

**2019年春季学期  
计算机学院《软件构造》课程**

**Lab 3实验报告**

|  |  |
| --- | --- |
| 姓名 | 王润哲 |
| 学号 | 1170801403 |
| 班号 | 1703008 |
| 电子邮件 | 1733108435@qq.com |
| 手机号码 | 13030087519 |

**目录**

[1 实验目标概述 1](#_Toc3922069)

[2 实验环境配置 1](#_Toc3922070)

[3 实验过程 1](#_Toc3922071)

[3.1 待开发的三个应用场景 1](#_Toc3922072)

[3.2 基于语法的图数据输入 1](#_Toc3922073)

[3.3 面向复用的设计：CircularOrbit<L,E> 1](#_Toc3922074)

[3.4 面向复用的设计：Track 2](#_Toc3922075)

[3.5 面向复用的设计：L 2](#_Toc3922076)

[3.6 面向复用的设计：PhysicalObject 2](#_Toc3922077)

[3.7 可复用API设计 2](#_Toc3922078)

[3.8 图的可视化：第三方API的复用 2](#_Toc3922079)

[3.9 设计模式应用 2](#_Toc3922080)

[3.10 应用设计与开发 2](#_Toc3922081)

[3.10.1 TrackGame 2](#_Toc3922082)

[3.10.2 StellarSystem 2](#_Toc3922083)

[3.10.3 AtomStructure 2](#_Toc3922084)

[3.10.4 PersonalAppEcosystem 2](#_Toc3922085)

[3.10.5 SocialNetworkCircle 2](#_Toc3922086)

[3.11 应对应用面临的新变化 2](#_Toc3922087)

[3.11.1 TrackGame 3](#_Toc3922088)

[3.11.2 StellarSystem 3](#_Toc3922089)

[3.11.3 AtomStructure 3](#_Toc3922090)

[3.11.4 PersonalAppEcosystem 3](#_Toc3922091)

[3.11.5 SocialNetworkCircle 3](#_Toc3922092)

[3.12 Git仓库结构 3](#_Toc3922093)

[4 实验进度记录 3](#_Toc3922094)

[5 实验过程中遇到的困难与解决途径 3](#_Toc3922095)

[6 实验过程中收获的经验、教训、感想 4](#_Toc3922096)

[6.1 实验过程中收获的经验和教训 4](#_Toc3922097)

[6.2 针对以下方面的感受 4](#_Toc3922098)

# 实验目标概述

根据实验手册简要撰写。

本次实验覆盖课程第 3、5、6 章的内容，目标是编写具有可复用性和可维护性的软件，主要使用以下软件构造技术：

⚫ 子类型、泛型、多态、重写、重载

⚫ 继承、代理、组合

⚫ 常见的 OO 设计模式

⚫ 语法驱动的编程、正则表达式

⚫ 基于状态的编程

⚫ API 设计、API 复用

本次实验给定了五个具体应用（径赛方案编排、太阳系行星模拟、原子结构

可视化、个人移动 App 生态系统、个人社交系统），学生不是直接针对五个应用

分别编程实现，而是通过 ADT 和泛型等抽象技术，开发一套可复用的 ADT 及其实现，充分考虑这些应用之间的相似性和差异性，使 ADT 有更大程度的复用（可复用性）和更容易面向各种变化（可维护性）。

# 实验环境配置

简要陈述你配置本次实验所需环境的过程，必要时可以给出屏幕截图。

特别是要记录配置过程中遇到的问题和困难，以及如何解决的。

本次实验环境配置与lab2相同

在这里给出你的GitHub Lab3仓库的URL地址（Lab3-学号）。

<https://github.com/1170801403/lab3_1170801403>

（之前的lab3仓库被误删了，这个是自己建立的一个仓库）

# 实验过程

请仔细对照实验手册，针对每一项任务，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！）。

## 待开发的三个应用场景

首先请列出你要完成的具体应用场景（至少3个，1和2中选一，3必选，4和5中选一，鼓励完成更多的应用场景）。

* 径赛方案编排
* 原子结构可视化
* 个人社交系统

分析你所选定的多个应用场景的异同，理解需求：它们在哪些方面有共性、哪些方面有差异。

相同点：三个应用都需要实现若干圆形的轨道，实现圆形轨道上的物体，物体之间的关系以及这些所有这些对象之间的变化。

不同点：田径比赛中没有中心物体；轨道物体之间不需要关系；轨道物体与中心物体之间不需要关系；但是田径比赛需要分组,每分一组就是实现一个新的轨道系统；需要实现两种不同的比赛编排方案；同时需要支持为运动员更换赛道，更换组。

原子结构的可视化需要一个中心点物体，3.12的扩展版本要求将中心点物体表示成质子和中子；但是轨道系统只需要实现一个；轨道系统上的物体没有差别，都是电子；由于现实世界中无法准确测得电子的位置，因此无需考虑电子的绝对位置；但是需要实现电子在不同轨道之间的跃迁。

个人社交系统是最复杂的一个系统，要求从社交日志中恢复社交网络结构，判定每个用户在哪个轨道上；计算通过某个好友能间接认识多少好友，同时考虑亲密度因素；增加或删除某一条社交关系之后重新调整社交网络结构；计算社交网络上任意两个人之间的逻辑距离

## 基于语法的图数据输入

由于本次实验需要从文件中读入数据建立轨道，因此需要使用正则表达式读入文件，不同的轨道系统需要使用不同的正则表达式，通过将正则表达式进行分组从而更容易的从文件中得到所需的数据，在读入数据的同时对相应的数据结构进行初始化，这些数据结构在建立轨道时需要用到，即读入文件和建立轨道是两个不同的函数，便于之后的扩展。代码如下：

田径轨道：

**public** **void** creatingTrackFromFiles(String name)// 从外部文件读取数据构造轨道系统对象

{

String txt = **new** String();

String re1 = "(Athlete)"; // Word 1

String re2 = "(\\s+)"; // White Space 1

String re3 = "(:)"; // Any Single Character 1

String re4 = "(:)"; // Any Single Character 2

String re5 = "(=)"; // Any Single Character 3

String re6 = "(\\s+)"; // White Space 2

String re7 = "(<)"; // Any Single Character 4

String re8 = "((?:[a-z][a-z]+))"; // Word 2,姓名

String re9 = "(,)"; // Any Single Character 5

String re10 = "(\\d+)"; // Integer Number 1，编号

String re11 = "(,)"; // Any Single Character 6

String re12 = "((?:[a-z][a-z]+))"; // Word 3，国籍

String re13 = "(,)"; // Any Single Character 7

String re14 = "(\\d+)"; // Integer Number 2，年龄

String re15 = "(,)"; // Any Single Character 8

String re16 = "(\\d{1,2}\\.\\d{2}+)"; // Float 1，最佳成绩

String re17 = "(>)"; // Any Single Character 9

**try**

{

txt = **new** String(Files.*readAllBytes*(Paths.*get*("txt/" + name + "txt")));

}

**catch** (IOException e)

{

// **TODO**: handle exception

e.printStackTrace();

}

Pattern p = Pattern.*compile*(re1 + re2 + re3 + re4 + re5 + re6 + re7 + re8 + re9 + re10 + re11 + re12 + re13

+ re14 + re15 + re16 + re17, Pattern.***CASE\_INSENSITIVE*** | Pattern.***DOTALL***);

Matcher m = p.matcher(txt);

**while** (m.find())

{

String word1 = m.group(8);// 姓名

String int1 = m.group(10);// 编号

String word2 = m.group(12);// 国籍

String int2 = m.group(14);// 年龄

String float1 = m.group(16);// 最佳成绩

trackE1 temp = factory.manufactureE(word1, Integer.*parseInt*(int1), word2, Integer.*parseInt*(int2),

Float.*parseFloat*(float1));

athlete.add(temp);// parseInt和parseFloat都是构造常量

}

String re18 = "(Game)"; // Word 1

String re19 = "(\\s+)"; // White Space 1

String re20 = "(:)"; // Any Single Character 1

String re21 = "(:)"; // Any Single Character 2

String re22 = "(=)"; // Any Single Character 3

String re23 = "(\\s+)"; // White Space 2

String re24 = "(\\d+)"; // Integer Number 1

Pattern p1 = Pattern.*compile*(re18 + re19 + re20 + re21 + re22 + re23 + re24,

Pattern.***CASE\_INSENSITIVE*** | Pattern.***DOTALL***);

Matcher m1 = p1.matcher(txt);

**while** (m1.find())

{

String int11 = m1.group(7);

playType = Integer.*parseInt*(int11);// 得到比赛种类

}

String re25 = "(NumOfTracks)"; // Word 1

String re26 = "(\\s+)"; // White Space 1

String re27 = "(:)"; // Any Single Character 1

String re28 = "(:)"; // Any Single Character 2

String re29 = "(=)"; // Any Single Character 3

String re30 = "(\\s+)"; // White Space 2

String re31 = "(\\d+)"; // Integer Number 1

Pattern p2 = Pattern.*compile*(re25 + re26 + re27 + re28 + re29 + re30 + re31,

Pattern.***CASE\_INSENSITIVE*** | Pattern.***DOTALL***);

Matcher m2 = p2.matcher(txt);

**while** (m2.find())

{

String int12 = m2.group(7);

trackNumber = Integer.*parseInt*(int12);// 得到轨道数目

}

}

原子轨道：

**public** **void** creatingTrackFromFiles(String name)// 从外部文件读取数据构造轨道系统对象

{

String txt = **new** String();

**try**

{

txt = **new** String(Files.*readAllBytes*(Paths.*get*("txt/" + name + "txt")));

}

**catch** (IOException e)

{

// **TODO**: handle exception

e.printStackTrace();

}

String fre9 = "(NumberOfTracks)"; // Word 3

String fre10 = "(\\s+)"; // White Space 4

String fre11 = "(:)"; // Any Single Character 4

String fre12 = "(:)"; // Any Single Character 5

String fre13 = "(=)"; // Any Single Character 6

String fre14 = "(\\s+)"; // White Space 5

String fre15 = "(\\d+)"; // Integer Number 1

Pattern fp = Pattern.*compile*(fre9 + fre10 + fre11 + fre12 + fre13 + fre14 + fre15,

Pattern.***CASE\_INSENSITIVE*** | Pattern.***DOTALL***);

Matcher fm = fp.matcher(txt);

System.***out***.println(fre15);

**if** (fm.find())// 顶多匹配到一次，只需要用if

{

trackNumber2 = Integer.*parseInt*(fm.group(7));

}

String re1 = "(ElementName)"; // Word 1

String re2 = "(\\s+)"; // White Space 1

String re3 = "(:)"; // Any Single Character 1

String re4 = "(:)"; // Any Single Character 2

String re5 = "(=)"; // Any Single Character 3

String re6 = "(\\s+)"; // White Space 2

String re7 = "((?:[a-z][a-z]+))"; // Word 2

String re8 = "(\\s+)"; // White Space 3

String re9 = "(NumberOfTracks)"; // Word 3

String re10 = "(\\s+)"; // White Space 4

String re11 = "(:)"; // Any Single Character 4

String re12 = "(:)"; // Any Single Character 5

String re13 = "(=)"; // Any Single Character 6

String re14 = "(\\s+)"; // White Space 5

String re15 = "(\\d+)"; // Integer Number 1

String re16 = "(\\s+)"; // White Space 6

String re17 = "(NumberOfElectron)"; // Word 4

String re18 = "(\\s+)"; // White Space 7

String re19 = "(:)"; // Any Single Character 7

String re20 = "(:)"; // Any Single Character 8

String re21 = "(=)"; // Any Single Character 9

String re22 = "(\\s+)"; // White Space 8

// re22之后出现轨道数\电子数

String re23 = "(\\d+)"; // Integer Number 2

String re24 = "(\\/)"; // Any Single Character 10

// String re25="(\\d+/+d)"; // Integer Number 3

String re25 = "(\\d+)"; // Integer Number 3

String re26 = "(;)"; // Any Single Character 11

String re = re1 + re2 + re3 + re4 + re5 + re6 + re7 + re8 + re9 + re10 + re11 + re12 + re13 + re14 + re15 + re16

+ re17 + re18 + re19 + re20 + re21 + re22;

String numRe = re23 + re24 + re25 + re26;

**for** (**int** i = 0; i < trackNumber2 - 1; i++)

{

re = re + numRe;

}

String re27 = "(\\d+)"; // Integer Number 10

String re28 = "(\\/)"; // Any Single Character 18

String re29 = "(\\d+)"; // Integer Number 11

String finalNumRe = re27 + re28 + re29;

re = re + finalNumRe;

Pattern p = Pattern.*compile*(re, Pattern.***CASE\_INSENSITIVE*** | Pattern.***DOTALL***);

Matcher m = p.matcher(txt);

**if** (m.find())// 顶多匹配到一次，只需要用if

{

String word2 = m.group(7);// 原子核名称

central = factory.manufactureL(word2);

String int1 = m.group(15);// 原子核外轨道数目

trackNumber2 = Integer.*parseInt*(int1);// 确定核外轨道数目

**for** (**int** i = 0; i < trackNumber2 - 1; i++)

{

String temp = m.group(22 + i \* 4 + 3);

trackObjectNumber.put(i, Integer.*parseInt*(temp));// 通过轨道号查询该轨道上有多少个电子

}

// 结尾没有分号

String temp1 = m.group(22 + (trackNumber2 - 1) \* 4 + 3);

trackObjectNumber.put(trackNumber2 - 1, Integer.*parseInt*(temp1));

