本地仓库，使用*git push origin master*推送到远程仓库。







树状结构



## Re-implement the Social Network in Lab1

在这个任务是使用上一个任务中实现的*Graph*型ADT重新实现Lab1中的*FriendshipGraph*，这里需要将*Graph*看作一个已经封装好的ADT并使用它。因为需要使用*Person*类对象，所以*L*泛型参数需要修改为*Person*。这个任务目标在于体验泛型参数的可拓展性和结构的复用。

### FriendshipGraph类

**设计：**

*FriendshipGraph*类中有两种方法，*addVertex*和*addEdge*，可以分别调用*graph*的*add*方法和*set*方法来实现，对于*getDistance*方法，依然可以使用BFS+层数实现，但是里面所有涉及到顶点和边的操作都需要用*graph*对应操作来替代。

对于BFS来说，其实现需要*Queue*队列，因此需要设计一个*Queue*队列类作为数据结构。其能够实现进队，出队，查询是否为空等基本操作。

**实现思路：**

关于*addVertex*：

判断其合法性，然后将*person*对象加入到*graph*的点集*vertices*中。

关于*addEdge*：

判断其合法性，特别是要判断边的两个端点是否都已加入到点集中，随后调用*graph*的*set*方法，将权重记为1加入到边集*edges*中。

关于*getDistance*：

使用BFS来进行遍历，由此生成一颗BFS树，每个结点在树上都有对应的深度，这个深度就是所求的*distance*。实现时需要构造一个队列*Queue*对象，作为数据存储结构。同时还需要一个*depth[]*数组存储每个结点对应深度。

如何记录结点的深度，原理是：每个子节点的深度都是其父节点深度加一。在将子节点加入队列时，深度为其父节点加一，存入*depth*。

遍历时，若找到查找结点，记下深度后跳出循环。

**实现过程：**

这里提供一下*getDistance*的关键代码



队列*Queue*的实现代码



### Person类

**设计：**

*Person*类表示一个在朋友圈中具体的人，因此*Person*应该是*immutable*的，*Person*只有一个rep，为这个人的名称，名称应是*private final*的。*Person*类还需要提供一些基本的*observer*和*equals*方法。

**实现过程：**

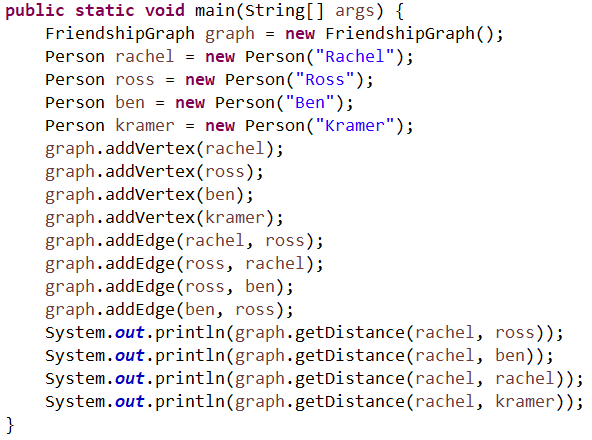
构造方法，rep为*private final*



### 客户端main()

**实现过程：**

将*main*函数中的代码拷贝过来。



### 测试用例

**设计：**

*FriendshipGraphTest*类负责测试*FriendshipGraph*类和*Person*类。*FriendshipGraph*类中对于*addVertex*，*addEdge*这种*Mutator*，需要使用其返回值来进行测试，对于*getDistance*这种*Observer*，直接通过返回值测试。

*addVertex*中输入划分为：空对象，正确对象，重复对象

*addEdge*中输入划分为：

对象为null/对象不为null，对象在点集中/对象不在点集中

*getDistance*中输入划分为：

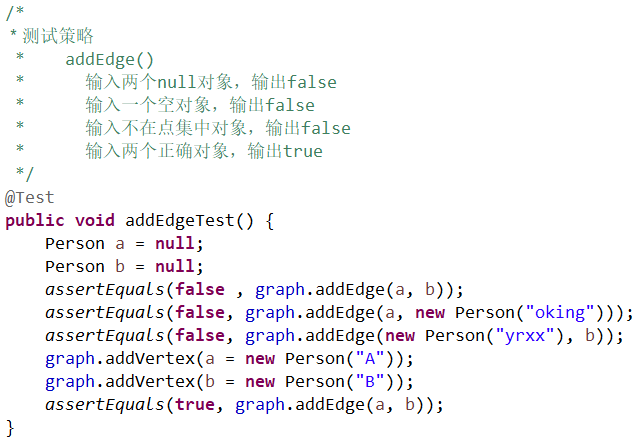
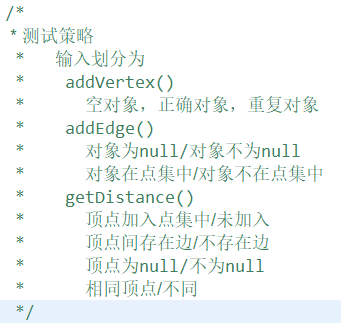
顶点加入点集中/未加入，顶点间存在边/不存在边，

顶点为null/不为null，相同顶点/不同

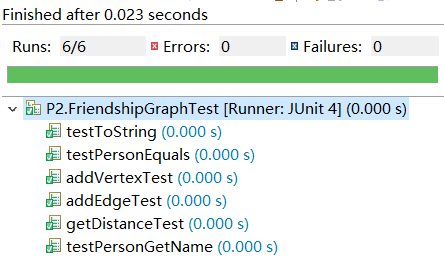
*Person*类由于是*immutable*对象，所以直接测试其*observer*方法，通过*creator*构造。

**实现过程：**

相关输入划分和测试函数



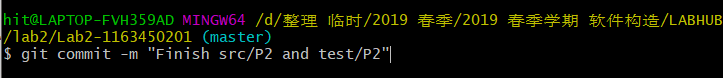
**实现结果：**



### 提交至Git仓库

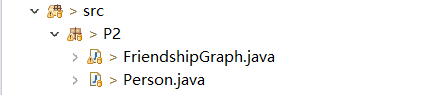
使用*git add .*将当前的修改添加到缓冲区，使用*git commit*添加到本地仓库，使用*git push origin master*推送到远程仓库。







