



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

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	沈子鸣
学号	1170301007
班号	1703010
电子邮件	2508754153@qq.com
手机号码	18800421860

目录

1 实验目标概述	1
2 实验环境配置	1
3 实验过程	2
3.1 待开发的三个应用场景	2
3.2 基于语法的图数据输入	2
3.2.1 AtomStructure	3
3.2.2 StellarSystem	4
3.2.3 SocialNetworkCircle	4
3.3 面向复用的设计: CircularOrbit<L,E>	6
3.3.1 AtomStructure	7
3.3.2 StellarSystem	7
3.3.3 SocialNetworkCircle	8
3.3.4 接口中的方法介绍	8
3.3.4.1 public void addTrack(Track track)	8
3.3.4.2 public void deleteTrack(Track track)	8
3.3.4.3 public void addCentralObject(L centralObject)	8
3.3.4.4 public void addPhysicalObjectToTrack(E physicalObject, Track track)	8
3.3.4.5 public void addRelationshipBetweenCentralAndPhysical(E physicalObject, boolean fromCentral, double weight)	8
3.3.4.6 public void addRelationshipBetweenPhysicalAndPhysical(E physicalObjectA, E physicalObjectB, double weight)	9
3.3.4.7 public void readFromFile(File file)	9
3.3.4.8 public void constructSocialNetworkCircle(List<PhysicalObject> friends, List<Edge<PhysicalObject>> physicalEdges, Map<PhysicalObject, Double>[] centralEdges)	9
3.3.4.9 public void deleteRelationshipFromPhysicalToPhysical(E physicalObjectA, E physicalObjectB)	9
3.3.4.10 public void deletePhysicalObjectFromTrack(E physicalObject, Track track) throws Exception	9
3.3.4.11 public void resetObjectsAndTrack()	9
3.4 面向复用的设计: Track	9
3.5 面向复用的设计: L	10
3.5.1 Nucleus	10

3.5.2 Star	10
3.5.3 Person	10
3.6 面向复用的设计: PhysicalObject	10
3.6.1 Electron	11
3.6.2 Planet	11
3.6.3 Friend	11
3.7 可复用 API 设计	11
3.7.1 public double getObjectDistributionEntropy(CircularOrbit<L, E> c)	11
3.7.2 public int getLogicalDistance(CircularOrbit<L, E> c, E e1, E e2)	12
3.7.3 public static Map<PhysicalObject, Double> targets(PhysicalObject source, List<Edge<PhysicalObject>> physicalEdges)	12
3.7.4 public double getPhysicalDistance(CircularOrbit<L, E> c, E e1, E e2)	12
3.7.5 public double getPhysicalDistanceFromCentralToObject(CircularOrbit<L, E> c, E e)	12
3.7.6 public Difference getDifference(CircularOrbit<L, E> c1, CircularOrbit<L, E> c2) throws Exception	13
3.7.7 public void planetMovingSimulate(CircularOrbit<L, PhysicalObject> c)	13
3.8 图的可视化: 第三方 API 的复用	13
3.9 设计模式应用	14
3.9.1 Factory method	14
3.9.2 Abstract factory	14
3.9.3 Iterator method	15
3.9.4 Façade method	16
3.9.5 Decorator method	16
3.9.6 Memento method	17
3.10 应用设计与开发	18
3.10.1 StellarSystem	19
3.10.1.1 读取文件	19
3.10.1.2 可视化	20
3.10.1.3 增加轨道	20
3.10.1.4 向特定轨道上增加物体	20
3.10.1.5 删除一条轨道	21
3.10.1.6 删除特定轨道上的某个物体	21
3.10.1.7 计算信息熵	21