}

}

社交网络：

**public** **void** creatingTrackFromFiles(String name)

{

String txt = **new** String();

**try**

{

txt = **new** String(Files.*readAllBytes*(Paths.*get*("txt/" + name + "txt")));

}

**catch** (IOException e)

{

// **TODO**: handle exception

e.printStackTrace();

}

String re1 = "(CentralUser)"; // Word 1

String re2 = "(\\s\*)"; // White Space 1

String re3 = "(:)"; // Any Single Character 1

String re4 = "(:)"; // Any Single Character 2

String re5 = "(=)"; // Any Single Character 3

String re6 = "(\\s\*)"; // White Space 2

String re7 = "(<)"; // Any Single Character 4

String re8 = "((?:[a-z][a-z]\*[0-9]\*[a-z0-9]\*))"; // Word 2

String re9 = "(,)"; // Any Single Character 5

String re10 = "(\\s\*)";

String re11 = "(\\d+)"; // Integer Number 1

String re12 = "(,)"; // Any Single Character 6

String re13 = "(\\s\*)";

String re14 = "([a-z])"; // Any Single Word Character (Not Whitespace) 2

String re15 = "(>)"; // Any Single Character 7

Pattern p = Pattern.*compile*(

re1 + re2 + re3 + re4 + re5 + re6 + re7 + re8 + re9 + re10 + re11 + re12 + re13 + re14 + re15,

Pattern.***CASE\_INSENSITIVE*** | Pattern.***DOTALL***);

Matcher m = p.matcher(txt);

**if** (m.find())// 只会find一次

{

System.***out***.println("find central!");

String word1 = m.group(1);

String c1 = m.group(2);

String c2 = m.group(3);

String c3 = m.group(4);

String ws2 = m.group(5);

String c4 = m.group(6);

String word2 = m.group(7);

String c5 = m.group(8);// 姓名

String int1 = m.group(9);

String c6 = m.group(10);

String w2 = m.group(11);// 年龄

String c7 = m.group(12);

String c8 = m.group(14);// 性别

central = factory.manufactureL(c5, Integer.*parseInt*(w2), c8.charAt(0));// 构造中心点物体

}

// new socialL1(c5, Integer.parseInt(w2), c8.charAt(0))

// Friend里的人不一定和中心点人有关系

String fre1 = "(Friend)"; // Word 1

String fre2 = "(\\s\*)"; // White Space 1

String fre3 = "(:)"; // Any Single Character 1

String fre4 = "(:)"; // Any Single Character 2

String fre5 = "(=)"; // Any Single Character 3

String fre6 = "(\\s\*)"; // White Space 2

String fre7 = "(<)"; // Any Single Character 4

String fre8 = "((?:[a-z][a-z]\*[0-9]\*[a-z0-9]\*))"; // Alphanum 1

String fre9 = "(,)"; // Any Single Character 5

String fre10 = "(\\s\*)"; // White Space 1

String fre11 = "(\\d+)"; // Integer Number 1

String fre12 = "(,)"; // Any Single Character 6

String fre13 = "(\\s\*)"; // White Space 1

String fre14 = "([a-z])"; // Any Single Word Character (Not Whitespace) 1

String fre15 = "(>)"; // Any Single Character 7

Pattern fp = Pattern.*compile*(fre1 + fre2 + fre3 + fre4 + fre5 + fre6 + fre7 + fre8 + fre9 + fre10 + fre11

+ fre12 + fre13 + fre14 + fre15, Pattern.***CASE\_INSENSITIVE*** | Pattern.***DOTALL***);

Matcher fm = fp.matcher(txt);

System.***out***.println("start to find friend!");

**while** (fm.find())// 不止匹配一次

{

System.***out***.println("find friend!");

String fword1 = fm.group(1);

String fws1 = fm.group(2);

String fc1 = fm.group(3);

String fc2 = fm.group(4);

String fc3 = fm.group(5);

String fws2 = fm.group(6);

String fc4 = fm.group(7);

String falphanum1 = fm.group(8);// 姓名

String fc5 = fm.group(9);

String fint1 = fm.group(10);

String ffc6 = fm.group(11);// 年龄

String fw1 = fm.group(12);

String fc7 = fm.group(14);// 性别

socialE1 tempFriend = factory.manufactureE(falphanum1, Integer.*parseInt*(ffc6), fc7.charAt(0));

// new socialE1(falphanum1, Integer.parseInt(ffc6), fc7.charAt(0));

friend.put(falphanum1, tempFriend);// 通过名字找到人，初始化friend映射

// socialNetWork重写了父类中的方法

relationship.put(tempFriend, **new** HashSet<socialE1>());// 匹配的同时对“关系”进行初始化

intimacyFriend.put(falphanum1, **new** HashMap<String, Float>());// 初始化映射

}

String sre1 = "(SocialTie)"; // Word 1

String sre2 = "(\\s\*)"; // White Space 1

String sre3 = "(:)"; // Any Single Character 1

String sre4 = "(:)"; // Any Single Character 2

String sre5 = "(=)"; // Any Single Character 3

String sre6 = "(\\s\*)"; // White Space 2

String sre7 = "(<)"; // Any Single Character 4

String sre8 = "((?:[a-z][a-z]\*[0-9]\*[a-z0-9]\*))";

String sre9 = "(,)"; // Any Single Character 5

String sre10 = "(\\s\*)";

String sre11 = "((?:[a-z][a-z]\*[0-9]\*[a-z0-9]\*))"; // Alphanum 2

String sre12 = "(,)"; // Any Single Character 6

String sre13 = "(\\s\*)";

String sre14 = "([01](?:\\.{1}\\d{1,3}){0,1})"; // Float 1

// ([+-]?\\d\*\\.\\d+)(?![-+0-9\\.])

String sre15 = "(>)"; // Any Single Character 7

Pattern normalSp = Pattern.*compile*(sre1 + sre2 + sre3 + sre4 + sre5 + sre6 + sre7 + sre8 + sre9 + sre10 + sre11

+ sre12 + sre13 + sre14 + sre15, Pattern.***CASE\_INSENSITIVE*** | Pattern.***DOTALL***);

Matcher normalSm = normalSp.matcher(txt);

// 当这个while循环执行完毕之后，轨道系统就建立起来了

// addLErelationship：本质上是一个集合，不需要执行，在addTrackObject里执行

// addEErelationship：一个映射，映射的value值还是一个映射，仅限于轨道物体

// superIntimacyFriend: 存储第一层轨道上的人与中心物体之间的关系，以及亲密度

// intimacyFriend：仅限于轨道物体，跟中心物体无任何关系，其中key值的数量==addLErelationship中元素的数量==轨道上的物体总数

**while** (normalSm.find())

{

System.***out***.println("Find it!");

String normalSalphanum1 = normalSm.group(8);// person1

String normalSalphanum2 = normalSm.group(11);// person2

String normalSfloat1 = normalSm.group(14);// 亲密度

socialTie.add(**new** tie(normalSalphanum1, normalSalphanum2, Float.*parseFloat*(normalSfloat1)));

}

}

## 面向复用的设计：CircularOrbit<L,E>

所要实现的三个轨道系统具有共性方法，在CircularOrbit<L,E>接口中实现它们的共性方法例如增加轨道、增加物体、增加关系、获取相应的数据结构等，函数以及函数说明如下：

/\*\*

\* 增加轨道

\* 新建一个轨道对象(track)，并新建一个轨道到物体的映射

\*/

**public** **void** addTrack();//增加一条轨道

/\*

\* 删除一条轨道

\* @param 轨道的编号

\* 在存储轨道的数据结构中删除这条轨道，同时在轨道系统中删除这条轨道上的所有物体，

\* 这些物体所关联的关系也会被删除

\*/

**public** **void** deleteTrack(**int** number);//去除一条轨道

/\*\*

\* 增加中心点物体

\* **@param** cre 中心物体

\* 以给定的参数新建一个中心点物体类的实例

\*/

**public** **void** addCentralObject(L cre);//增加中心点物体

/\*\*

\* 在给定轨道上增加一个给定物体，需要处理异常情况如给定的轨道编号不存在。如果成功添加物体，

\* 则新建物体到轨道的映射，物体到位置（角度）的映射，以及物体间关系的映射

\* **@param** ob 轨道编号

\* **@param** t 轨道物体

\* **@return** 成功添加物体则返回true,否则返回false并给出错误提示

\*/

**public** **boolean** addTrackObject(E ob, **int** t);//向特定轨道上增加一个物体（不考虑物理位置）

/\*\*

\* 删除给定轨道上的给定物体，需要处理异常情况如给定的轨道编号不存在，给定物体不在给定轨道上。

\* 如果成功删除物体，则需同时删除轨道到该物体的映射，该物体到轨道的映射，该物体到位置（角度）的映射，

\* 该物体与轨道物体关联的关系，该物体与中心点物体关联的关系

\* **@param** ob 轨道编号

\* **@param** t 轨道物体

\* **@return** 成功删除物体则返回true,否则返回false并给出错误提示

\*/

**public** **boolean** deleteTrackObject(E ob, **int** t);// 向特定轨道上删除一个物体（不考虑物理位置）

/\*\*

\* 增加给定轨道物体与中心物体之间的关系，需要处理异常情况如给定物体不在系统中。

\* 如果轨道物体符合要求，则在相应的数据结构Map中新增一对映射即可

\* **@param** e2 轨道物体

\* **@return** 如果增加成功则返回true,否则返回false并给出错误提示

\*/

**public** **boolean** addLErelationship(E e2);//增加中心物体和轨道物体之间的关系

/\*\*

\* 增加两个给定轨道物体之间的关系，需要处理异常情况如给定物体不在系统中。

\* 如果轨道物体符合要求，则在相应的数据结构Map中新增一对映射即可

\* **@param** e1 轨道物体

\* **@param** e2 轨道物体

\* **@return** 如果增加成功则返回true,否则返回false并给出错误提示

\*/

**public** **boolean** addEErelationship(E e1,E e2);//增加两个轨道物体之间的关系

/\*\*

\* 删除两个给定轨道物体之间的关系，需要处理异常情况如给定物体不在系统中。

\* 如果轨道物体符合要求，则在相应的数据结构Map中删除一对映射即可

\* **@param** e1 轨道物体

\* **@param** e2 轨道物体

\* **@return** 如果删除成功则返回true,否则返回false并给出错误提示

\*/

**public** **boolean** deleteEErelationship(E e1,E e2);

/\*\*

\* 删除给定轨道物体与中心物体之间的关系，需要处理异常情况如给定物体不在系统中。

\* 如果轨道物体符合要求，则在相应的数据结构Map中删除一对映射即可

\* **@param** e2 轨道物体

\* **@return**

\*/

**public** **boolean** deleteLErelationship(E e2);

/\*\*

\* 从给定的文件中读入建立轨道系统必要的参数，并存储在相应的数据结构中，

\* 不同类型的轨道系统读入文件采用不同的方式

\* **@param** name 文件名称

\*/

**public** **void** creatingTrackFromFiles(String name);//从外部文件读取数据构造轨道系统对象

/\*\*

\* 给定轨道系统和轨道编号，将该物体从原有轨道移动到轨道编号对应的轨道上，需要

\* 处理异常情况如轨道物体不在轨道系统中，轨道编号超出范围等。移动成功后需要改变相应、

\* 的轨道到物体的映射、物体到轨道的映射

\* **@param** object 轨道物体

\* **@param** t 轨道编号

\* **@return** 如果跃迁成功则返回true,否则返回false并给出错误提示

\*/

**public** **boolean** transit (E object, **int** t);//原子结构中的电子可以跃迁，社交网络中的人的地位可以变化

/\*\*

\* 通过防御性克隆安全返回轨道系统中轨道物体到其所在位置的映射

\* **@return** 轨道物体到其所在位置的映射

\*/

**public** Map<E, Double> getAngle();

/\*\*

\* 通过防御性克隆安全返回轨道系统中轨道物体到其所在轨道的映射

\* **@return** 轨道物体到其所在轨道的映射

\*/

**public** Map<E, Integer> getObjectTrack();

/\*\*

\* 通过防御性克隆安全返回轨道系统中存储轨道物体之间关系的映射

\* **@return** 存储轨道物体之间关系的映射

\*/

**public** Map<E, Set<E>> getRelationship();

/\*\*

\* 通过防御性克隆安全返回轨道系统中轨道到其上所有的物体的映射

\* **@return** 轨道到其上所有的物体的映射

\*/

**public** Map<Integer, Set<E>> getTrackObject();

/\*\*

\* 通过防御性克隆安全返回轨道系统中存储轨道物体与中心点物体关系的映射

\* **@return** 存储轨道物体与中心点物体关系的映射

\*/

**public** Set<E> getLErelationship();

/\*\*

\* 通过防御性克隆安全返回轨道系统中存储轨道物体之间联系信息映射

\* **@return** 存储轨道物体之间联系信息映射

\*/

**public** List<tie> getSocialTie();

/\*\*

\* 返回轨道系统的中心点物体(不可变类型)

\* **@return** 轨道系统的中心点物体

\*/

**public** L getCentral();

接着完成ConcreteCircularObject类实现接口中的所有方法，ConcreteCircularObject作为三个具体轨道系统的父类，定义了若干数据结构以保存图中物体、关系、位置等信息：