树状结构





## Playing Chess

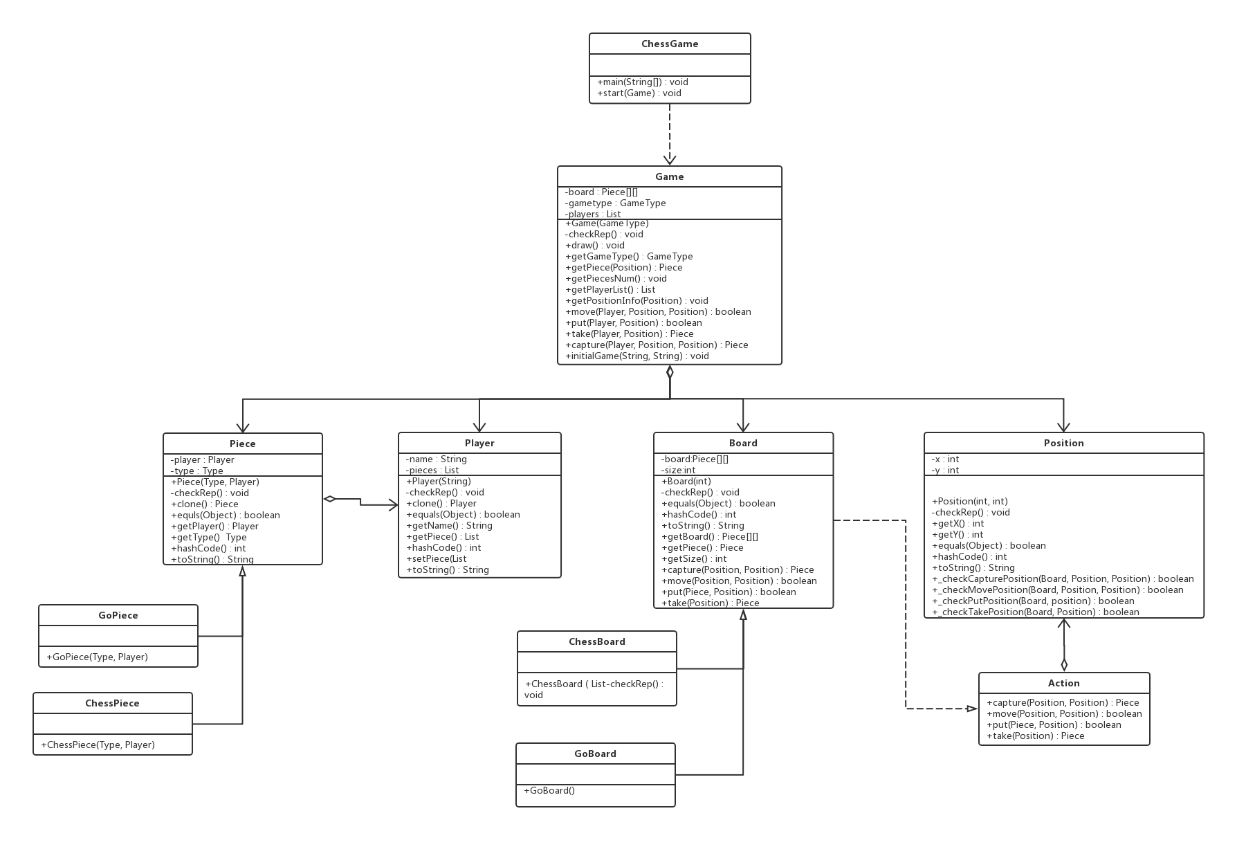
该任务要求自行设计ADT实现一个下棋程序，要求能够实现围棋和国际象棋两种棋类。这个任务主要练习OOP思想和ADT的抽象能力。

### ADT设计/实现方案

自行设计的ADT为

|  |  |  |
| --- | --- | --- |
| **名称** | **类型** | **含义** |
| Action | 接口 Interface | 表示下棋所执行的动作 |
| Board | 类 Class | 表示棋盘的父类 |
| ChessBoard | 类 Class extends Board | 表示象棋棋盘 |
| GoBoard | 类 Class extends Board | 表示围棋盘 |
| Piece | 类 Class | 表示棋子的父类 |
| ChessPiece | 类 Class extends Piece | 表示象棋子 |
| GoPiece | 类 Class extends Piece | 表示围棋子 |
| Player | 类 Class | 表示游戏玩家 |
| Position | 类 Class | 表示棋盘位置 |
| Game | 类 Class | 表示棋类游戏 |
| Chess Game | 类 Class | 棋类游戏的主程序 |
| GameType | 枚举类 Enum | 表示棋类游戏的种类 |
| Type | 枚举类 Enum | 表示棋子种类 |

总的ADT之间的关系如下图



**思路：**

手册要求必须实现*Game*，*Player*，*Board*，*Piece*，*Position*，*Action*这些ADT来组成一场棋类游戏。一场棋类游戏包括*玩家*，*棋子*，*棋盘*三个物体。而其中又存在4种下棋动作。其中，每一枚*Piece*都有所属的*玩家*，同时每一枚*棋子*都有其固定的属性。对于*Player*来说，玩家都需要有一个*名称*，同时玩家还应知道自己手上有那些棋子。*Board*是用来放置棋子的，也就是说棋盘能够保存棋子的*相应位置*，棋盘也应是唯一知道棋子位置的对象。棋盘对于棋子可以进行一系列动作的*具体操作*，而操作的*指令可以由玩家发送*，然后通过游戏中转给*棋盘执行*。*Game*，也就是包含所有物体的一个概念，保存场上所有物体的信息，具有两种棋类游戏的属性，对于不同的游戏类型，负责信息的转发，调用相应游戏类型物体动作的执行。*ChessGame*作为主程序，创造*Game*对象，代表玩家操作game对象，发送命令。

Action（Interface 接口）

**设计：**

*Action*接口用来实现下棋的四个动作*move，put，take，capture*。precondition根据手册设计，其中对具体参数都有明确的限制，对于非法参数会放在Game中处理。

实现：

定义四个方法分别为

*boolean put(Piece pie, Position pos);*

*boolean move(Position source, Position target);*

*Piece take(Position pos);*

*Piece capture(Position source, Position target);*

Board（Class implements Action）

**设计：**

*Board*作为所有棋盘的父类，需要具有一般棋盘的特征，由于手册中要求棋盘为n\*n方形，因此不考虑非正方形情况。无论是以格子作为棋子的放置处还是以交叉点作为棋子的放置处，都可以抽象出将其视为坐标点处理，其中，国际象棋的棋盘为8\*8的坐标系，围棋因为边线上也可以放置棋子，所以是19\*19的坐标系。那么棋盘的尺寸就可以用size来表示。

考虑到存取的速度，采用*Map*来实现棋盘，每个位置映射唯一一个棋子。 *Board*对象应该实现基本的增查删改，同时实现对棋子的四种操作。

|  |  |
| --- | --- |
| **属性** | **内容/说明** |
| Rep | board : Map<Position,Piece>  Map，存储棋盘上的棋子  size : int 表示棋盘的尺寸 |
| Mutable/Immutable | 作为一个棋盘，需要时刻对棋子状态进行更新，board的值会变动，因此是mutable类。 |
| AF | AF(board, size) = 一个保存棋子位置的棋盘 |
| RI | 作为棋盘的尺寸，其一定 > 0  棋盘中存储的对象必须为Piece或null两者之一 |
| Safety from rep exposure | board和size都是private，仅子类可以访问，size为基本类型int不可变。 |