3.10.1.8 计算每个行星在时刻 t 时的位置	21
3.10.1.9 计算恒星和某行星之间的物理距离	21
3.10.1.10 计算两行星之间的物理距离	22
3.10.1.11 模拟行星运动动态可视化	22
3.10.1.12 检查轨道合法性	22
3.10.2 AtomStructure	22
3.10.2.1 读取文件	23
3.10.2.2 可视化	23
3.10.2.3 增加轨道	23
3.10.2.4 向特定轨道上增加物体	23
3.10.2.5 删除一条轨道	24
3.10.2.6 删除特定轨道上的某个物体	24
3.10.2.7 计算信息熵	24
3.10.2.8 模拟电子跃迁	24
3.10.2.9 撤销跃迁操作（恢复历史跃迁状态）	24
3.10.3 SocialNetworkCircle	25
3.10.3.1 读取文件	25
3.10.3.2 可视化	25
3.10.3.3 增加轨道	25
3.10.3.4 向特定轨道上增加物体	26
3.10.3.5 删除一条轨道	26
3.10.3.6 删除特定轨道上的某个物体	26
3.10.3.7 计算信息熵	26
3.10.3.8 计算信息扩散度	26
3.10.3.9 增加/删除一条关系	27
3.10.3.10 计算两朋友之间的逻辑距离	28
3.10.3.11 检查轨道合法性	28
3.11 应对应用面临的新变化	29
3.11.1 StellarSystem	29
3.11.2 AtomStructure	30
3.11.3 SocialNetworkCircle	31
3.12 Git 仓库结构	31
4 实验进度记录	32
5 实验过程中遇到的困难与解决途径	32
6 实验过程中收获的经验、教训、感想	34
6.1 实验过程中收获的经验教训	34
6.2 针对以下方面的感受	35

1 实验目标概述

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）。

2 实验环境配置

本次实验环境配置与之前相同。

硬件环境：Intel Core i5-7200U，64 位；4G；

软件环境：Windows 10 家庭中文版

IDE：Eclipse Java 2018-12

关于 Java：JDK 8, JRE 8

关于 Git：Git 2.20.1

```
C:\Users\Shen>java -version
java version "1.8.0_202"
Java(TM) SE Runtime Environment (build 1.8.0_202-b08)
Java HotSpot(TM) 64-Bit Server VM (build 25.202-b08, mixed mode)
```

```
C:\Users\Shen>git --version
git version 2.20.1.windows.1
```

GitHub 仓库的 URL 地址：

<https://github.com/ComputerScienceHIT/Lab3-1170301007>

3 实验过程

3.1 待开发的三个应用场景

- StellarSystem 行星运动模拟
- AtomStructure 原子结构模型
- SocialNetworkCircle 社交网络的好友分布

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：三种轨道系统都有唯一的中心物体和一组轨道，轨道上通常是有物体的。对轨道的操作基本相同，包括增删轨道，对物体的某些操作也相同，比如增删物体。又由于我们需要对轨道系统做可视化，那么轨道物体都应该有一组参数来唯一确定自己在画布上的位置。

不同点：三种轨道中，只有 SocialNetworkCircle 是支持物体间关系存在的。只有 AtomStructure 对轨道上的物体是不做区分的。StellarSystem 被认为一条轨道上只能有一个行星物体存在，并且只有 StellarSystem 支持轨道物体的具体位置的。SocialNetworkCircle 中物体和轨道的所属关系，是由关系网络图中到中心物体的最短路径决定的，其余两个系统都是直接给定物体在哪条轨道上。到了具体实际应用的开发中，还要考虑 AtomStructure 中存在轨道物体跃迁，SocialNetworkCircle 中关系图随关系网络的变化而重构，StellarSystem 中需要设计动态化可视。三者读文件的配置文件要求规范也完全不同。另外，AtomStructure 中需要对跃迁功能设计 Memento 模式。

3.2 基于语法的图数据输入

数据输入是由 void readFromFile(File file)方法来实现的。该方法被定义在轨道系统的接口 CircularOrbit 当中，而三种轨道系统的数据输入又各不相同，所以，在具体轨道系统的类实现中，该方法需要重写。

另外，由经验和分析可知，在读数据的过程中，因为配置文件中的数据只有在符合 spec 要求时才算作是合法数据，所以在读数据时，最好按行读取，按行解析，这样有利于提高效率。而非全部读取后，再做解析。

在解析数据时，对每一行数据都要用事先分析好的可能的所有正则表达式语法来解析，这样也就可以忽略配置文件中不同格式数据顺序不同的影响。

事实上，在轨道系统读数据的同时，根据这些数据构造相应系统，构造和读数据两个行为是捆绑的。

3.2.1 AtomStructure

原子结构模型中有三种不同的数据格式，分别是：

- 元素名称，以“ElementName ::=”开头
- 轨道数目，以“NumberOfTracks ::=”开头
- 各轨道上的电子数量，以“NumberOfElectron ::=”开头，并以 A/B 的形式指出“第 A 条轨道上有 B 个电子”。