**final** List<Track> physical = **new** ArrayList<Track>();// 列表编号表示轨道层号，轨道对象中包含物体E,未把Track当成占位符

**final** Map<Integer, Set<E>> trackObject = **new** HashMap<Integer, Set<E>>();// 轨道映射物体的列表

**final** Map<E, Integer> objectTrack = **new** HashMap<E, Integer>();// 物体映射轨道的列表

**final** Map<E, Set<E>> relationship = **new** HashMap<E, Set<E>>();// 存储轨道物体之间的关系

**final** Set<E> LErelationship = **new** HashSet<E>();// 存储中心点物体和轨道物体之间的关系，在社交网络中是指第一层人

**final** Map<E, Double> angle = **new** HashMap<E, Double>();// 存储每个物体的角度

// final Map<E, Integer> objectGroup = new HashMap<E, Integer>();// 物体到组别的映射

**final** physicalShelf shelf = **new** physicalShelf();

**final** trackFactory realTrackFactory = **new** trackFactory();

**final** List<tie> socialTie = **new** ArrayList<tie>();

注意到为最后一个对象List<tie> socialTie，其中tie类型是为关系定义的ADT，tie是不可变类型，其具体实现代码如下：

**public** **class** tie

{

**private** **final** String name1;

**private** **final** String name2;

**private** **final** **float** ini;

**public** tie(String name1, String name2, **float** ini)

{

**this**.name1 = name1;

**this**.name2 = name2;

**this**.ini = ini;

}

**public** String getName1()

{

**return** name1;

}

**public** String getName2()

{

**return** name2;

}

**public** **float** getIni()

{

**return** ini;

}

}

ConcreteCircularObject中各方法的具体实验以及正确性已在实验课上验收过，此处不再赘述，下面从ConcreteCircularObject派生出三个具体轨道系统类：trackGame（田径比赛）,atomStructure（原子轨道）,socialNetWork（社交网络）。

其中，trackGame完全继承ConcreteCircularObject，不做任何改动，但是田径比赛需要分组，所以需要若干各轨道系统，即需要若干个trackGame，因此需要定义一个管理trackGame的类functionTrackGame，functionTrackGame负责读入文件，产生不同的比赛方案（即产生不同数目的trackGame，同时采用不同的策略让运动员在轨道上形成排列，此处需要用到strategy模式），调整运动员之间的轨道。

atomStructure增加了若干对象：

**private** Map<Integer, Integer> trackObjectNumber = **new** HashMap<Integer, Integer>();// 每个轨道上有多少个电子

**int** trackNumber2 = 0;// 有多少个轨道

**private** atomFactory factory = **new** atomFactory();

同时负责读入文件，建立atomStructure轨道系统，完成电子跃迁。

socialNetWork中也增加了若干对象：

**private** **final** Map<String, socialE1> friend = **new** HashMap<String, socialE1>();// 通过名字找到人

**private** **final** Map<String, Map<String, Float>> intimacyFriend = **new** HashMap<String, Map<String, Float>>();// 存储轨道人间的亲密度

**private** **final** Map<String, Float> superIntimacyFriend = **new** HashMap<String, Float>();// 存储与中心人的亲密度

**private** **final** List<String> temp1 = **new** ArrayList<String>();// 存储例外情况

**private** **final** List<String> temp2 = **new** ArrayList<String>();// 存储例外情况

**private** **final** List<Float> temp3 = **new** ArrayList<Float>();// 存储例外情况

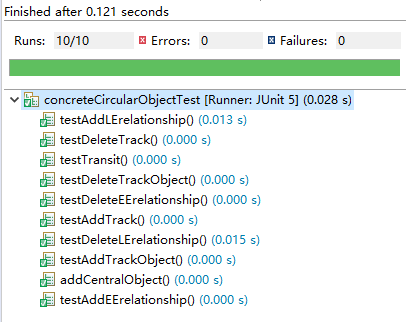
**private** socialFactory factory = **new** socialFactory();//工厂方法

socialNetWork负责从文件中读入数据，建立轨道，求出某个朋友的“信息扩散度”，增加\删除轨道物体与轨道物体之间的关系，增加\删除轨道物体与中心物体之间的关系。

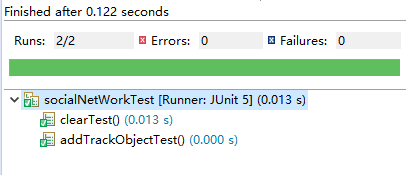
三个具体类中的方法实现已在实验课上向助教验收，此处不再赘述。接下来是Junit Test，由于具体类中的很多方法都是通过父类中的方法聚合而成的，同时，部分方法在读入文件并可视化的时候已经检验了其正确性，所以Junit Test主要集中在对ConcreteCircularObject中方法的测试以及对中心物体、轨道物体、轨道、关系的测试。

测试的策略总结起来就是新建一个轨道系统，然后以特定的参数调用被测试的方法，判断它能否处理异常以及返回正确结果，具体代码不再赘述，测试的结果如下：

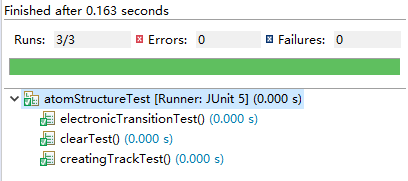
对轨道系统中方法的测试：



图（1）ConcreteCircularObject中具体函数的测试结果

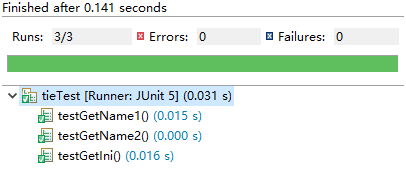


图（2）socialNetWork中具体函数的测试结果



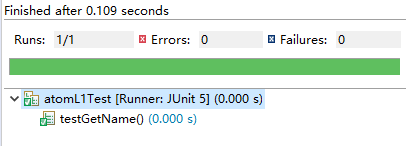
图（3）atomStructure中具体函数的测试结果

对关系中方法的测试：

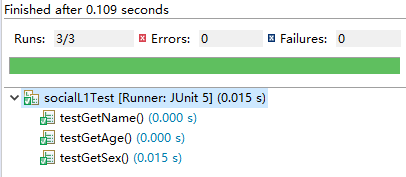


图（4）tie中具体函数的测试结果

对中心点物体中方法的测试：

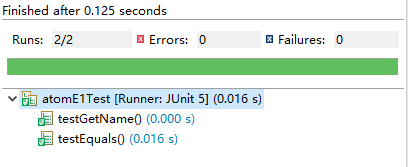


图（5）atomL1中具体函数的测试结果

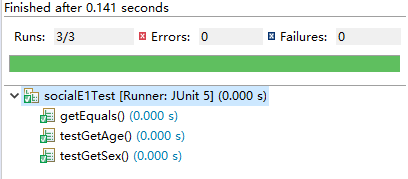


图（6）socialL1中具体函数的测试结果

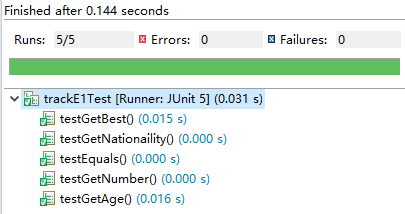
对轨道物体中方法的测试：



图（7）atomE1中具体函数的测试结果

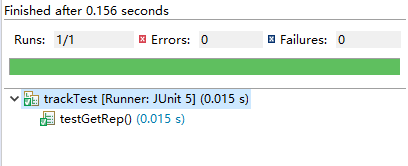


图（8）socialE1中具体函数的测试结果



图（9）trackE1中具体函数的测试结果

对轨道中方法的测试：



图（10）track中具体函数的测试结果

## 面向复用的设计：Track

track轨道类是通用的，只有半径参数，三个轨道系统中轨道都是一样的，需要注意轨道是不可变对象，定义半径时需要采用private final修饰：

**public** **class** Track

{

**private** **final** **int** rep;

// Abstraction function:

// 作为所有轨道的父类，具有共同属性：“半径”

// Representation invariant:

// 一个轨道物体的名称是String类型，不可变

// Safety from rep exposure:

// String本身是不可变的，同时中心物体的名称是final类型，指向不会改变

**public** **void** checkRep()

{

*assertTrue*(rep>=0);

}

**public** Track(**int** rep)// 构造函数

{

checkRep();

**this**.rep = rep;

}

**public** **int** getRep()

{

checkRep();

**return** rep;

}

}

## 面向复用的设计：L

L是中心点物体的父类，所有的中心点物体都有一些共同的特征，例如都有名字，而扩展版本还要求中心点物体可以由多个物体构成，同时它时不可变对象，定义物体特征时需要采用private final修饰，中心点物体的父类代码如下：

**public** **class** L1

{

**private** **final** String name;//中心物体是不可变类型

//Abstraction function:

//作为所有中心物体的父类，具有共同属性：“名称”

//Representation invariant:

//一个中心物体的名称是String类型，不可变

//Safety from rep exposure:

//String本身是不可变的，同时中心物体的名称是final类型，指向不会改变

**public** **void** checkRep()

{

*assertTrue*(name!=**null**);

}

**public** String getName()

{

checkRep();

**return** name;

}

**public** L1(String name)//构造函数

{

**this**.name = name;

checkRep();

}

}

从中心点物体派生出子类atomL1(原子系统中的原子核)和socialL1（社交网络中的中心人），田径比赛轨道系统中心点没有物体因此不需要派生，具体代码如下：

atomL1:

**public** **class** atomL1 **extends** L1

{

**public** atomL1(String name)

{

**super**(name);

}

}

socialL1:

**public** **class** socialL1 **extends** L1

{

**private** **final** **int** age;

**private** **final** **char** sex;

**public** socialL1(String name, **int** age, **char** sex)

{

**super**(name);

**this**.age = age;

**this**.sex = sex;

}

**public** **int** getAge()

{

**return** age;

}

**public** **char** getSex()

{

**return** sex;

}

}

## 面向复用的设计：PhysicalObject

所有处在轨道上的物体都有一些共同的特征，例如都有名字，需要注意轨道物体是不可变类型，定义物体特征时需要采用private final修饰，轨道物体父类E1的代码如下：

**public** **class** E1//所有的方法都已经实现，依然可以用abstract

{

**private** **final** String name;//物体名称

//Abstraction function:

//作为所有轨道物体的父类，具有共同属性：“名称”

//Representation invariant:

//一个轨道物体的名称是String类型，不可变

//Safety from rep exposure:

//String本身是不可变的，同时中心物体的名称是final类型，指向不会改变

**public** **void** checkRep()

{

*assertTrue*(name!=**null**);

}

**public** String getName()

{

checkRep();

**return** name;

}

**public** E1(String name)

{

**this**.name = name;

checkRep();

}

**public** **boolean** equals(E1 e)

{

**if**(e.getName().equals(**this**.name))

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

}

从E1可以派生出trackE1（轨道上的运动员）、atomE1（轨道上的电子）、socialE1（轨道上的人），具体代码如下：

trackE1:

**public** **class** trackE1 **extends** E1

{

**private** **final** **int** number;//号码

**private** **final** String nationaility;//国籍

**private** **final** **int** age;//年龄

**private** **final** **double** best;//本年度最好成绩

**public** trackE1(String name,**int** number,String nationaility,**int** age,**double** best)

{

**super**(name);//父类构造函数

**this**.number = number;

**this**.nationaility = nationaility;

**this**.age = age;

**this**.best = best;

}

//以下四类都是不可变的

**public** **int** getNumber()

{

**return** number;

}

**public** String getNationaility()

{

**return** nationaility;

}

**public** **int** getAge()//返回的只是一个常量

{

**return** age;

}

**public** **double** getBest()

{

**return** best;

}

**public** **boolean** equals(trackE1 temp)//比较两个运动员是否是同一个运动员

{

**if**(temp.getName().equals(**this**.getName())&&temp.getAge() == **this**.age&&temp.getNumber() == **this**.number&&temp.getNationaility().equals(**this**.nationaility))

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

}

atomE1:

**public** **class** atomE1 **extends** E1

{

**public** atomE1(String name)

{

**super**(name);

}

}

socialE1:

**public** **class** socialE1 **extends** E1

{

**private** **final** **int** age;

**private** **final** **char** sex;

**public** socialE1(String name, **int** realAge, **char** string)

{

**super**(name);

**this**.age = realAge;

**this**.sex = string;

}

**public** **int** getAge()

{

**return** age;

}

**public** **char** getSex()

{

**return** sex;

}

**public** **boolean** equals(socialE1 e)

{

**if** (e.getName().equals(**this**.getName()) && e.getAge() == **this**.age && e.getSex() == **this**.sex)

{

**return** **true**;

}

**else**

{

**return** **false**;

}

}

}

## 可复用API设计

可复用API设计是针对ConcreteCircularObject设计的，具有普遍性，首先是计算熵值，通过轨道信息和熵值公式不难计算：

**public** **double** getObjectDistributionEntropy(circularOrbit c)

{

**double** entropy = 0.00;

**int** num = 0;

Map<Integer, Set<E>> temp = c.getTrackObject();

**int** trackNumber = temp.size();

List<Double> probability = **new** ArrayList<Double>();

Iterator<Integer> iterator = temp.keySet().iterator();

**while** (iterator.hasNext())

{

Set<E> etemp = temp.get(iterator.next());

num = num + temp.size();

}

Iterator<Integer> iterator2 = temp.keySet().iterator();

**while** (iterator2.hasNext())

{

Set<E> etemp2 = temp.get(iterator2.next());

**double** e = (**double**) etemp2.size() / (**double**) num;

probability.add(e);