**实现：**

|  |  |
| --- | --- |
| **方法名称** | **具体实现** |
| public Boolean move(Position source, Position target) | 继承自Action接口，具体实现为将数组中source位置的棋子取出移动到target位置，将原位置棋子抹除。要求source和target位置合法且起始位置不为空，目标位置为空。 |
| public boolean put(Piece piece, Position position) | 继承自Action接口，具体实现为将参数中piece对象存储在对应位置数组中。要求piece对象不为空且放置位置为空。 |
| public Piece take(Position position) | 继承自Action接口，具体实现为将对应位置棋子抹除，然后返回被抹除棋子的拷贝对象。要求位置合法且存在棋子。 |
| public Piece capture(Position source, Position target) | 继承自Action接口，具体实现为将起始位置棋子移动至目标位置，将目标位置原棋子抹除，返回其拷贝对象。 |
| @Override  public boolean equals(Object otherobject) | 实现Board类的比较方法，具体实现为比较board的尺寸并且对board中二维数组中每一个元素进行比较，如果都相等则这两个board对象相等。 |
| public Piece getPiece(Position position) | 取得某个位置上的棋子对象的拷贝 |
| private void checkRep() | 检查size是否大于0，遍历二维数组，检查里面的对象是否为Piece对象或null |

ChessBoard（Class extends Board）

**设计：**

*ChessBoard*继承自*Board*父类，实现的是国际象棋棋盘。棋盘里只能存储象棋子。在构建象棋盘时，构造器会接收两个象棋*list*，分别为两个玩家拥有的象棋，然后在棋盘中将对应象棋子摆放到对应的位置。

|  |  |
| --- | --- |
| **属性** | **内容/说明** |
| Rep | 同Board类 |
| Mutable/Immutable | 同Board类，mutable类。 |
| AF | AF(board, size, P1, P2) = 一个保存象棋棋子位置的象棋棋盘 |
| RI | board所存储对象为ChessPiece或null  size > 0, size == 19 |
| Safety from rep exposure | board和size都是protected，仅子类可以访问，size为基本类型int不可变。 |

|  |  |
| --- | --- |
| **方法名称** | **具体实现** |
| 同Board类 |  |
| private void checkRep() | 检查size是否等于8  对象棋盘所有位置进行检查，检查棋子位置是否摆放正确 |
| public ChessBoard(List<Piece> P1, List<Piece> P2) | 调用父类构造器，将size设置为8.按照国际象棋摆放规则，将P1，P2中棋子摆放到正确位置 |
| @Override  public boolean put(Piece piece, Position pos) | 抛出异常  "象棋盘中调用了put方法" |
| @Override  public Piece take(Position pos) | 抛出异常  "象棋盘中调用了take方法" |

GoBoard（Class extends Board）

**设计：**

*GoBoard*继承自*Board*父类，实现的是围棋棋盘。棋盘里只能存储围棋子。围棋盘初始状态应为空盘。

|  |  |
| --- | --- |
| **属性** | **内容/说明** |
| Rep | 同Board类 |
| Mutable/Immutable | 同Board类，mutable类。 |
| AF | AF(board, size) = 一个保存棋子位置的围棋盘 |
| RI | board所存储对象为GoPiece或null  size = 19 |
| Safety from rep exposure | board和size都是private，仅子类可以访问，size为基本类型int不可变。 |

|  |  |
| --- | --- |
| **方法名称** | **具体实现** |
| 同Board类 |  |
| public GoBoard() | 调用父类构造器，size设置为19 |
| private void checkRep() | 检查size大小，遍历检查数组中元素，是否为null或黑白围棋子 |
| @Override  public boolean move(Position source, Position target) | 抛出异常  "围棋盘中调用了move方法" |
| @Override  public Piece capture(Position source, Position target) | 抛出异常  "围棋盘中调用了capture方法" |

Piece（Class）

**设计：**

*Piece*作为所有棋子的父类，需要具有一般棋子的特征。根据手册要求，棋类游戏中只有围棋和国际象棋两种棋类，而且游戏中存在两个不同的玩家，因此棋子中需要存在的属性有：所属的玩家，棋子的种类。棋子在游戏中是不会产生变化的，因此*Piece*设计为*immutable*类型。*Piece*类中提供基本的*observer*方法使用和一些其他方法。

|  |  |
| --- | --- |
| **属性** | **内容/说明** |
| Rep | type : Type 棋子种类  player : Player 游戏玩家 |
| Mutable/Immutable | 因为type和player都是final的，因此是immutable |
| AF | AF(type, player) = 一个保存有所属玩家和种类的棋子 |
| RI | type非空  player非空 |
| Safety from rep exposure | type和player都是private final，初始化后不能修改。type为枚举类型，Player类自身提供了良好防御性。没有mutator方法 |

|  |  |
| --- | --- |
| **方法名称** | **具体实现** |
| public Piece(Type type, Player player) | 设置该棋子的type和player |
| private void checkRep() | 对type和player进行检查，判断是否为空对象 |
| public Piece clone() | 新建一个Piece对象，将内容拷贝返回 |
| @Override  public boolean equals(Object otherobject) | 检查两个对象的type和player是否相等，调用player的equals方法来对player进行判断。 |
| public Player getPlayer() | 返回棋子所属玩家的对象引用 |
| public Type getType() | 返回棋子type属性 |
| @Override  public int hashCode() | 调用Objects的hash方法获得哈希码 |
| @Override  public String toString() | 打印棋子的种类和玩家信息 |

GoPiece（Class extends Piece）

**设计：**

*GoPiece*继承自*Piece*父类，实现围棋子。围棋棋子的种类只能为黑*BLACK*或白*WHITE*之一。

|  |  |
| --- | --- |
| **属性** | **内容/说明** |
| Rep | 同Piece父类 |
| Mutable/Immutable | 同Piece父类，immutable类型 |
| AF | AF(type, player) = 一个保存所属玩家和种类的围棋棋子 |
| RI | type只能为*WHITE*和*BLACK*其一  player非空 |
| Safety from rep exposure | type和player都是private final，初始化后不能修改。type为枚举类型，Player类自身提供了良好防御性。没有mutator方法 |

|  |  |
| --- | --- |
| **方法名称** | **具体实现** |
| 同Piece父类 |  |
| public GoPiece(Type type, Player player) | 调用父类构造器 |
| private void checkRep() | 检查type是否为黑或白，检查玩家是否为空对象 |

ChessPiece（Class entends Piece）

**设计：**

*ChessPiece*继承自*Piece*父类，实现象棋棋子。象棋棋子的种类为六种种类之一。

|  |  |
| --- | --- |
| **属性** | **内容/说明** |
| Rep | 同Piece父类 |
| Mutable/Immutable | 同Piece父类，immutable类型 |
| AF | AF(type, player) = 一个保存所属玩家和种类的象棋棋子 |
| RI | type为*PAWN，ROOK，KNIGHT，BISHOP，KING，QUEEN*其一  player非空 |
| Safety from rep exposure | type和player都是private final，初始化后不能修改。type为枚举类型，Player类自身提供了良好防御性。没有mutator方法 |

|  |  |
| --- | --- |
| **方法名称** | **具体实现** |
| 同Piece父类 |  |
| public ChessPiece(Type type, Player player) | 调用父类构造器实现 |
| private void checkRep() | 检查type是否为6种类型之一，检查玩家对象是否为空 |