所以，用三种不同的正则表达式来匹配每一行数据，line 变量为读文件时的每一行数据，代码如下：

对元素名称解析，得到元素名称变量 char[2] elementName，同时构造一个以 elementName 为 name 变量的 Nucleus，将其作为轨道系统的中心物体：

```
pattern1 = Pattern.compile("ElementName\\s*::=\\s*([a-zA-Z]{1,2})");
matcher1 = pattern1.matcher(line);
while (matcher1.find()) {
    elementName[0] = matcher1.group(1).length() >= 1 ? matcher1.group(1).charAt(0) : null;
    elementName[1] = matcher1.group(1).length() >= 2 ? matcher1.group(1).charAt(1) : null;
    this.addCentralObject(new Nucleus(elementName));
}
```

对轨道数目解析，得到轨道数目变量 int numberOfTrack，同时向轨道系统中添加进 numberOfTrack 条轨道：

```
pattern2 = Pattern.compile("NumberOfTracks\\s*::=\\s*(\\d*)");
matcher2 = pattern2.matcher(line);
while (matcher2.find()) {
    numberOfTrack = Integer.valueOf(matcher2.group(1));
    for (int i = 1; i <= numberOfTrack; i++) {
        this.addTrack(trackFactory.produce(i));
    }
}
```

对轨道上的电子数解析，由于轨道数目不定，所以正则表达式对 A/B 形字串的解析长度不定，故先按照格式全部读取，再以/和；作为分割符，将数据分割成一个数组 String[] numberOfElectronString，该数组长度为偶数，从索引 0 处开始，每相邻两个字串为一对，前者表示第几条轨道，后者表示该条轨道上有几个电子，以此为依据，遍历该字串数据，同时调用添加轨道物体方法 addPhysicalObjectToTrack 向轨道系统中的相应轨道上添加正确数量的物体。

```
pattern3 = Pattern.compile("NumberOfElectron\\s*::=\\s*([0-9/;]*)");
matcher3 = pattern3.matcher(line);
String[] numberOfElectronString = null;
while (matcher3.find()) {
    numberOfElectronString = matcher3.group(1).split("/|;");
    for (int i = 0; i < numberOfElectronString.length; i += 2) {
        numberOfElectron.put(Integer.parseInt(numberOfElectronString[i]),
            Integer.parseInt(numberOfElectronString[i + 1]));
    }
    for (int i = 1; i <= numberOfTrack; i++) {
        for (int j = 1; j <= numberOfElectron.get(i); j++) {
            this.addPhysicalObjectToTrack(electronFactory.produce(), trackFactory.produce(i));
        }
    }
}
```

3.2.2 StellarSystem

行星运动模拟系统中有两种数据格式：

- 中心恒星，以 “Stellar ::=” 开头
- 轨道上的行星，以 “Planet ::=” 开头

因为在正则表达式中对具体数据格式匹配的语法都相同，用全局静态变量来表示这些语法：

```
private static final String numberRegex = "([0-9]*|([0-9]*.[0-9]*|([0-9]*.[0-9]*e[0-9]*))";
private static final String LabelRegex = "([a-zA-Z0-9]*)";
private static final String commaRegex = "\\s*,\\s*";
```

具体数据格式都按照实验手册上的 spec 给出。用两种正则表达式来匹配每一行的数据，假定 line 是每一行数据字符串，则代码如下：

对中心恒星解析，得到恒星的名称，半径（km）和质量（kg）三个参数，用这三个参数来构造一个恒星，并将其加入轨道系统：

```
pattern1 = Pattern.compile("Stellar\\s*::=\\s*" + "<" + LabelRegex + commaRegex + numberRegex
    + commaRegex + numberRegex + ">");
matcher1 = pattern1.matcher(line);
while (matcher1.find()) {
    starName = matcher1.group(1);
    starRadius = parseNumber(matcher1.group(2));
    starMass = parseNumber(matcher1.group(3));
    this.addCentralObject(new Star(starName, starRadius, starMass));
}
```