}

**for** (**int** i = 0; i < probability.size(); i++)

{

**double** e1 = probability.get(i);

entropy = entropy + e1 \* Math.*log*(e1);

}

**return** (-1) \* entropy;

}

接着是计算两个物体间最短逻辑距离，用到层序遍历算法，通过物体间的关系确定某个物体相对于另一个物体的逻辑层数：

**public** **int** getLogicalDistance(circularOrbit c, E e1, E e2)

{

// 层序遍历

**if** (e1.equals(e2))

{

**return** 0;

}

**int** length = 1;

// Map<E, Integer> grade = new HashMap<E, Integer>();

Map<E, Set<E>> real = c.getRelationship();

List<E> queue = **new** ArrayList<E>();

Set<E> flag = **new** HashSet<E>();// 集合内的人都已经被访问过了

queue.add(e1);

**while** (queue.size() != 0)

{

**int** size = queue.size();

List<E> cQueue = **new** ArrayList<E>();

**for** (**int** i = 0; i < size; i++)

{

E e = queue.get(0);// 队首元素出队

queue.remove(0);

**if** (real.containsKey(e))

{

Set<E> tempReal = real.get(e);// 获取当前点的所有关联点的集合

Iterator<E> iterator = tempReal.iterator();// 对该集合进行遍历

**while** (iterator.hasNext())

{

E te = iterator.next();

// 这一步意味着必须重写三个应用类中的equals方法

**if** (te.equals(e2))

{

**return** length;

}

**else**

{

**if** (!flag.contains(te))

{

flag.add(te);

cQueue.add(te);

}

}

}

}

**else**

{

System.***out***.println("the person shouldn't appear in the orbit system!");

**return** -8;

}

}

queue = cQueue;

length++;

}

System.***out***.println("There is no path between the two person!");

**return** -2;// 全部遍历之后还没有return,说明两点之间没有路径

}

接着是计算最短物理距离，通过存储物体角度的数据结构获取物体的极坐标，进而获取物体的直角坐标，通过两点间距离的直角坐标公式，得出两物体间的最短物理距离：

**public** **double** getPhysicalDistance(circularOrbit c, E e1, E e2)

{

**double** distance;

Map<E, Double> temp = c.getAngle();

Map<E, Integer> repTemp = c.getObjectTrack();

**double** angle1 = temp.get(e2);

**double** angle2 = temp.get(e2);

**int** rep1 = repTemp.get(e1) + 1;

**int** rep2 = repTemp.get(e2) + 1;

**double** x1 = rep1 \* Math.*cos*(angle1);

**double** x2 = rep2 \* Math.*cos*(angle2);

**double** y1 = rep1 \* Math.*sin*(angle1);

**double** y2 = rep2 \* Math.*sin*(angle2);

distance = Math.*sqrt*((x1 - x2) \* (x1 - x2) + (y1 - y2) \* (y1 - y2));

**return** distance;

}

接下来是计算两个轨道系统的Difference，这里需要实现Difference这个ADT，通过两个映射（Map）来存储两个轨道系统中每条对应轨道的物体差别和物体数量之差：

**public** **class** Difference<L,E>

{

**int** differenceNumber;//轨道的数量差

Map<Integer,Integer> numberTrack = **new** HashMap<Integer, Integer>();//第i条轨道上的数量差

Map<Integer, String> stringTrack = **new** HashMap<Integer, String>();//第i条轨道上的物体差

}

定义好了Difference类之后，就可以实现API中的getDifference函数，该函数首先需要判断轨道系统属于哪个类型，然后需要处理两个轨道系统中每条对应轨道的物体差异、物体数量差异，同时需要处理相应的异常情况，具体代码如下：

**public** Difference getDifference(circularOrbit c1, circularOrbit c2)// c1和c2必须是同类型的才能比较，怎么判定？

{

Difference d = **new** Difference<L, E>();

**int** basicSize;

**int** numberDifference;

**int** trackNumber1 = c1.getTrackObject().size();

**int** trackNumber2 = c2.getTrackObject().size();

numberDifference = trackNumber1 - trackNumber2;

d.differenceNumber = numberDifference;

**boolean** trackType1 = c1 **instanceof** trackGame;

**boolean** trackType2 = c1 **instanceof** trackGame;

**boolean** atomType1 = c1 **instanceof** atomStructure;

**boolean** atomType2 = c2 **instanceof** atomStructure;

**boolean** socialType1 = c1 **instanceof** socialNetworkCircle;

**boolean** socialType2 = c2 **instanceof** socialNetworkCircle;

**if** (trackType1 == **true** && trackType2 == **true**)

// 分组问题的处理:

// 先把组找好，再按照组的顺序比较，需要改动trackGame中分组的过程

{

Map<Integer, Set<socialE1>> temp1 = c1.getTrackObject();// 为什么需要改成socialE1，因为E是object对象，没有name

Map<Integer, Set<socialE1>> temp2 = c2.getTrackObject();

**if** (trackNumber1 == trackNumber2)

{

**for** (**int** i = 0; i < trackNumber1; i++)

{

String first = **new** String();

String second = **new** String();

Set<socialE1> stemp1 = temp1.get(i);

Set<socialE1> stemp2 = temp2.get(i);

Iterator<socialE1> iterator = stemp1.iterator();

**while** (iterator.hasNext())

{

first = first + iterator.next().getName();

}

Iterator<socialE1> iterator2 = stemp2.iterator();

**while** (iterator2.hasNext())

{

second = second + iterator2.next().getName();

}

d.numberTrack.put(i, temp1.get(i).size() - temp2.get(i).size());

d.stringTrack.put(i, first + "-" + second);

}

}

**else** **if** (trackNumber1 > trackNumber2)// 1的轨道数更多

{

**for** (**int** i = 0; i < trackNumber2; i++)

{

String first = **new** String();

String second = **new** String();

Set<socialE1> stemp1 = temp1.get(i);

Set<socialE1> stemp2 = temp2.get(i);

Iterator<socialE1> iterator = stemp1.iterator();

**while** (iterator.hasNext())

{

first = first + iterator.next().getName();

}

Iterator<socialE1> iterator2 = stemp2.iterator();

**while** (iterator2.hasNext())

{

second = second + iterator2.next().getName();

}

d.numberTrack.put(i, temp1.get(i).size() - temp2.get(i).size());

d.stringTrack.put(i, first + "-" + second);

}

**for** (**int** j = trackNumber2; j < trackNumber1; j++)

{

String first = **new** String();

Set<socialE1> stemp1 = temp1.get(j);

Iterator<socialE1> iterator = stemp1.iterator();

**while** (iterator.hasNext())

{

first = first + iterator.next().getName();

}

d.numberTrack.put(j, temp1.get(j).size());

d.stringTrack.put(j, first + "-");

}

}

**else**// 2的轨道数更多

{

**for** (**int** i = 0; i < trackNumber1; i++)

{

String first = **new** String();

String second = **new** String();

Set<socialE1> stemp1 = temp1.get(i);

Set<socialE1> stemp2 = temp2.get(i);

Iterator<socialE1> iterator = stemp1.iterator();

**while** (iterator.hasNext())

{

first = first + iterator.next().getName();

}

Iterator<socialE1> iterator2 = stemp2.iterator();

**while** (iterator2.hasNext())

{

second = second + iterator2.next().getName();

}

d.numberTrack.put(i, temp1.get(i).size() - temp2.get(i).size());

d.stringTrack.put(i, first + "-" + second);

}

**for** (**int** j = trackNumber1; j < trackNumber2; j++)

{

String second = **new** String();

Set<socialE1> stemp2 = temp2.get(j);

Iterator<socialE1> iterator = stemp2.iterator();

**while** (iterator.hasNext())

{

second = second + iterator.next().getName();

}

d.numberTrack.put(j, -temp1.get(j).size());

d.stringTrack.put(j, "-" + second);

}

}

}

**else** **if** (atomType1 == **true** && atomType2 == **true**)

{

**if** (trackNumber1 == trackNumber2)

{

Map<Integer, Set<atomE1>> atemp = c1.getTrackObject();

Map<Integer, Set<atomE1>> atemp2 = c2.getTrackObject();

**for** (**int** i = 0; i < trackNumber1; i++)

{

// 归根到底还是要往电子轨道上加电子

Set<atomE1> stemp = atemp.get(i);

Set<atomE1> stemp2 = atemp2.get(i);

**int** dnumber = stemp.size() - stemp2.size();

d.numberTrack.put(i, dnumber);

}

}

**else** **if** (trackNumber1 > trackNumber2)

{

Map<Integer, Set<atomE1>> atemp = c1.getTrackObject();

Map<Integer, Set<atomE1>> atemp2 = c2.getTrackObject();

**for** (**int** i = 0; i < trackNumber2; i++)

{

Set<atomE1> stemp = atemp.get(i);

Set<atomE1> stemp2 = atemp2.get(i);

**int** dnumber = stemp.size() - stemp2.size();

d.numberTrack.put(i, dnumber);

}

**for** (**int** j = trackNumber2; j < trackNumber1; j++)

{

Set<atomE1> stemp = atemp.get(j);

d.numberTrack.put(j, stemp.size());

}

}

**else**

{

Map<Integer, Set<atomE1>> atemp = c1.getTrackObject();

Map<Integer, Set<atomE1>> atemp2 = c2.getTrackObject();

**for** (**int** i = 0; i < trackNumber1; i++)

{

Set<atomE1> stemp = atemp.get(i);

Set<atomE1> stemp2 = atemp2.get(i);

**int** dnumber = stemp.size() - stemp2.size();

d.numberTrack.put(i, dnumber);

}

**for** (**int** j = trackNumber1; j < trackNumber2; j++)

{

Set<atomE1> stemp = atemp2.get(j);

d.numberTrack.put(j, -stemp.size());

}

}

}

**else** **if** (socialType1 == **true** && socialType2 == **true**)

{

**if** (trackNumber1 == trackNumber2)

{

Map<Integer, Set<socialE1>> temp1 = c1.getTrackObject();

Map<Integer, Set<socialE1>> temp2 = c2.getTrackObject();

**for** (**int** i = 0; i < trackNumber1; i++)

{

Set<socialE1> stemp1 = temp1.get(i);

Set<socialE1> stemp2 = temp2.get(i);

List<socialE1> firstHave = **new** ArrayList<socialE1>();

List<socialE1> secondHave = **new** ArrayList<socialE1>();

Iterator<socialE1> iterator = stemp1.iterator();

**while** (iterator.hasNext())

{

socialE1 person1 = iterator.next();

**if** (!stemp2.contains(person1))

{

firstHave.add(person1);

}

}

Iterator<socialE1> iterator2 = stemp2.iterator();

**while** (iterator2.hasNext())

{

socialE1 person2 = iterator.next();

**if** (!stemp1.contains(person2))

{

secondHave.add(person2);

}

}

String first = **new** String();

**for** (**int** j = 0; j < firstHave.size(); j++)

{

first = first + "," + firstHave.get(j).getName();

}

String second = **new** String();

**for** (**int** k = 0; k < secondHave.size(); k++)

{

second = second + "," + secondHave.get(k).getName();

}

d.stringTrack.put(i, "{" + first + "}" + "-" + "{" + second + "}");

// 该条轨道上物体数量的差异

**int** dnumber = stemp1.size() - stemp2.size();

d.numberTrack.put(i, dnumber);

}

}

**else** **if** (trackNumber1 > trackNumber2)// 1系统的轨道数目更多

{

Map<Integer, Set<socialE1>> temp1 = c1.getTrackObject();

Map<Integer, Set<socialE1>> temp2 = c2.getTrackObject();

**for** (**int** i = 0; i < trackNumber2; i++)

{

List<socialE1> firstHave = **new** ArrayList<socialE1>();

List<socialE1> secondHave = **new** ArrayList<socialE1>();

Set<socialE1> stemp1 = temp1.get(i);

Set<socialE1> stemp2 = temp2.get(i);

Iterator<socialE1> iterator = stemp1.iterator();

**while** (iterator.hasNext())

{

socialE1 person1 = iterator.next();

**if** (!stemp2.contains(person1))

{

firstHave.add(person1);

}

}

Iterator<socialE1> iterator2 = stemp2.iterator();

**while** (iterator2.hasNext())

{

socialE1 person2 = iterator.next();

**if** (!stemp1.contains(person2))

{

secondHave.add(person2);

}

}

String first = **new** String();

**for** (**int** j = 0; j < firstHave.size(); j++)

{

first = first + "," + firstHave.get(j).getName();

}

String second = **new** String();

**for** (**int** k = 0; k < secondHave.size(); k++)

{

second = second + "," + secondHave.get(k).getName();

}

d.stringTrack.put(i, "{" + first + "}" + "-" + "{" + second + "}");

// 该条轨道上物体数量的差异

**int** dnumber = stemp1.size() - stemp2.size();

d.numberTrack.put(i, dnumber);

}

**for** (**int** i = trackNumber2; i < trackNumber1; i++)

{

List<socialE1> firstHave = **new** ArrayList<socialE1>();

Set<socialE1> stemp1 = temp1.get(i);

Iterator<socialE1> iterator = stemp1.iterator();

**while** (iterator.hasNext())

{

socialE1 person1 = iterator.next();

firstHave.add(person1);

}

String first = **new** String();

**for** (**int** j = 0; j < firstHave.size(); j++)

{

first = first + "," + firstHave.get(j).getName();

}

d.stringTrack.put(i, "{" + first + "}" + "-" + "0");