Player（Class）

**设计：**

*Player*表示棋类游戏中的玩家，一个玩家是一个具有名称且拥有一些棋子的对象。玩家的名称是用来区分不同玩家之间的标签，拥有棋子*list*是为了管理所拥有的棋子，在对棋子操作时，能够及时更新信息。由于手册中要求能够对一场游戏中间能够查询双方玩家的棋子数量，所以棋子*list*的另一个作用就是快速查询棋子数量。

由于*Player*存在一个list，因此它是一个*mutable*对象。*Player*内应有一些基本的*observer*方法和一个用于更新棋子list的*mutator*方法。

|  |  |
| --- | --- |
| **属性** | **内容/说明** |
| Rep | name : String 玩家名称  pieces : List<Piece> 玩家拥有的棋子 |
| Mutable/Immutable | 由于存在List类型，因此是mutable类型 |
| AF | AF(name, pieces) = 一个棋类游戏中的玩家 |
| RI | name非null，非空串  pieces里不存在null对象 |
| Safety from rep exposure | 所有的域都是private，name为final。setPiece()方法做了防御性拷贝。getPiece()做了防御性拷贝。  Player自身提供了clone()方法。客户端无法直接对内部的域做操作。 |

|  |  |
| --- | --- |
| **方法名称** | **具体实现** |
| public Player(String name) | 将参数name赋值给玩家名称中 |
| private void checkRep() | 检查名称是否为空串或空对象，检查list中是否存在null对象 |
| @Override  public boolean equals(Object otherobject) | 比较玩家的名称是否与其他玩家相同，如果相同则判断为相等 |
| @Override  public int hashCode() | 调用Objects的hash方法产生哈希码 |
| @Override  public String toString() | 输出玩家的名称，拥有棋子的数量 |
| public void setPiece(List<Piece> pieces) | 将参数pieces进行防御性拷贝，赋值给对象内部的pieces域。 |
| public List<Piece> getPiece() | 返回piecesList的拷贝 |
| public String getName() | 返回玩家的名称 |
| public Player clone() | 新建一个Player对象，将内容拷贝后传递给新对象。 |

Position（Class）

**设计：**

*Position*表示棋盘上的位置，以*(x,y)*形式表示。因此设计的rep中包括x和y，手册中对棋盘定义为一个以左下角为原点的坐标系，因此x和y都必须大于等于0。对于一个坐标，一旦设置后就不应产生变化，因此*Position*是*immutable*的。*Position*中应提供基本的*observer*来获取坐标信息。

与此同时，根据手册要求，应当提供对位置合法性进行检查方法。位置合法性检查只和位置，棋盘的大小有关，所以这里选择将检查函数作为*静态方法*放在*Position*类中。

|  |  |
| --- | --- |
| **属性** | **内容/说明** |
| Rep | x : int x坐标  y : int y坐标 |
| Mutable/Immutable | 由于都是基本类型且不可修改，为immutable类型 |
| AF | AF(x,y) = 一个棋盘中的位置坐标 |
| RI | x >= 0  y >= 0 |
| Safety from rep exposure | x和y都是private final修饰 |

|  |  |
| --- | --- |
| **方法名称** | **具体实现** |
| public Position(int x, int y) | 将参数x y赋值给x y域 |
| private void checkRep() | 检查x y是否小于0 |
| @Override  public String toString() | 输出x y坐标 |
| @Override  public boolean equals(Object otherobject) | 判断x y和其他Position对象的x y是否相等 |
| @Override  public int hashCode() | 调用Objects的hash方法生成哈希码 |
| public int getX() | 返回x坐标 |
| public int getY() | 返回y坐标 |
| public static boolean  checkPutPosition(Board board, Position position) | 提供给put函数使用，检查x和y坐标是否在棋盘大小范围内，然后检查参数位置是否存在棋子 |
| public static boolean  checkTakePosition(Board board, Position position) | 提供给take函数使用，检查x和y坐标是否在棋盘大小范围内，然后检查参数位置是否不存在棋子 |
| public static boolean  checkMovePosition(Board board, Position source, Position target) | 提供给move函数使用，检查x和y坐标是否在棋盘大小范围内，然后检查两个位置是否相同，再检查起始位置是否不存在棋子，目标位置是否存在棋子 |
| public static boolean  checkCapturePosition(Board board, Position source,Position target) | 提供给capture函数使用，检查x和y坐标是否在棋盘大小范围内，然后检查两个位置是否相同，再检查起始位置是否不存在棋子，目标位置是否不存在棋子 |

Game（Class）

**设计：**

*Game*表示一场棋类游戏，这个游戏有三个物体组成，分别为棋盘，玩家，棋子，*Game*作为与主程序交互的唯一对象，起到了桥梁的作用，将对于主程序发来的*put，take，move，capture*动作转化为对三个物体的操作。因此*Game*的功能主要是协调组合上述三个物体，共同实现手册中的4个动作。

具体顺序为：命令行—Game的4个动作—分别调用其他对象的对应4个动作。

*Game*将下面所有的对象封装到自己的类中，对主程序或者说客户端只暴露自己的接口。

同时，对位置的合法性检查和对玩家的合法性检查都在这里面实现，游戏类型和对于不同类型的初始化也在*Game*类里实现。除此以外，*Game*中也需要提供基本的*observer*和*producer*。

|  |  |
| --- | --- |
| **属性** | **内容/说明** |
| Rep | gametype : GameType 游戏类型  board : Board 游戏棋盘  players : List<Player> 游戏玩家列表 |
| Mutable/Immutable | 由于list可变，为mutable类型 |
| AF | AF(gametype, board, players) = 一个可以进行棋类游戏的游戏类 |
| RI | gametype只能为Type.CHESS或Type.GO二者之一  board中只存储Piece对象或null  player中只存储player对象且不为空 |
| Safety from rep exposure | 所有的域都是private的，gametype和players为final，board不提供mutator方法，所有操作都在board内部进行  对所有涉及到向player或board对象传值的方法（比如take(), put()）都做了防御性拷贝或传入的是immutable对象 |