对轨道上的行星解析，得到行星的名称，状态，颜色，半径（km），所处的轨道半径（km），公转速度（km/s），公转方向（true 表示顺时针，false 表示逆时针），初始角度（该角度是从 x 正半轴逆时针旋转到达行星与恒星所连线的所经过的角度）等参数，用这些参数来构造轨道和行星，将其加入到轨道系统中。

```
pattern2 = Pattern.compile("Planet\\s*::=\\s*" + "<" + LabelRegex + commaRegex + LabelRegex
    + commaRegex + LabelRegex + commaRegex + numberRegex + commaRegex + numberRegex + commaRegex
    + numberRegex + commaRegex + "(CW|CCW)" + commaRegex + numberRegex + ">");
matcher2 = pattern2.matcher(line);
while (matcher2.find()) {
    planetName = matcher2.group(1);
    planetState = matcher2.group(2);
    planetColor = matcher2.group(3);
    planetRadius = parseNumber(matcher2.group(4));
    trackRadius = parseNumber(matcher2.group(5));
    revolutionSpeed = parseNumber(matcher2.group(6));
    revolutionDirection = matcher2.group(7).equals("CW") ? true : false;
    originDegree = parseNumber(matcher2.group(8));
    this.addTrack(trackFactory.produce(trackRadius));
    this.addPhysicalObjectToTrack(
        planetFactory.produce(planetName, planetState, planetColor, planetRadius,
            revolutionSpeed, revolutionDirection, originDegree),
        trackFactory.produce(trackRadius));
}
```

3.2.3 SocialNetworkCircle

个人社交关系网络中有三种数据格式：

- 中心物体（用户），以 “CentralUser ::=” 开头
- 用户的朋友，以 “Friend ::=” 开头
- 社交关系，以 “SocialTie ::=” 开头

因为在正则表达式中对具体数据格式匹配的语法都相同，用全局静态变量来

表示这些语法：

```
private static final String labelRegex = "([a-zA-Z0-9]*)";
private static final String commaRegex = "\\s*,\\s*";
```

社交网络的数据解析难度不大，但是因为社交关系数据中只给出了朋友的名字，所以并不能在一次净读取中直接构造出关系网络（轨道系统），因为如果先读入的关系数据，那么此时还并不知道关系涉及到的朋友信息，不能用来构建轨道系统。所以就在一次读取中，把关系数据存起来。

假定读取的每行数据为 line，解析代码如下：

对用户的解析，获取用户的姓名，年龄和性别数据，用这些来构造一个 Person 对象，将其作为轨道系统的中心物体：

```
pattern1 = Pattern.compile("CentralUser\\s*::=\\s*<" + labelRegex + commaRegex + "(\\d*)"
    + commaRegex + "([M|F]{1})" + ">");
matcher1 = pattern1.matcher(line);
while (matcher1.find()) {
    name = matcher1.group(1);
    age = Integer.valueOf(matcher1.group(2));
    sex = matcher1.group(3).charAt(0);
    this.addCentralObject(new Person(name, age, sex));
}
```

对朋友的解析，获取朋友的姓名，年龄和性别数据，用这些来构造一个 Friend 对象，并将其加入到朋友列表中 List<PhysicalObject> friends，这一变量将在后面用来构造系统。

```
pattern2 = Pattern.compile("Friend\\s*::=\\s*<" + labelRegex + commaRegex + "(\\d*)" + commaRegex
    + "([M|F]{1})" + ">");
matcher2 = pattern2.matcher(line);
while (matcher2.find()) {
    name = matcher2.group(1);
    age = Integer.valueOf(matcher2.group(2));
    sex = matcher2.group(3).charAt(0);
    friends.add(friendFactory.produce(name, age, sex));
}
```

对关系的解析，获取关系的双方朋友的姓名和亲密度，将这些信息放入 Map<String[], Double> adjacency 中，该变量的 key 是一个长度默认为 2 的字符串数组，表示双方朋友名字，value 是一个(0,1]的 double 值，用来表示亲密度。adjacency 变量也是在后面用来构造轨道的。

```
pattern3 = Pattern.compile("SocialTie\\s*::=\\s*<" + labelRegex + commaRegex + labelRegex
    + commaRegex + "(0.\\d{1,3})>");
matcher3 = pattern3.matcher(line);
while (matcher3.find()) {
    Double intimacy = new Double(Double.parseDouble(matcher3.group(3)));
    String names[] = new String[2];
    names[0] = matcher3.group(1);
    names[1] = matcher3.group(2);
    adjacency.put(names, intimacy);
}
```