}

}

**else**// 2系统的轨道数目更多

{

Map<Integer, Set<socialE1>> temp1 = c1.getTrackObject();

Map<Integer, Set<socialE1>> temp2 = c2.getTrackObject();

**for** (**int** i = 0; i < trackNumber1; i++)

{

Set<socialE1> stemp1 = temp1.get(i);

Set<socialE1> stemp2 = temp2.get(i);

List<socialE1> firstHave = **new** ArrayList<socialE1>();

List<socialE1> secondHave = **new** ArrayList<socialE1>();

Iterator<socialE1> iterator = stemp1.iterator();

**while** (iterator.hasNext())

{

socialE1 person1 = iterator.next();

**if** (!stemp2.contains(person1))

{

firstHave.add(person1);

}

}

Iterator<socialE1> iterator2 = stemp2.iterator();

**while** (iterator2.hasNext())

{

socialE1 person2 = iterator.next();

**if** (!stemp1.contains(person2))

{

secondHave.add(person2);

}

}

String first = **new** String();

**for** (**int** j = 0; j < firstHave.size(); j++)

{

first = first + "," + firstHave.get(j).getName();

}

String second = **new** String();

**for** (**int** k = 0; k < secondHave.size(); k++)

{

second = second + "," + secondHave.get(k).getName();

}

d.stringTrack.put(i, "{" + first + "}" + "-" + "{" + second + "}");

// 该条轨道上物体数量的差异

**int** dnumber = stemp1.size() - stemp2.size();

d.numberTrack.put(i, dnumber);

}

**for** (**int** i = trackNumber1; i < trackNumber2; i++)

{

Set<socialE1> stemp2 = temp2.get(i);

Iterator<socialE1> iterator = stemp2.iterator();

List<socialE1> secondHave = **new** ArrayList<socialE1>();

**while** (iterator.hasNext())

{

socialE1 person2 = iterator.next();

secondHave.add(person2);

}

String second = **new** String();

**for** (**int** j = 0; j < secondHave.size(); j++)

{

second = second + "," + secondHave.get(j).getName();

}

d.stringTrack.put(i, "0" + "-" + "{" + second + "}");

}

}

}

**else**

{

System.***out***.println("The two orbit systems are not the same type!");

}

**return** d;

}

## 图的可视化：第三方API的复用

在图的可视化中，采用Swing绘图，面向CircularOrbit，利用不同轨道系统的共性，绘制出轨道、物体与关系的示意图，首先在circularOrbitHelper中定义若干对象，绘图时需要使用：

**int** trackNumber;

Map<Integer, Set<E>> temptrackObject;

Map<Integer, Integer> thingsNumberPerTrack = **new** HashMap<Integer, Integer>();// 必须要开辟一段空间

Map<E, position> thingsPosition = **new** HashMap<E, position>();

Map<E, Set<E>> EErelationship;

Set<E> LErelationship;

List<tie> tempsocialTie;

L cretral;

然后写出JFrame和JPanel的子类，在这些子类中定义画图的具体方法，注意还需要实现一个position类，用来标识物体在轨道上的位置：

**class** position

{

**private** **final** **int** x;

**private** **final** **int** y;

**public** position(**int** x, **int** y)

{

**this**.x = x;

**this**.y = y;

}

**public** **int** getX()

{

**return** **this**.x;

}

**public** **int** getY()

{

**return** **this**.y;

}

}

两个子类的代码如下：

JFrame的子类MyFrame：

**class** myFrame **extends** JFrame

{

**public** **static** **final** String ***TITLE*** = "Java图形绘制";

**public** **static** **final** **int** ***WIDTH*** = 1000;

**public** **static** **final** **int** ***HEIGHT*** = 618;

**public** myFrame()

{

**super**();

initFrame();

}

**private** **void** initFrame()

{

// 设置 窗口标题 和 窗口大小

setTitle(***TITLE***);

setSize(***WIDTH***, ***HEIGHT***);

// 设置窗口关闭按钮的默认操作(点击关闭时退出进程)

setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

// 把窗口位置设置到屏幕的中心

setLocationRelativeTo(**null**);

// 设置窗口的内容面板

myPanel panel = **new** myPanel(**this**);

setContentPane(panel);

}

}

JPanel的子类MyPanel：

**public** **void** paintComponent(Graphics g)

{

// g.drawRoundRect(10, 10, 100, 100, 100, 100);

**int** length = 50;

**int** doubleLength = 2 \* length;

**super**.paintComponent(g);

**int** width = getWidth();

**int** height = getHeight();

g.fillOval(width / 2 - 16, height / 2 - 16, 32, 32);// 画中心点物体

Iterator<Integer> iterator = temptrackObject.keySet().iterator();

**int** m = 0;

**while** (iterator.hasNext())

{

// 画轨道部分

**int** realNumber = iterator.next();

System.***out***.println("realNumber" + realNumber);

g.drawOval(width / 2 - (m + 1) \* length, height / 2 - (m + 1) \* length, doubleLength + doubleLength \* m,

doubleLength + doubleLength \* m);

**int** thingsNumber = thingsNumberPerTrack.get(realNumber);

System.***out***.println("thingsNumber" + thingsNumber);

**double** partition;

**if** (thingsNumber != 0)// 加了一条轨道之后，新轨道上是没有物体的

{

partition = (Math.***PI***) / (**double**)(thingsNumber);

// 画点部分

**double** a = 0.00;

// 获得该轨道上的所有物体

List<E> things = **new** ArrayList<E>(temptrackObject.get(realNumber));

**for** (**int** j = 0; j < thingsNumber; j++)

{

a = a + partition;

**int** x = width / 2 - (m + 1) \* length + (length \* (m + 1))

+ (**int**) (((m + 1) \* length) \* Math.*cos*(a + j \* partition));

**int** y = height / 2 - (m + 1) \* length + (length \* (m + 1))

+ (**int**) (((m + 1) \* length) \* Math.*sin*(a + j \* partition));

// thingsPosition.put(temptrackObject.get(i), new position(x, y));

thingsPosition.put(things.get(j), **new** position(x, y));// 轨道上的每个物体对应于一个坐标

g.setFont(**new** Font("宋体", Font.***PLAIN***, 12));

g.fillOval(x - 8, y - 8, 16, 16);

g.setColor(Color.***GREEN***);

g.drawString(things.get(j).getName(), x - 12, y - 12);

g.setColor(Color.***BLACK***);

}

}

m++;

}

g.setColor(Color.***RED***);

System.***out***.println("i is " + m);

System.***out***.println(temptrackObject.size());

// 画轨道物体间的连线

//System.out.println("准备画线");// 以上代码没有问题

Iterator<E> iterator1 = EErelationship.keySet().iterator();

g.setColor(Color.***RED***);

**while** (iterator1.hasNext())

{

//System.out.println("开始画线啦！");

E r1 = iterator1.next();

Set<E> rr1 = EErelationship.get(r1);

Iterator<E> iterator2 = rr1.iterator();

//System.out.println("画线");

**while** (iterator2.hasNext())// 删除第一层轨道之后，这个循环从来没进过

{

g.setColor(Color.***RED***);

//System.out.println("xxxxxxxxxx");

E m1 = iterator2.next();

g.drawLine(thingsPosition.get(r1).getX(), thingsPosition.get(r1).getY(),

thingsPosition.get(m1).getX(), thingsPosition.get(m1).getY());

// 在连线上注明亲密度,亲密度的颜色是蓝色

g.setColor(Color.***blue***);

**for**(**int** mn=0; mn<tempsocialTie.size(); mn++)

{

**if**((r1.getName().equals(tempsocialTie.get(mn).getName1())&&m1.getName().equals(tempsocialTie.get(mn).getName2()))||(r1.getName().equals(tempsocialTie.get(mn).getName2())&&m1.getName().equals(tempsocialTie.get(mn).getName1())))

{

**int** numberx = (thingsPosition.get(r1).getX()+thingsPosition.get(m1).getX())/2;

**int** numbery = (thingsPosition.get(r1).getY()+thingsPosition.get(m1).getY())/2;

g.drawString(Float.*toString*(tempsocialTie.get(mn).getIni()), numberx, numbery);

}

}

}

}

// 画中心点与轨道物体的连线

Iterator<E> iterator2 = LErelationship.iterator();

//System.out.println("eee" + LErelationship.size());

**while** (iterator2.hasNext())

{

g.setColor(Color.***RED***);

E r2 = iterator2.next();

g.drawLine(width / 2, height / 2, thingsPosition.get(r2).getX(), thingsPosition.get(r2).getY());

g.setColor(Color.***blue***);

**for**(**int** v=0; v<tempsocialTie.size(); v++)

{

//这个判断实际上比较冗余，如果把friend加到concretecircularOrbit里面就会好很多

**if**((r2.getName().equals(tempsocialTie.get(v).getName1())&&cretral.getName().equals(tempsocialTie.get(v).getName2()))||(r2.getName().equals(tempsocialTie.get(v).getName2())&&cretral.getName().equals(tempsocialTie.get(v).getName1())))

{

**int** numberx = (thingsPosition.get(r2).getX()+width/2)/2;

**int** numbery = (thingsPosition.get(r2).getY()+height/2)/2;

g.drawString(Float.*toString*(tempsocialTie.get(v).getIni()), numberx, numbery);

}

}

}

}

}

最后用通过这两个子类构造出visualize()函数，实现可视化：

**public** **void** visualize()// 最终要调用的函数

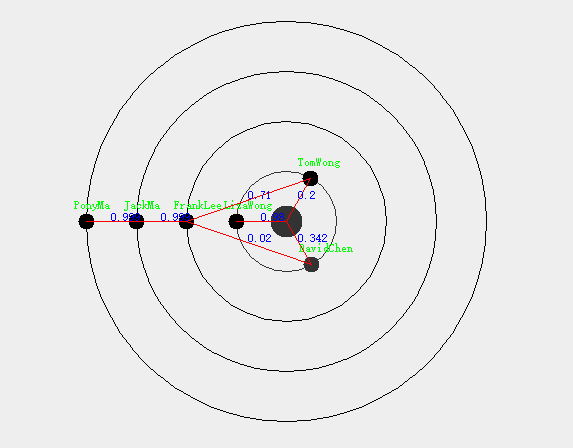
{

myFrame frame = **new** myFrame();

frame.setVisible(**true**);

}

调用该函数画图的一个例子如下：



图（11）visualize()函数画出的一个轨道系统

## 设计模式应用

请分小节介绍每种设计模式在你的ADT和应用设计中的具体应用。

(1) 构造 Track、PhysicalObject 等对象时，使用 factory method 设计模式，设计其工厂类并实现之。

为Trak类和三个具体轨道系统中的三种轨道物体和中心物体分别建立工厂，通过工厂获得新的实例，其中为轨道系统建立的工厂，每个工厂既能生产中心物体类，又能生产轨道物体类：

原子轨道的工厂类：

**public** **class** atomFactory

{

**public** atomL1 manufactureL(String name)

{

**return** **new** atomL1(name);

}

**public** atomE1 manufactureE(String name)

{

**return** **new** atomE1(name);

}

}

社交网络的工厂类：

**public** **class** socialFactory

{

**public** socialL1 manufactureL(String name, **int** age, **char** sex)

{

**return** **new** socialL1(name,age,sex);

}

**public** socialE1 manufactureE(String name, **int** realAge, **char** realSex)

{

**return** **new** socialE1(name,realAge,realSex);

}

}

田径比赛的工厂类：

**public** **class** trackFactory

{

**public** L1 manufactureL(String name)

{

**return** **new** L1(name);

}

**public** trackE1 manufactureE(String name,**int** number,String nationaility,**int** age,**double** best)

{

**return** **new** trackE1(name, number, nationaility, age, best);

}

}

轨道的工厂类：

**public** **class** trackFactory

{

**public** Track manufacture(**int** rep)

{

**return** **new** Track(rep);

}

}

(2) 构造 ConcreteCircularOrbit 对象时，针对不同应用中所需的不同类型的 L 和 E，使用 abstract factory 或 builder 设计模式。

(2)中的要求已经在(1)中完成。

(3) 请使用 Iterator 设计模式，设计迭代器，客户端可在遍历 CircularOrbit 对象中的各 PhysicalObject 对象时使用，遍历次序为：从内部轨道逐步向外、

同一轨道上的物体按照其角度从小到大的次序（若不考虑绝对位置，则随机

遍历）。

首先定义容器接口aggregate，在其中定义iterator方法；然后定义迭代器接口physicalIterator，在其中定义next()和hasNext()方法；接着实现容器接口physicalShelf，最后实现迭代器接口physicalShelfIterator：

**public** **interface** aggregate<E>

{

**public** **abstract** physicalIterator iterator();

}

**public** **interface** physicalIterator//属于physical的迭代器

{

**public** **abstract** **boolean** hasNext();

**public** **abstract** E1 next();

}

**public** **class** physicalShelf **implements** aggregate

{

**private** E1[] physical;

**private** **int** last = 0;

// 构造函数

// 查找物体

**public** E1 getphysicalAt(**int** index)

{

**return** physical[index];

}

// 添加物体

**public** **void** appendphysical(E1 book)

{

**this**.physical[last] = book;

last++;