|  |  |
| --- | --- |
| **方法名称** | **具体实现** |
| public Game(GameType gametype) | 将参数gametype设置为游戏类型 |
| private void checkRep() | 检查棋盘内是否存在Piece或null以外的对象 |
| @Override  public boolean equals(Object otherobject) | 判断两个对象的类型，玩家，棋盘是否相等，分别调用对应的equals方法。 |
| @Override  public int hashCode() | 调用Objects的hash方法获得哈希码 |
| @Override  public String toString() | 生成描述游戏类型，玩家的字符串。 |
| public void initialGame(String p1, String p2) | 接受两个参数p1和p2分别表示两个玩家的名称。根据游戏类型的不同分为：  1. 围棋  初始化围棋棋盘，新建两个玩家对象，接着将棋子列表输入玩家的pieces中，将两个玩家添加入列表players中。  2. 象棋  初始化象棋盘，新建两个玩家对象，将玩家添加入players，设置两个棋子List P1，P2，填充入对应的国际象棋棋子，将P1，P2输入玩家pieces中，接着新建ChessBoard对象，将P1P2作为参数输入构造器。 |
| public List<Player> getPlayerList() | 返回玩家列表的拷贝 |
| public GameType getGameType() | 返回gametype |
| public Piece getPiece(Position position) | 调用board对象的getPiece方法 |
| public boolean put(Player player, Position position) | 调用Position静态方法检查合法性，判断玩家属于黑方或白方，生成不同的棋子，调用board的put方法，将棋子，位置输入，更新玩家棋子列表。然后添加记录到StringBuilder |
| public boolean move(Player player, Position source, Position target) | 调用Position静态方法检查合法性，判断该棋子是否属于玩家，调用board的move方法。然后添加记录到StringBuilder |
| public Piece take(Player player, Position position) | 调用Position静态方法检查合法性，判断该棋子是否不属于玩家，调用board的take方法，更新玩家棋子列表。然后添加记录到StringBuilder |
| public Piece capture(Player player, Position source, Position target) | 调用Position静态方法检查合法性，判断起始位置棋子是否不属于玩家，目标位置棋子是否属于玩家。调用board的capture方法，更新玩家棋子列表。然后添加记录到StringBuilder |
| public void getPiecesNum() | 获得玩家列表中的Player对象，获得其中的pieces棋子列表的大小输出 |
| public void getPositionInfo(Position pos) | 调用Position的静态方法进行合法性检查，然后调用board的getPiece方法进行判断。 |
| public void draw() | 遍历board进行绘制 |
| public void passTurn(Player player) | 在String Builder中添加跳过记录 |
| public String getHistory() | 返回String Builder中所有的历史纪录 |

### 主程序MyChessAndGoGame设计/实现方案

**设计：**

MyChessAnsGoGame作为主程序，代表用户端向ADT发起操作，因此MyChessAnsGoGame需要一个主程序入口main()和一个启动方法start()。main方法中，通过接收命令行输入选择游戏类型，建立不同的游戏对象。然后接收两个玩家的名称，对游戏对象进行初始化。start方法控制整个游戏的流程，用户在命令行的输入映射到ADT的方法的操作在start中执行。