在处理好配置文件中的数据后，用得到的数据来构造轨道系统。因为我们得到了朋友对象列表和关系字符串数据，通过遍历可以不难构造出轨道系统中的关系表示。

```

for (String[] names : adjacency.keySet()) {
    double infimacy = adjacency.get(names);
    if (names[0].equals(this.centralObject.getName())) {
        for (PhysicalObject friend : friends) {
            if (friend.getName().equals(names[1])) {
                centralEdges[0].put(friend, infimacy);
                this.addRelationshipBetweenCentralAndPhysical(friend, true, infimacy);
                this.addRelationshipBetweenCentralAndPhysical(friend, false, infimacy);
            }
        }
    } else if (names[1].equals(this.centralObject.getName())) {
        for (PhysicalObject friend : friends) {
            if (friend.getName().equals(names[0])) {
                centralEdges[1].put(friend, infimacy);
                this.addRelationshipBetweenCentralAndPhysical(friend, true, infimacy);
                this.addRelationshipBetweenCentralAndPhysical(friend, false, infimacy);
            }
        }
    } else {
        PhysicalObject friendA = null, friendB = null;
        for (PhysicalObject friend : friends) {
            if (friend.getName().equals(names[0])) {
                friendA = friend;
            }
            if (friend.getName().equals(names[1])) {
                friendB = friend;
            }
        }
        physicalEdges.add(new Edge<PhysicalObject>(friendA, friendB, infimacy));
        this.addRelationshipBetweenPhysicalAndPhysical(friendA, friendB, infimacy);
    }
}

```

最后再调用一个封装好的方法

```

public void constructSocialNetworkCircle(List<PhysicalObject> friends, List<Edge<PhysicalObject>> physicalEdges,
    Map<PhysicalObject, Double>[] centralEdges);

```

构造社交网络轨道系统中的轨道和轨道上的物体。

3.3 面向复用的设计：CircularOrbit<L,E>

将 CircularOrbit<L, E>设计成接口，L 表示中心物体的类型，在本实验中为 Nucleus, Star 和 Person 其中之一。E 表示轨道物体的类型，在本实验中被设计成为 PhysicalObject 接口类型，具体轨道系统上的具体轨道物体类型都认为是 PhysicalObject 接口类型。

事实上，虽然不同的轨道系统所拥有的方法并不完全相同，但是因为被设计成了面向接口编程，所以在具体类中去单独写自己独有的方法，是不能被调用的。因此所有轨道系统中的所有方法都要在接口中声明好方法签名，这个策略看起来并不优雅。

此外，设计一个类 ConcreteCircularOrbit<L, E>来应用这个接口，这个类中给出三个轨道系统中共用的方法，而不共用的方法就空着，除了读文件的方法，这个方法虽然三者不是共用的，但三者的读文件在重写时都要调用 ConcreteCircularOrbit<L, E>中的读文件方法，这个父类中规定了，在每次读文件时都要对 rep 重置。

```

@SuppressWarnings("unchecked")
@Override
public void readFromFile(File file) {
    this.centralObject = null;
    this.tracks = new ArrayList<Track>();
    this.objectsInTrack = new HashMap<Track, List<E>>();
    this.relationBetweenCentralAndObject = new Map[2];
    this.relationBetweenCentralAndObject[0] = new HashMap<E, Double>();
    this.relationBetweenCentralAndObject[1] = new HashMap<E, Double>();
    this.relationBetweenObjects = new ArrayList<Edge<E>>();
    checkRep();
}

```

在具体的轨道系统类中先调用父类的 `readFromFile` 方法，像这样：

```

@Override
public void readFromFile(File file) {
    super.readFromFile(file);
}

```

3.3.1 AtomStructure

作为原子结构模型，默认继承父类的 `L` 和 `E` 分别是 `Nucleus` 和 `PhysicalObject`。

该类只重写了接口的 `readFromFile` 和 `Memento` 设计模式中的 `save` 和 `restore` 方法。其余方法在原子结构系统中都被断言为假，并给出提示信息：

```

/**
 * This method is non-existent in atom structure.
 */
@Override
public void addRelationshipBetweenPhysicalAndPhysical(PhysicalObject physicalObjectA,
    PhysicalObject physicalObjectB, double weight) {
    assert false : "shouldn't call this method";
}

```

3.3.2 StellarSystem

作为行星运动模拟，默认继承父类的 `L` 和 `E` 分别是 `Star` 和 `PhysicalObject`。

该类只重写了接口的 `readFromFile`，与行星系统无关的方法都被断言为假，并给出提示信息。该类不能使用的方法和原子结构的差不多，比原子结构模型不能使用的方法多了 `Memento` 设计模式相关的方法。此外，该具体类中还定义了一个静态 `public` 方法，和一个静态 `private` 方法，

```

public static double parseNumber(String s) throws Exception

private static double parseScientific(String s) throws Exception