}

// 获得容器中物体的数量

**public** **int** getLength()

{

**return** physical.length;

}

// 获得物体迭代器对象

@Override

**public** physicalIterator iterator()

{

**return** **new** physicalShelfIterator(**this**);

}

}

**public** **class** physicalShelfIterator **implements** physicalIterator

{

**private** physicalShelf realPhysicalShelf;

**private** **int** index;

**public** physicalShelfIterator(physicalShelf physicalShelf) {

**this**.realPhysicalShelf = physicalShelf;

**this**.index = 0;

}

//检查是否还有下一个物体

**public** **boolean** hasNext() {

**if**(index < realPhysicalShelf.getLength()) {

**return** **true**;

}

**else** {

**return** **false**;

}

}

//返回指定位置的书籍

@Override

**public** E1 next() {

E1 physical = realPhysicalShelf.getphysicalAt(index);

index ++;

**return** physical;

}

}

(4) 在设计可复用 API 时，请遵循 façade 设计模式，将所有 API 放置在 helper类CircularOrbitAPIs 当中。

实际上CircularOrbitAPIs就起到一个暴露函数的作用，将所有的API方法放置在CircularOrbitAPIs当中就已经实现了façade设计模式，其中每一个API方法可以视为一个外观类。

(5) 在 TrackGame 应用中，请使用 strategy 设计模式实现多种不同的比赛方案编排策略，并可容易的切换不同的算法。

strategy设计模式向客户端隐藏了实现不同编排方案的细节，从而降低了两者的依赖度，实现了解耦。strategy设计模式需要一个strategy抽象类，然后不同的子类分别实现它，在子类中调用不同的比赛方案编排函数，最后需要实现一个管理类strategyOrganizer，它根据用户传入的字符串决定新建哪一个strategy子类并调用相应的比赛方案编排函数。而在客户端代码看来，无论客户输入什么样的字符，调用的代码都是一模一样的，但实际上却调用了不同的比赛方案编排函数：

strategy抽象类：

**public** **abstract** **class** trackStrategy

{

**public** **abstract** **void** strategy(functionTrackGame game);

}

strategy抽象类的子类：

**public** **class** trackStrategyG **extends** trackStrategy

{

@Override

**public** **void** strategy(functionTrackGame game)

{

game.autoCompetitionB();

}

}

strategy抽象类的子类：

**public** **class** trackStrategyR **extends** trackStrategy

{

@Override

**public** **void** strategy(functionTrackGame game)

{

game.autoCompetitionA();

}

}

管理类strategyOrganizer：

**public** **class** trackOrganizer

{

**private** trackStrategy yes;

**public** trackOrganizer(String choice)

{

**switch** (choice)

{

**case** "r":

yes = **new** trackStrategyR();

**break**;

**case** "g":

yes = **new** trackStrategyG();

**break**;

**default**:

**break**;

}

}

**public** **void** arrange(functionTrackGame game)

{

yes.strategy(game);

}

}

(6) 在 AtomStructure 应用中，请使用 state 和 memento 设计模式管理电子跃迁的状态，并可进行状态的恢复。意即：可保存电子每次跃迁前后的轨道信息。

使用备忘录模式的好处，体现在电子跃迁中，就是可以随时恢复到电子跃迁之前任何一步的状态。备忘录模式首先需要定义状态state这个ADT：

**public** **class** atomState

{

**private** Map<Integer, Integer> trackObjectNumber;

// static atomState instance = new atomState();

**public** atomState(Map<Integer,Integer> trackNumber)

{

**this**.trackObjectNumber = trackNumber;

}

**public** Map<Integer, Integer> getTrackObjectNumber()

{

Map<Integer, Integer> tempTrackObjectNumber = **new** HashMap<Integer, Integer>();

Iterator<Integer> iterator = trackObjectNumber.keySet().iterator();

**while** (iterator.hasNext())

{

**int** temp = iterator.next();

tempTrackObjectNumber.put(temp, trackObjectNumber.get(temp));

}

**return** tempTrackObjectNumber;

}

}

然后需要定义：

Originator（原发器）：它是一个普通类，可以创建一个备忘录，并存储它的当前内部状态，也可以使用备忘录来恢复其内部状态，一般将需要保存内部状态的类设计为原发器。

Memento（备忘录)：存储原发器的内部状态，根据原发器来决定保存哪些内部状态。备忘录的设计一般可以参考原发器的设计，根据实际需要确定备忘录类中的属性。需要注意的是，除了原发器本身与负责人类之外，备忘录对象不能直接供其他类使用，原发器的设计在不同的编程语言中实现机制会有所不同。

Caretaker（负责人）：负责人又称为管理者，它负责保存备忘录，但是不能对备忘录的内容进行操作或检查。在负责人类中可以存储一个或多个备忘录对象，它只负责存储对象，而不能修改对象，也无须知道对象的实现细节。

备忘录模式的核心是备忘录类以及用于管理备忘录的负责人类的设计。

具体实现代码如下：

备忘录：

**public** **class** atomMemento

{

**private** atomState state;

**public** atomMemento(atomState state)

{

**this**.state = state;

}

**public** atomState getState()

{

**return** **this**.state;

}

}

原发器：

**public** **class** atomOriginator

{

**private** atomState state;

**public** **void** setState(atomState state)

{

**this**.state = state;

}

**public** atomMemento addMemento()

{

**return** **new** atomMemento(**this**.state);

}

**public** **void** restore(atomMemento m)

{

**this**.state = m.getState();

}

}

负责人：

**public** **class** atomCareTaker

{

**private** ArrayList<atomMemento> mementos = **new** ArrayList<atomMemento>();

**public** **void** addMemento(atomMemento m)

{

mementos.add(m);

}

**public** atomMemento getMemento(**int** i)

{

**if**(mementos.size()-i<0)

{

**throw** **new** RuntimeException("Cannot rollback so many backs!");

}

**return** mementos.get(mementos.size()-i);//获得倒数某一步的状态

}

}

## 应用设计与开发

利用上述设计和实现的ADT，实现手册里要求的各项功能。

以下各小节，只需保留和完成你所选定的应用即可。

最终面向客户的应用集中在一个类application中，这个类的main函数会首先提示用户选择哪种轨道模型，然后提示用户输入轨道模型的大小以及其他参数，同时对用户的异常输入进行处理，然后新建一个functionTrackGame或atomStructure或socialNetWork的实例，接下来用户通过输入不同的数字来对轨道系统进行不同的操作，由于每一个操作都在实验课上与助教老师进行了验收，并确认了功能的正确性，所以具体操作的实现在此不再赘述，对于每个方法，只给出对应的命令列表以及部分可视化截图

### TrackGame

**case** 0:

//结束操作

again = **false**;

**break**;

**case** 1:

// 调用可视化函数

*trackVisualize*(track);

**break**;

**case** 2:

// 增加轨道

*trackAddTrack*(track);

**break**;

**case** 3:

// 向特定轨道上增加物体

*trackAddObject*(track);

**break**;

**case** 4:

// 删除轨道

*trackDeleteTrack*(track);

**break**;

**case** 5:

// 在特定轨道上删除物体,没有给出正确提示

*trackDeleteObject*(track);

**break**;

**case** 6:

//求熵值

System.***out***.println("The entropy of the system is:" + *trackEntropy*(track));

**break**;

**case** 7:

//更换轨道

*trackChange*(track);

**break**;

**case** 8:

//检查轨道系统的合法性

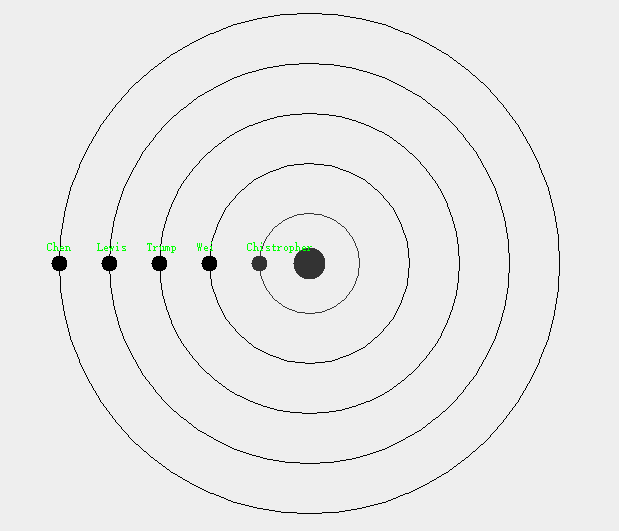
*trackCheck*(track);

**break**;

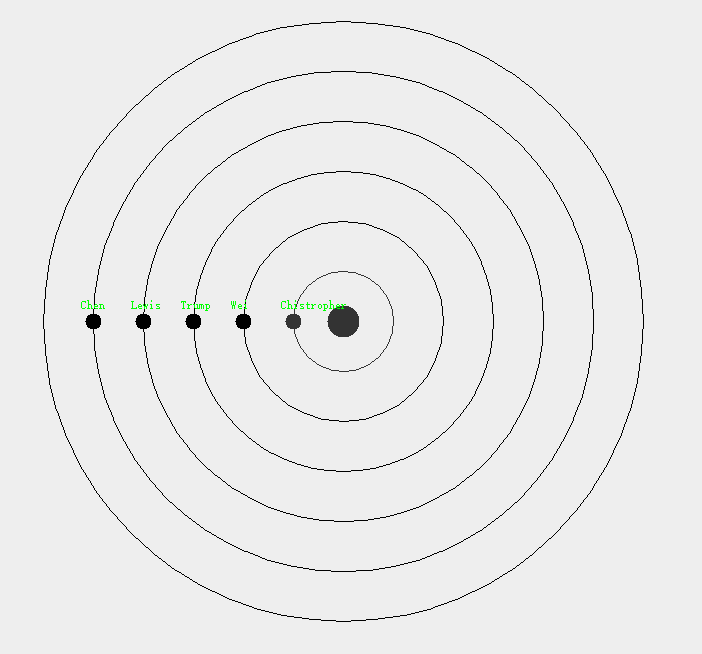
**default**:

**break**;

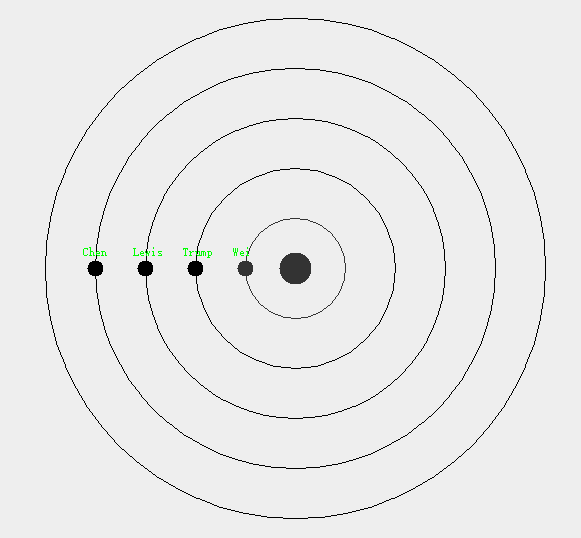
注：由于实验课上已经将所有功能进行验收并已经确保正确性，因此此处只列出部分截图，并未覆盖全部功能。



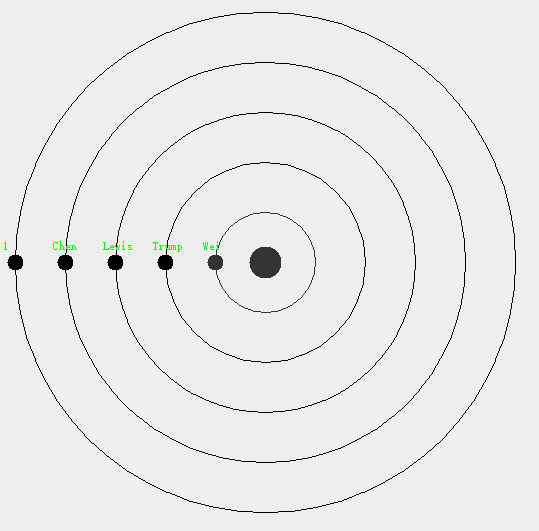
图（12）初始状态



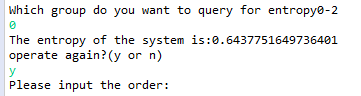
图（13）增加一条轨道



图（14）删除最内层轨道



图（15）在最外层轨道上增加物体



图（16）计算此时轨道系统的熵值

### AtomStructure

**case** 0:

//操作结束

again = **false**;

**break**;

**case** 1:

// 调用可视化函数

*atomVisualize*(atom);

**break**;

**case** 2:

// 增加轨道

*atomAddTrack*(atom);

**break**;

**case** 3:

// 向特定轨道上增加物体

*atomAddObject*(atom);

**break**;

**case** 4:

// 删除轨道

*atomDeleteTrack*(atom);

**break**;

**case** 5:

// 在特定轨道上删除物体

*atomDeleteObject*(atom);

**break**;

**case** 6:

//求熵值

System.***out***.println("The entropy of the system is:" + *atomEntropy*(atom));

**break**;

**case** 7:

//电子跃迁

*atomTransit*(atom, careTaker, originator);

**break**;

**case** 8:

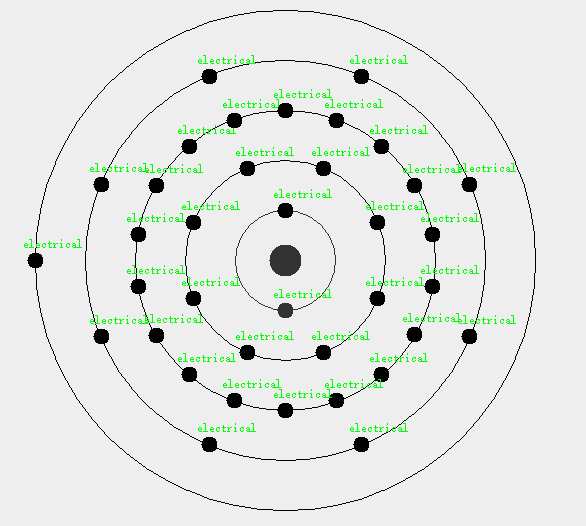
//回溯

*atomGoBack*(atom, careTaker, originator);