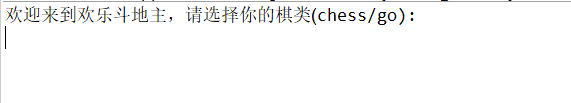
**实现方案：**

main函数

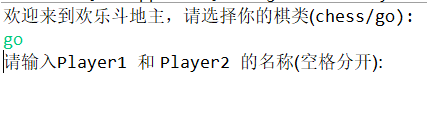
调用start方法

start函数

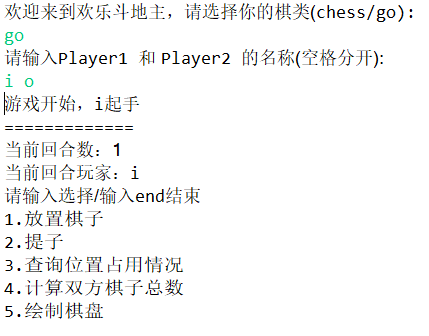
首先通过命令行选择游戏类型



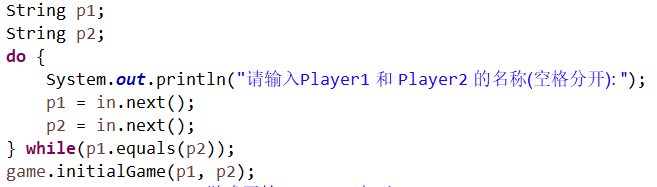
比如选择围棋go



接着输入玩家姓名



在输入玩家名称后，start函数中会调用game的initialGame方法进行初始化，该方法内会根据game当前的类型选择不同的初始化方式。



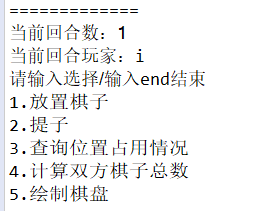
main函数中game初始化



initialGame部分源码

接着开始游戏。

start函数每一个回合都会绘制菜单和显示当前回合的玩家和回合数。



用户通过控制台输入命令，函数通过Scanner接收并解析，通过switch控制结构跳转到对应的模块。

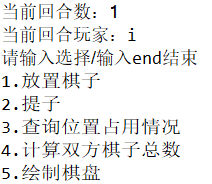


部分源码

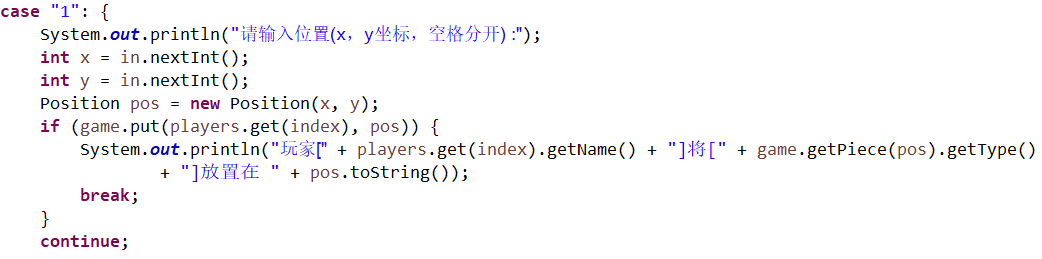
下面具体介绍如何将命令行输入映射到ADT的操作。

对于围棋，一共有两种操作：put和take。

围棋的游戏界面为

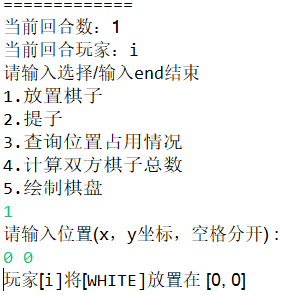


玩家在进行放置棋子操作时，输入 1 。switch结构会跳转到case 为1的代码块中，执行放置棋子的操作。

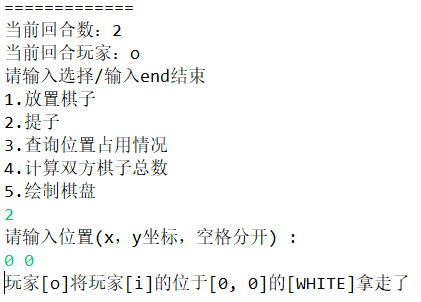


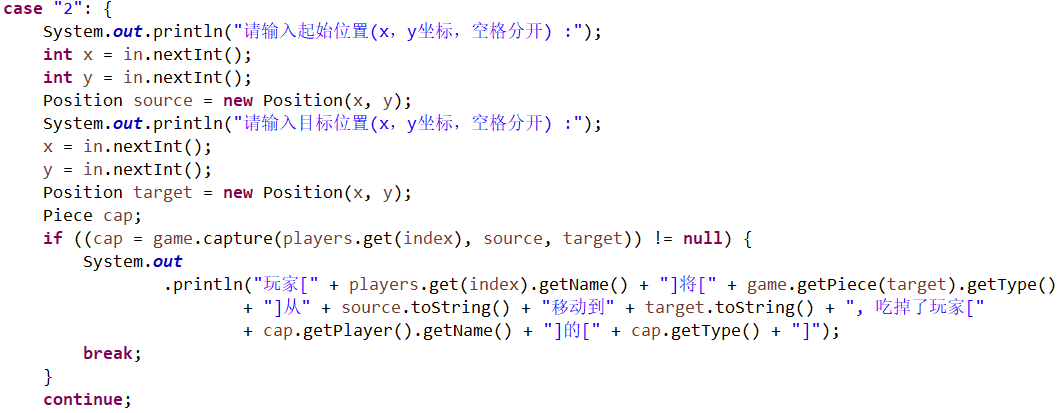
在源码中可以看到case代码块中调用了game的put方法，put方法回返回一个boolean，如果为true，说明放置成功。如果失败则继续当前回合，用户可以选择其他操作。

接下来，游戏会提示用户输入要放置的坐标，输入完成后控制台会输出相关信息。



如果用户选择输入2.则会进行提子操作，游戏会提示用户输入坐标，对输入坐标检查后，调用game的take方法进行提子操作。

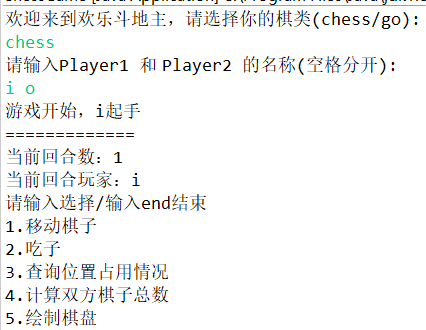


在对应的源码中**用户的输入映射到了case2中game的take方法中**。

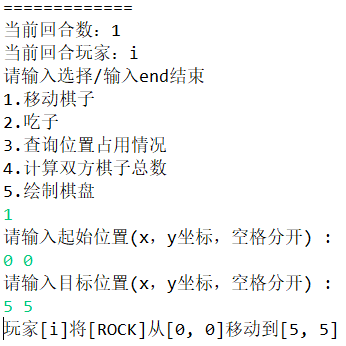
若操作成功，控制台会输出一串操作信息。

对于国际象棋，用户的操作有move和capture。

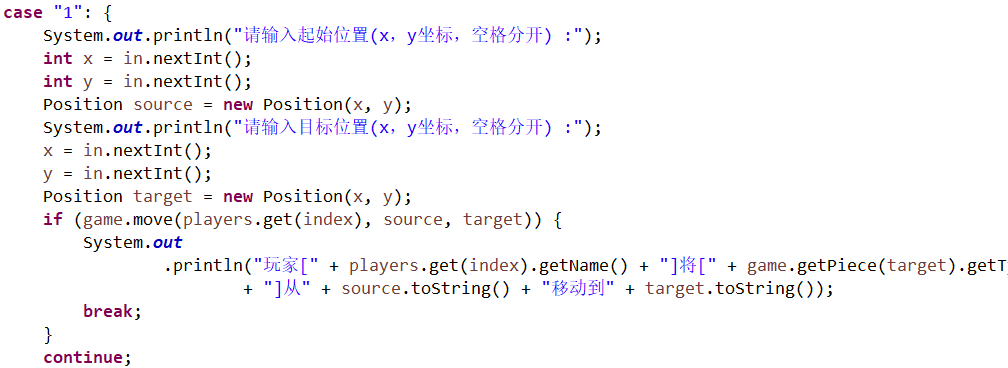
象棋的界面为



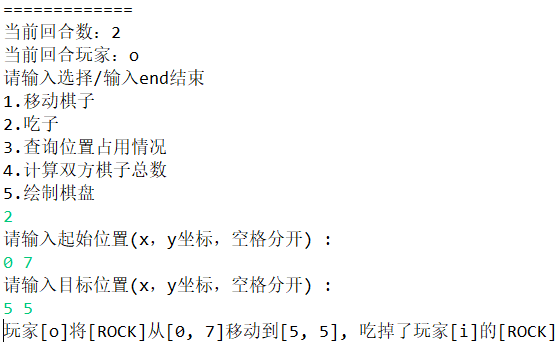
用户在操作时，输入1，则进行move操作，此时游戏会提示用户输入起始位置和目标位置，之后会对位置进行检查，如果检查合法，则会输出操作信息。



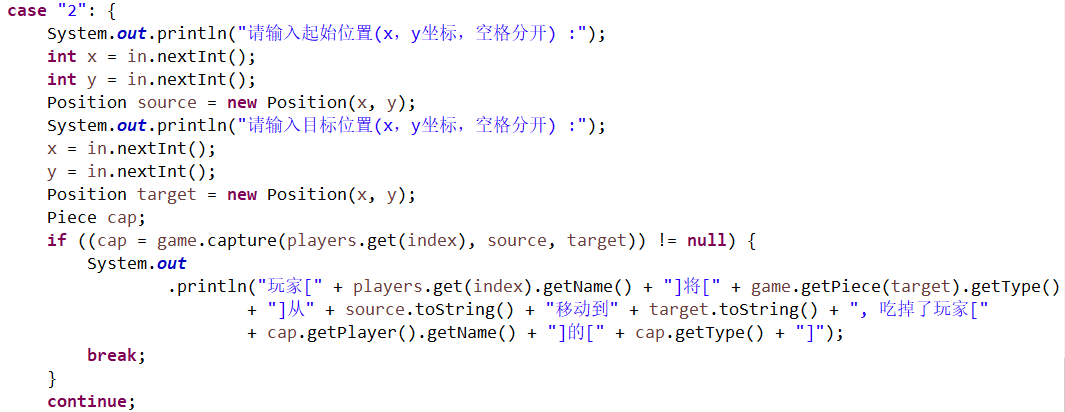
在对应源码中，**用户输入的命令行被映射到了case1中**，随后输入的两个坐标被作为起始和目标位置输入到game的move方法中，如果move方法操作成功，则会控制台输出操作信息。



如果用户输入2，则会进行吃子，capture操作，游戏会提示用户输入起始和目标位置，输入之后进行位置检查，若合法则输出操作信息。



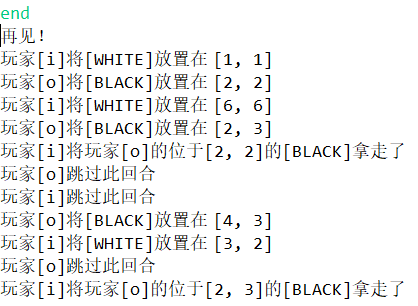
对应源码中，**用户输入被映射到case2**，随后输入的坐标作为起始和目标位置输入到game的capture方法中。