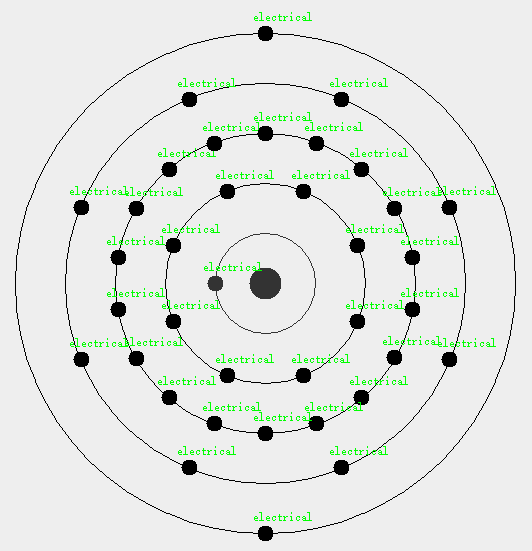
**break**;

**default**:

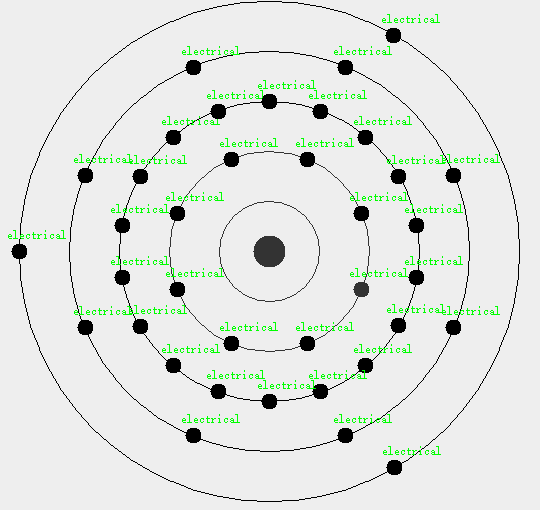
**break**;



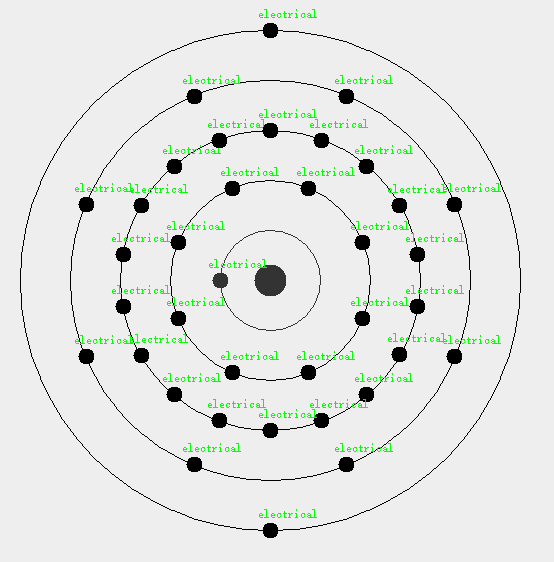
图（17）初始状态



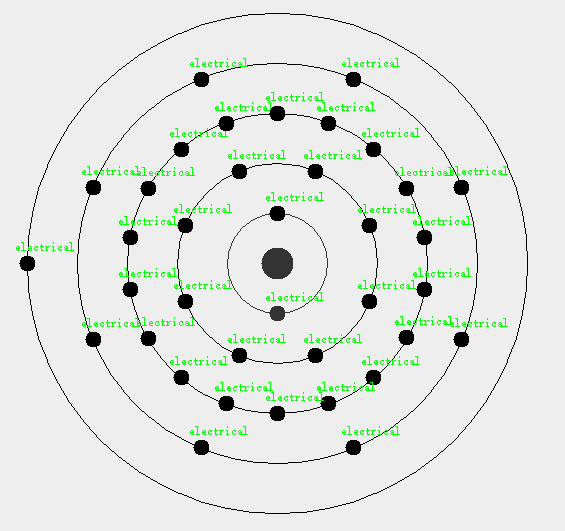
图（18）最内层轨道电子向最外层轨道跃迁



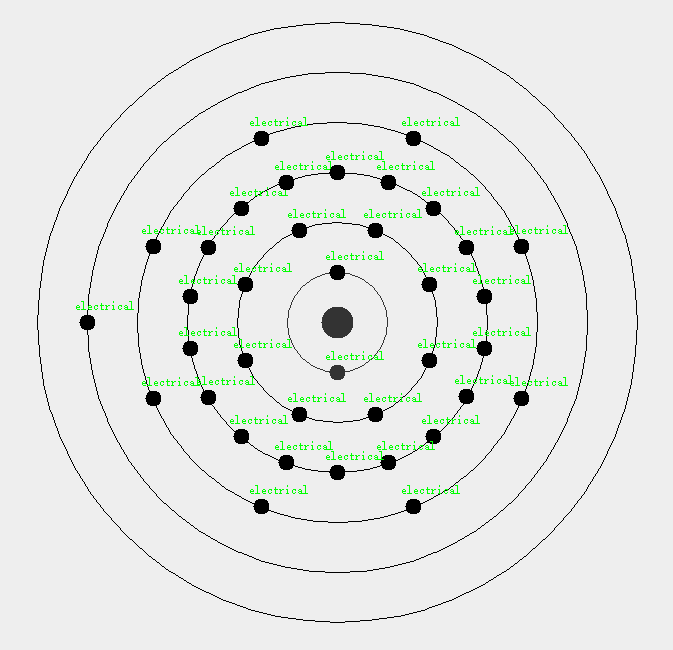
图（19）最内层轨道电子继续向最外层轨道跃迁



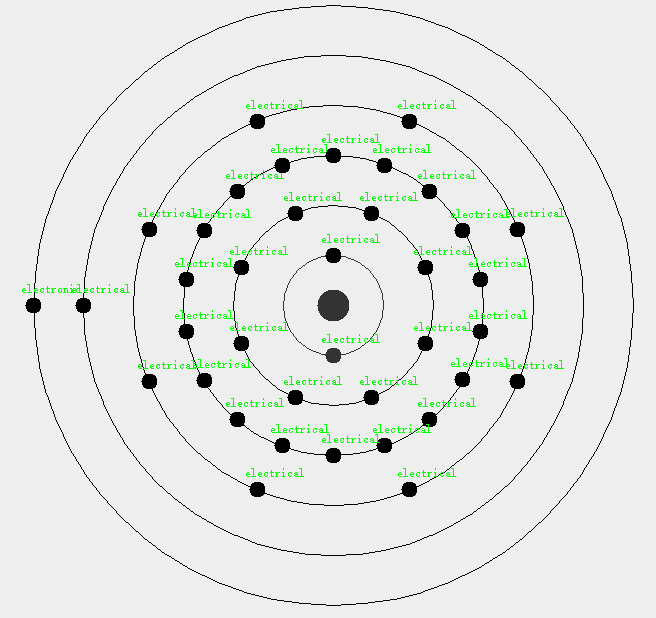
图（20）通过备忘录模式返回之前的状态

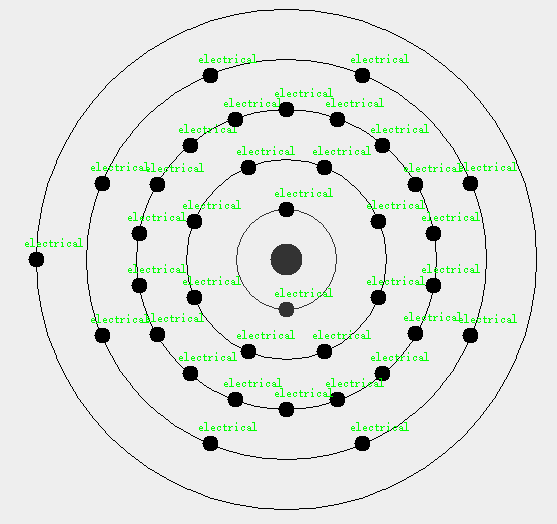


图（21）通过备忘录模式继续返回之前的状态

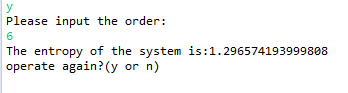


图（22）返回到初始状态后，增加新轨道

图（23）在新增的轨道上增加电子



图（24）删除最外层的轨道



图（25）计算此时轨道系统的熵值

### SocialNetworkCircle

**case** 0:

again = **false**;

**break**;

**case** 1:

// 调用可视化函数

*socialVisualize*(society);

**break**;

**case** 2:

// 增加轨道,与中心点物体没有关系的人也能出现在轨道上

*socialAddTrack*(society);

**break**;

**case** 3:

// 向特定轨道上增加物体,与中心点物体没有关系的人也能出现在轨道上

*socialAddObject*(society);

**break**;

**case** 4:

// 删除轨道,删除第一条轨道,已经重新调整图结构

*socialDeleteTrack*(society);

**break**;

**case** 5:

// 在特定轨道上删除物体,需要重新调整图结构

*socialDeleteObject*(society);

**break**;

**case** 6:

//求熵值

System.***out***.println("The entropy of the system is:" + *socialEntropy*(society));

**break**;

**case** 7:

//求通过第一层的某个好友能间接认识多少个好友

*socialIntimacy*(society);

**break**;

**case** 8:

//增加/删除关系后重新调整图结构

*socialAR*(society);

**break**;

**case** 9:

//计算两个人之间的最短逻辑距离

String name1;

String name2;

Scanner disIn = **new** Scanner(System.***in***);

System.***out***.println("Please input the first man's name:");

name1 = disIn.next();

System.***out***.println("Please input the second man's name:");

name2 = disIn.next();

System.***out***.println(

"The shortest logical distance is：" + *socialGetDistance*(society, name1, name2));

**break**;

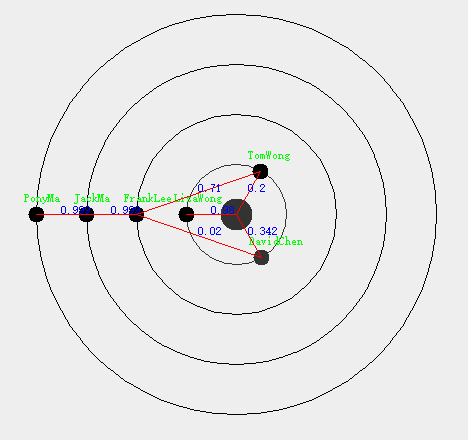
**case** 10:

//检查轨道系统的合法性

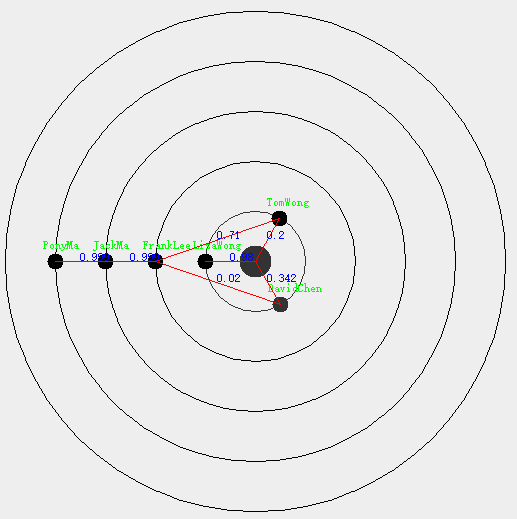
*socialCheck*(society);

**default**:

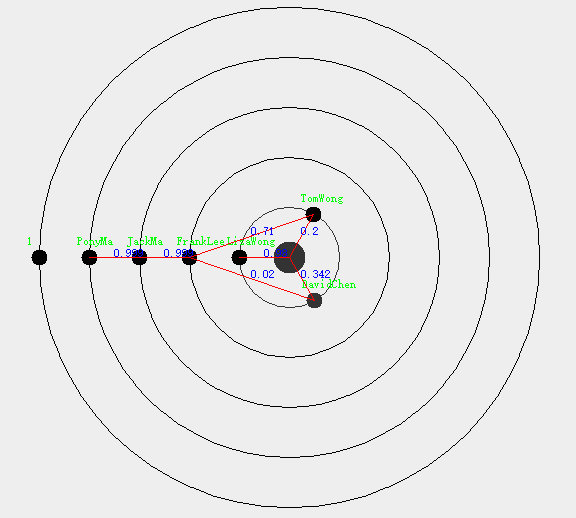
**break**;