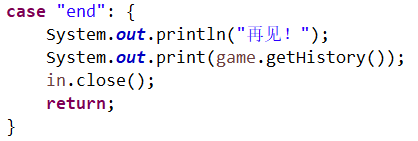
除了上述操作，还有两个通用操作1. 查询双方棋子数量2. 查询某位置的占用情况，这两个操作同时通过switch结构将输入映射到case3和case4中。

如果过用户选择跳过该回合，输入6后，程序跳转到case6中，此时调用game记录跳过步骤。

如果用户输入end选择结束游戏，程序跳转到case “end”，首先会调用game的getHistory方法获得历史记录然后输出，接着return返回main函数。



退出界面



源代码

### ADT和主程序的测试方案

**Piece的测试方案：**

由于Piece为immutable类型，直接对其父类测试即可。

|  |  |
| --- | --- |
| **方法** | **策略/划分** |
| getPlayer() | 构造Player测试  piece = produced by Piece() |
| getType() | 构造Type测试  piece = produced by Piece() |

**Action的测试方案：**

Action作为接口，无法直接测试，实现其接口的类为Board。Action中声明了put，move，take，cpature方法，为了对这些方法进行测试，先在ActionTest抽象类中写好测试函数，每个函数的实例都通过抽象方法emptyInstance获得，这个方法随后在Board类中实现。

|  |  |
| --- | --- |
| **方法** | **策略/划分** |
| getPiece() | position: 存在棋子的合法位置，不存在棋子的合法位置  board = produced by Board(), produced by put() |
| put() | piece: 普通棋子  position: 不存在棋子的合法位置  board,piece = produced by Board(), observed by getPiece() |
| take() | position: 存在棋子的合法位置  board，piece = produced by Board(), produced by put(),  observed by getPiece() |
| move() | source: 存在棋子的合法位置  target: 不存在棋子的合法位置  board,piece = produced by Board(), produced by put(),  observed by getPiece() |
| capture() | source: 存在棋子的合法位置target： 存在棋子的合法位置  board,piece = produced by Board(), produced by put(),  observed by getPiece() |

**Board测试方案：**

Board实现了Action接口，同时添加了新的方法。BoardTest继承自ActionTest，需要对emptyInstance方法实现。同时添加一些新的测试函数。

|  |  |
| --- | --- |
| **方法** | **策略/划分** |
| Board() | size: >0, =0 |
| getSize() | size: >0, =0  size = produced by Board() |
| toString() | 棋盘 |
| equals() | 空棋盘，带棋子的棋盘 |

**ChessBoard测试方案**：

ChessBoardTest继承自BoardTest，由于重写了父类的一些方法，因此需要对父类测试方法重写。该类中需要取消put和take方法测试，因为ChessBoard不允许这些操作。

|  |  |
| --- | --- |
| **方法** | **策略/划分** |
| 同父类 |  |

**GoBoard测试方案：**

GoBoardTest继承自BoardTest，由于重写了父类的一些方法，因此需要对父类测试方法重写。该类中需要取消move和capture方法测试，因为GoBoard不允许这些操作。

|  |  |
| --- | --- |
| **方法** | **策略/划分** |
| 同父类 |  |

**Player测试方案：**

直接测试

|  |  |
| --- | --- |
| **方法** | **策略/划分** |
| setPiece() | 通过传入List并修改来测试防御性拷贝的能力  pieces = observed by getPiece() |
| getPiece() | 通过获得List再修改测试  pieces = produced by setPiece() |
| clone() | 通过拷贝获取备份，对备份进行修改，测试防御性拷贝能力 |

**Position测试方案：**

直接测试

|  |  |
| --- | --- |
| **方法** | **策略/划分** |
| Position() | x: >0, 0, <0  y: >0, 0, <0  position = observed by getX(),getY() |
| get() | position = produced by Position() |
| checkPutPosition() | position:  x: 棋盘内位置，棋盘外位置  y: 棋盘内位置，棋盘外位置  board: 位置存在棋子，位置不存在棋子 |
| checkTakePosition() | position:  x: 棋盘内位置，棋盘外位置  y: 棋盘内位置，棋盘外位置  position = produced by Position()  board: 位置存在棋子，位置不存在棋子  board = produced by Board(), produced by put() |
| checkMovePosition() | position:  x: 棋盘内位置，棋盘外位置  y: 棋盘内位置，棋盘外位置  起始位置和目标位置相同  position = produced by Position()  board: 起始位置存在棋子，起始位置不存在棋子  目标位置存在棋子, 目标位置不存在棋子  board = produced by Board(), produced by put() |
| checkCapturePosition() | position:  x: 棋盘内位置，棋盘外位置  y: 棋盘内位置，棋盘外位置  起始位置和目标位置相同  position = produced by Position()  board: 起始位置存在棋子，起始位置不存在棋子  目标位置存在棋子, 目标位置不存在棋子  board = produced by Board(), produced by put() |

**Game测试方案：**

由于*Game*会对位置，玩家等信息检查，所以比较复杂，也是直接测试。

|  |  |
| --- | --- |
| **方法** | **策略/划分** |
| put() | player: 该回合玩家，非该回合玩家  pos: 合法位置，非法位置  位置存在棋子，位置不存在棋子  board = produced by Board() |
| move() | player: 该回合玩家  source: 合法位置，非法位置  该位置不存在棋子，位置存在棋子  该位置棋子不属于玩家，该位置棋子属于玩家  target: 合法位置，非法位置  该位置已经存在棋子，该位置不存在棋子  两位置相同  board = produced by Board(), put() |
| take() | player: 该回合玩家  position: 合法位置，非法位置  位置存在棋子，位置不存在棋子  该位置棋子不属于玩家，该位置棋子属于玩家  board = produced by Board(), put() |
| capture() | player: 该回合玩家  source: 合法位置，非法位置  位置存在棋子，位置不存在棋子  该位置棋子不属于玩家，该位置棋子属于玩家  target: 合法位置，非法位置  该位置不存在棋子，该位置存在棋子  该位置棋子不属于玩家，该位置棋子属于玩家  两位置相同  board = produced by Board(), put() |

**ChessGame测试方案：**

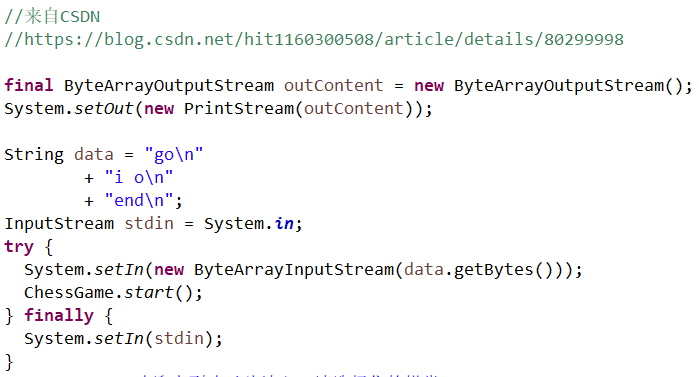
通过模拟一次完整的下棋过程，分别对围棋和国际象棋单独测试。其中使用*ByteArrayOutputStream*和*ByteArrayInputStream*模拟控制台输出输入。