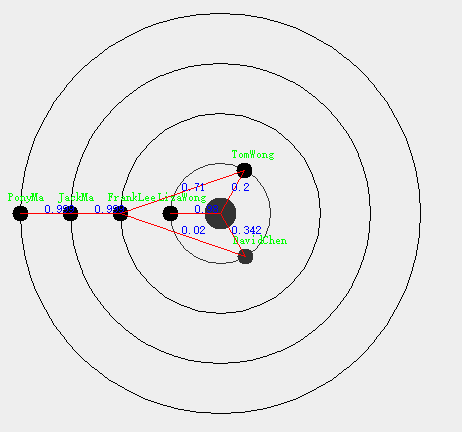
图（26）初始状态



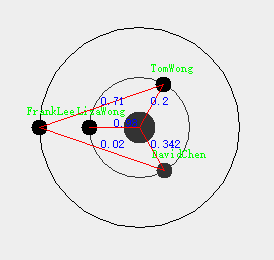
图（27）增加一条轨道



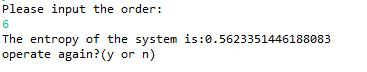
图（28）在轨道上增加物体



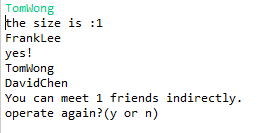
图（29）删除新增的轨道



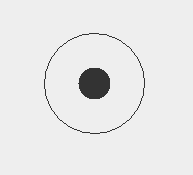
图（30）删除第3条轨道后重新调整图结构



图（31）计算此时轨道的熵值



图（32）计算中心点人通过TomWong能间接认识多少朋友



图（33）删除第一条轨道之后重新调整图结构

## 应对应用面临的新变化

以下各小节，只需保留和完成你所选定的应用即可。

由于改进后的功能已在实验课上验收并且已经确认了正确性，因此只给出命令参数和扩展功能的截图

### TrackGame

**case** 0:

//结束操作

again = **false**;

**break**;

**case** 1:

// 调用可视化函数

*trackVisualize*(track);

**break**;

**case** 2:

// 增加轨道

*trackAddTrack*(track);

**break**;

**case** 3:

// 向特定轨道上增加物体

*trackAddObject*(track);

**break**;

**case** 4:

// 删除轨道

*trackDeleteTrack*(track);

**break**;

**case** 5:

// 在特定轨道上删除物体,没有给出正确提示

*trackDeleteObject*(track);

**break**;

**case** 6:

//计算熵值

System.***out***.println("The entropy of the system is:" + *trackEntropy*(track));

**break**;

**case** 7:

//调整运动员轨道

*trackChange*(track);

**break**;

**case** 8:

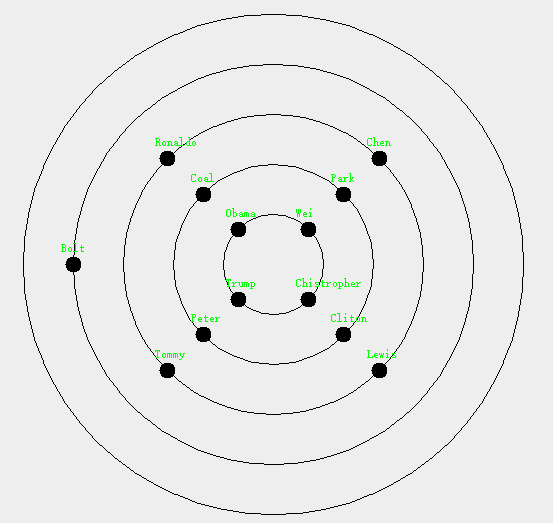
//检查轨道的合法性

*trackCheck*(track);

**break**;

**default**:

### break;



图（34）实现接力赛，并提示多出了一个运动员

### AtomStructure

**case** 0:

//结束操作

again = **false**;

**break**;

**case** 1:

// 调用可视化函数

*atomVisualize*(atom);

**break**;

**case** 2:

// 增加轨道

*atomAddTrack*(atom);

**break**;

**case** 3:

// 向特定轨道上增加物体

*atomAddObject*(atom);

**break**;

**case** 4:

// 删除轨道

*atomDeleteTrack*(atom);

**break**;

**case** 5:

// 在特定轨道上删除物体

*atomDeleteObject*(atom);

**break**;

**case** 6:

//计算熵值

System.***out***.println("The entropy of the system is:" + *atomEntropy*(atom));

**break**;

**case** 7:

//电子跃迁

*atomTransit*(atom, careTaker, originator);

**break**;

**case** 8:

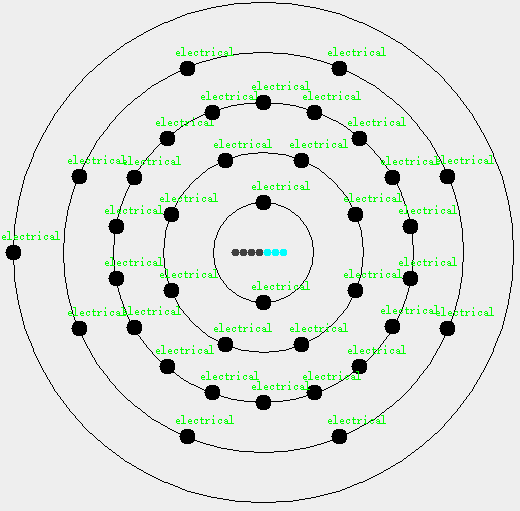
//回溯

*atomGoBack*(atom, careTaker, originator);

**break**;

**default**:

### break;



图（35）原子核由若干个质子和中子构成

### SocialNetworkCircle

**case** 0:

//结束操作

again = **false**;

**break**;

**case** 1:

//调用可视化函数

*socialVisualize*(society);

**break**;

**case** 2:

//增加轨道,与中心点物体没有关系的人也能出现在轨道上

*socialAddTrack*(society);

**break**;

**case** 3:

//向特定轨道上增加物体,与中心点物体没有关系的人也能出现在轨道上

*socialAddObject*(society);

**break**;

**case** 4:

//删除轨道,删除第一条轨道,已经重新调整图结构

*socialDeleteTrack*(society);

**break**;

**case** 5:

//在特定轨道上删除物体,需要重新调整图结构

*socialDeleteObject*(society);

**break**;

**case** 6:

//计算熵值

System.***out***.println("The entropy of the system is:" + *socialEntropy*(society));

**break**;

**case** 7:

//计算通过某个好友能间接认识多少好友

*socialIntimacy*(society);

**break**;

**case** 8:

//增加/删除关系后重新调整图结构

*socialAR*(society);

**break**;

**case** 9:

//计算最短逻辑距离

String name1;

String name2;

Scanner disIn = **new** Scanner(System.***in***);

System.***out***.println("Please input the first man's name:");

name1 = disIn.next();

System.***out***.println("Please input the second man's name:");

name2 = disIn.next();

System.***out***.println(

"The shortest logical distance is：" + *socialGetDistance*(society, name1, name2));

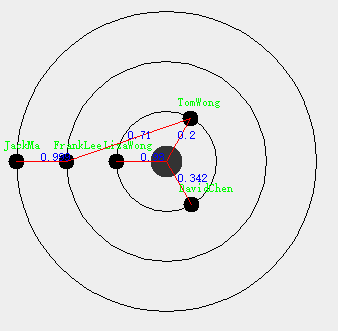
**break**;

**case** 10:

*socialCheck*(society);

**default**:

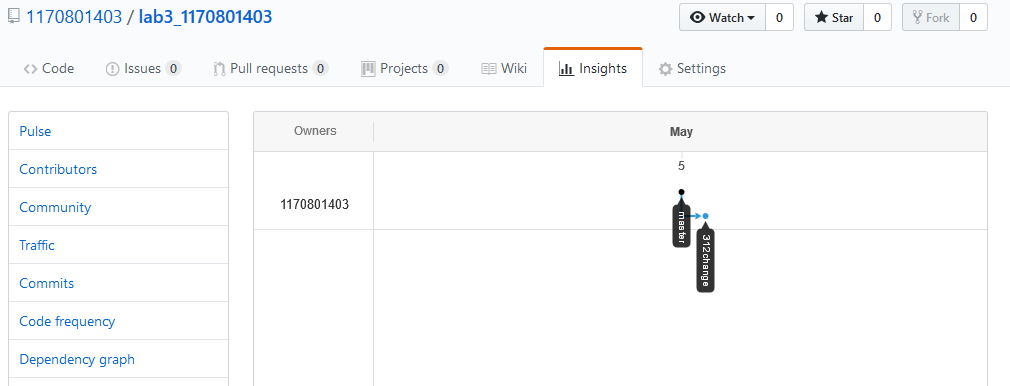
**break**;



图（36）同样一份文件，由于限制了单向关系，便于之前的轨道系统出现差异

## Git仓库结构

请在完成全部实验要求之后，利用Git log指令或Git图形化客户端或GitHub上项目仓库的Insight页面，给出你的仓库到目前为止的Object Graph，尤其是区分清楚312change分支和master分支所指向的位置。



图（37）仓库目前的objectGraph

# 实验进度记录

请使用表格方式记录你的进度情况，以超过半小时的连续编程时间为一行。

每次结束编程时，请向该表格中增加一行。不要事后胡乱填写。

不要嫌烦，该表格可帮助你汇总你在每个任务上付出的时间和精力，发现自己不擅长的任务，后续有意识的弥补。

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 时间段 | 计划任务 | 实际完成情况 |
| 2019.4.18 | 8.00-22.00 | 完成轨道物体、中心物体的类 | 按时完成 |
| 2018.4.21 | 9.00-20.00 | 完成田径比赛、原子结构轨道系统 | 按时完成 |
| 2018.4.22 | 8.00-12.00 | 完成社交网络轨道系统 | 按时完成 |
| 2018.4.23 | 13.00-18.00 | 完成API中熵值、最短逻辑距离函数 | 超时完成 |
| 2018.4.24 | 8.00-12.00 | 完成API中求轨道差异、最短物理距离函数 | 按时完成 |
| 2018.4.24 | 15.00-23.00 | 学习可视化 | 超时完成 |
| 2018.4.25 | 8.00-23.00 | 完成可视化函数 | 按时完成 |
| 2018.4.26 | 8.00-16.00 | 完成socialNetWork特有的功能 | 按时完成 |
| 2018.4.27 | 13.00-18.00 | 完成trackGame编排比赛等功能 | 超时完成 |
| 2018.4.28 | 8.00-20.00 | 完成面向客户的应用 | 按时完成 |
| 2018.4.29 | 14.00-23.00 | 完成312Change中的内容 | 按时完成 |

# 实验过程中遇到的困难与解决途径

|  |  |
| --- | --- |
| 遇到的难点 | 解决途径 |
| 可视化函数不会写 | 通过与同学交流，查阅资料，加上自己动手解决 |
| 对泛型的使用出现问题 | 通过请教老师解决 |
| 对设计模式不熟悉 | 通过网络上的教程自学相关设计模式，并将其用于开发之中，感受到了设计模式的强大 |
| 不会用正则表达式 | 通过网上的教程自学正则表达式同时请教同学 |
| 对异常的处理不够熟练 | 通过学习课件、学习网上课外资料外加动手实践，最后对异常的处理更加得心应手 |

# 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

需要持之以恒的钻研才能写出优质的代码，对道理应该理解深刻，一知半解只会带来更多的麻烦，同时要积极与他人交流

## 针对以下方面的感受

1. 重新思考Lab2中的问题：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？本实验设计的ADT在五个不同的应用场景下使用，你是否体会到复用的好处？

面向ADT编程需要考虑的问题更多。如果ADT设计的足够精良，那么会未今后的开发减少很多麻烦。

1. 重新思考Lab2中的问题：为ADT撰写复杂的specification, invariants, RI, AF，时刻注意ADT是否有rep exposure，这些工作的意义是什么？你是否愿意在以后的编程中坚持这么做？

这些工作的意义是明确每个ADT的作用、属性，为以后其他人的复用做好准备，同时防止客户端破坏代码。应该在以后的编程中坚持这么做。

1. 之前你将别人提供的API用于自己的程序开发中，本次实验你尝试着开发给别人使用的API，是否能够体会到其中的难处和乐趣？

开发别人使用的API考虑的事情更多，考虑事情的层面更加抽象。但是当一个API被成功的开发出来时，成就感也是无法比拟的。

1. 在编程中使用设计模式，增加了很多类，但在复用和可维护性方面带来了收益。你如何看待设计模式？

我认为设计模式时一个很强大的工具，凝聚着前人的智慧，当开发一个功能时，首先应该想到能不能使用设计模式。当然，也不应该为了使用设计模式而使用设计模式，一切以程序开发的要求为准。

1. 你之前在使用其他软件时，应该体会过输入各种命令向系统发出指令。本次实验你开发了一个解析器，使用语法和正则表达式去解析输入文件并据此构造对象。你对语法驱动编程有何感受？

我认为语法驱动编程对于新手来说很有挑战性，但是一旦掌握了语法驱动编程，就能大大提高自己的生产力，同时也充满了成就感。

1. Lab1和Lab2的大部分工作都不是从0开始，而是基于他人给出的设计方案和初始代码。本次实验是你完全从0开始进行ADT的设计并用OOP实现，经过三周之后，你感觉“设计ADT”的难度主要体现在哪些地方？你是如何克服的？

我感觉设计ADT的难度主要体现在它的安全性和可复用性上。在每一步开发之前，我会先考虑之后可能遇到的情况，如果开发方案违背了安全性和可复用性原则，就换其他的开发方案。

1. 你在完成本实验时，是否有参考Lab4和Lab5的实验手册？若有，你如何在本次实验中同时去考虑后续两个实验的要求的？

有参考lab4和lab5的实验手册。将后续两个实验中对异常的处理加入到了本实验中。

1. 关于本实验的工作量、难度、deadline。

本实验的工作量较大，难度适中，deadline略微紧张，但加快进度也能够完成。

1. 到目前为止你对《软件构造》课程的评价。

我从这门课程中学到了很多知识，通过实验也提高了编程技巧，我认为这是一门很好的课程。