*setIn*方法可以对输入进行重定向，将输入从*system.in*重定向为字符串*String*，然后通过*System.setOut*方法将输出写入*outContent*中。

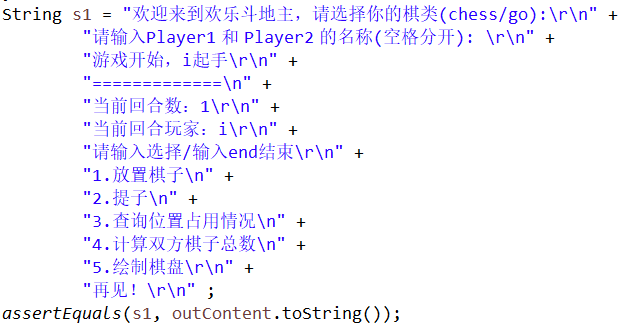
通过比较*outContent*的内容，确定游戏能够正常的输出。

**测试过程：**

重定向输出和输入



比较outContent



## Multi-Startup Set (MIT)

请自行设计目录结构。

注意：该任务为选做，不评判，不计分。

# 实验进度记录

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 3-11 | 20：00-22：00 | 完成Graph测试和一个具体实现 | 按计划完成 |
| 3-12 | 18：00-22：30 | 完成另一个具体实现和单元测试 | 测试未完成 |
| 3-13 | 21：00-22：20 | 完成测试和poem的单元测试 | poem测试未完成 |
| 3-15 | 20：00-22：40 | 完成poem具体实现 | 按计划完成 |
| 3-16 | 15：00-17：00 | 完成P2 | 按计划完成 |
| 3-18 | 13：40-15：30 | 设计P3 ADT | 完成一半 |
| 3-19 | 20：00-22：00 | 继续设计ADT | 完成 |
| 3-20 | 20：00-22：00 | 写P3一半的spec | 完成 |
| 3-21 | 14：00-17：30 | 写P3的Board及子类单元测试 | 完成 |
| 3-22 | 18：00-22：00 | 写Board及子类实现和实现Position | 完成 |
| 3-23 | 15：00-23：00 | 写Player和Piece的单元测试及spec | 单元测试没写完 |
| 3-25 | 13：00-15：00 | 写Game的spec及实现 | 完成 |
| 3-26 | 20：00-22：00 | 写ChessGame实现及单元测试 | 没完成 |
| 4-4 | 13：00-22：00 | 写报告及完成ChessGame | 没完成报告 |
| 4-5 | 20：00-23：00 | 写报告 | 完成 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| RI AF checkRep不会写？ | 上课听课，课后查阅相关阅读材料解决。 |
| P3的Action接口应该由哪个类实现？ | 为了保持Player和Piece的不变性，减少复杂性，最后用Board来实现。 |
| P3单元测试中Action如何测试？ | 仿照P1中Graph测试方法，建立测试Action的抽象类，让Board的测试继承抽象类。 |
| 如何对控制台输入输出进行测试从而实现对控制台程序的测试？ | 查阅博客，最后采用System.setIn 和 System.setOut重定向输入输出流。通过outContent保存控制台输出。 |
| 如何避免Board子类调用不属于它的方法？ | 重新覆盖不允许的操作，令其抛出异常 |

# 实验过程中收获的经验、教训、感想

## 实验过程中收获的经验和教训

**经验：**

使用ADT设计时需要抽取出物体的一般特征，高层次的ADT不应具体很具体的物体特征或实现，随着层次的逐渐下沉，ADT也应逐渐具体化。

对于通用的行为，可以使用接口来对类进行约束，这样可以保持类行为的一致性。

接口位于最高层次，抽象类位于中间层次，类位于下层，子类依此类推。

如果能避免使用数组，就不要使用数组

**教训：**

先写spec，再写单元测试，最后写具体实现！

ADT之间尽量仅存在单向关系，减少双向关系的出现，杜绝交叉关系。

ADT之间的依赖性应减少，功能方面减少重合部分。

如果一个类可以设计为immutable类型，就不要设计为mutable类型，因为设计者永远不知道哪个地方会存在表示泄露。

## 针对以下方面的感受

1. 面向ADT的编程和直接面向应用场景编程，你体会到二者有何差异？

ADT的通用性更强，但是相应的设计难度也更大。直接面向应用的编程非常快速，但是拓展性很差，一旦需求变动，最严重的情况是重构。

1. 使用泛型和不使用泛型的编程，对你来说有何差异？

泛型的使用使得程序拓展性更强，使用泛型让我感觉很方便。

1. 在给出ADT的规约后就开始编写测试用例，优势是什么？你是否能够适应这种测试方式？

优势在于设计者不会受到自己实现的影响，有意漏掉一些意外情况。也就是，规约完成后就写测试用例相当于设计者此时是作为客户来对程序进行测试。同时，此时写测试，规约与测试具有很强的联系，若实现代码后编写，需要再看一遍规约，容易出现疏漏。

目前还在适应中，脱离代码设计规约还是比较难以适应的，因为这样太抽象，很难深入考虑到一些细节。

1. P1设计的ADT在多个应用场景下使用，这种复用带来什么好处？

带来很强的拓展性，抽象出的ADT具有“以不变应万变”的能力。

1. P3要求你从0开始设计ADT并使用它们完成一个具体应用，你是否已适应从具体应用场景到ADT的“抽象映射”？相比起P1给出了ADT非常明确的rep和方法、ADT之间的逻辑关系，P3要求你自主设计这些内容，你的感受如何？

起初有点不适应，设计出的ADT关系混乱不明确，经过多次的迭代，才逐渐明确各个主要ADT的设计。从填充到从0实现，中间差距很大。

1. 为ADT撰写specification, invariants, RI, AF，时刻注意ADT是否有rep exposure，这些工作的意义是什么？你是否愿意在以后编程中坚持这么做？

这些相当于一份保证书。保证自己的设计具有安全性，同时帮助自己明确ADT的含义。对于将来再次复用时，能够快速抓住重点。对于其他编码人员，能够确认这份代码的安全性，使得调用或修改时有保证和约束。节省时间，提高编码效率。

愿意。

1. 关于本实验的工作量、难度、deadline。

工作量很大，但是难度与lab1相比并不是增加的很多，与lab3相比更是简单很多，毕竟lab3手册都看不明白。ddl时间是充裕的。

1. 《软件构造》课程进展到目前，你对该课程有何体会和建议？

这门课不注重与编程的技巧和算法等细节的东西，而是着重于系统上，工程上的成体系的理论。这门课将编程作为一系列环节中的一环，让我们从整个软件的生命周期的视角审视编程这件事。使得视野得到提高。但是问题也在于此，课堂上理论讲了很多，但是缺少了很多例子，希望课程可以提供更多具体的例子或者将例子集中编为手册挂在网上，让同学们学习。