```

`parseNumber` 用来将一个表示数字的字串 `s` 转换成对应的真实小数。因为这个 `s` 可能是科学记数法表示的字串，在 `parseNumber` 中还要用 `if-else` 来判断，如果包含 `e` 的话，调用 `private` 方法 `parseScientific` 来用科学记数法规则转换。

parseNumber 可以在其它地方用静态方法来调用。

3.3.3 SocialNetworkCircle

作为社交关系网络,默认继承父类的 L 和 E 分别是 Person 和 PhysicalObject。

该类重写了接口的 readFromFile 和 constructSocialNetworkCircle 方法,除了 Memento 设计模式相关的方法被断言为假外,都能使用。该类还自己定义了一个 public 的静态方法,

```
public static Map<PhysicalObject, Integer> getSocialTrackRadius(List<PhysicalObject> friends,  
    List<Edge<PhysicalObject>> physicalEdges, Map<PhysicalObject, Double>[] centralEdges) {
```

该方法可以根据轨道物体,轨道物体之间的邻接关系和轨道物体和中心物体之间的关系来构造得出轨道和物体之间的所属关系,这其中用了广度优先搜索来构造最短路径。该方法可以在其他地方用静态方法来调用。

3.3.4 接口中的方法介绍

各 observer 方法就不介绍了。

3.3.4.1 public void addTrack(Track track)

将轨道添加到轨道系统中。在添加时已经对存储轨道的 list 做好按轨道半径的升序排序。

3.3.4.2 public void deleteTrack(Track track)

从轨道系统中删除轨道。

3.3.4.3 public void addCentralObject(L centralObject)

将中心物体添加到圆形轨道中。这个方法应该只在有中心物体的轨道系统中调用。

3.3.4.4 public void addPhysicalObjectToTrack(E physicalObject, Track track)

将轨道物体添加到轨道系统中。在添加时,已经对相应的轨道 list 做好按物体的角度的升序排序。

3.3.4.5 public void addRelationshipBetweenCentralAndPhysical(E physicalObject, boolean fromCentral, double weight)

在中心物体和具有亲密度的轨道物体之间添加关系。这种方法只能在有关系

的轨道系统上调用，其关系主要是社交关系网络。

3.3.4.6 public void addRelationshipBetweenPhysicalAndPhysical(E physicalObjectA, E physicalObjectB, double weight)

添加两个轨道物体之间的关系，亲密度为 `weight`。此方法应仅在具有关系的轨道系统中被调用。

3.3.4.7 public void readFromFile(File file)

所有具体的轨道系统的该方法都是不同的，因此这种方法将在每个具体的类中被重写。但无论是什么样的轨道系统，在从文件中读取之前，必须重置它的表示属性。

3.3.4.8 public void constructSocialNetworkCircle(List<PhysicalObject> friends, List<Edge<PhysicalObject>> physicalEdges, Map<PhysicalObject, Double>[] centralEdges)

根据轨道物体列表，轨道物体和中心物体之间的双向带权边表示来构造关系网络。这个方法应该只在读取文件或重构社交关系网络时被调用。

3.3.4.9 public void deleteRelationshipFromPhysicalToPhysical(E physicalObjectA, E physicalObjectB)

删除两个轨道物体之间的关系。

3.3.4.10 public void deletePhysicalObjectFromTrack(E physicalObject, Track track) throws Exception

从特定的轨道上删除某轨道物体。

3.3.4.11 public void resetObjectsAndTrack()

删除轨道中的所有轨道物体。此方法应仅在重构轨道系统时被调用。

3.4 面向复用的设计：Track

首先，Track 要求被设计成 IMMUTABLE 的类，而对于轨道来说，只有半径

是值得我们关注的属性。又因为后期有要在行星系统中添加椭圆轨道的需求，或者其他种类轨道的需求，这里将 `Track` 设计成接口，然后将圆形轨道设计成具体类。因为轨道的半径参数是必须的，将其放入构造器中初始化，又半径是 `double` 类型的，规定为 `private final` 后，再用 `observer` 获取时，也就不需要做防御式拷贝了，这个类自然就是 `IMMUTABLE` 的了。

3.5 面向复用的设计：L

首先，`L` 也被要求设计成 `IMMUTABLE` 的类，而对于中心物体来说，不同的轨道系统的中心物体都不同，对此并未有设计模式的要求，那就简单地直接设计三个中心物体类。这三个类中涉及到的所有 `rep` 都是基本类型和 `String` 类型，都声明为 `private` 和 `final` 后，没有 `mutator` 方法，这个类就是 `IMMUTABLE` 的了。

3.5.1 Nucleus

原子结构模型中的操作几乎不涉及中心物体，而配置文件中给出了原子结构的原子元素，那这里原子核的 `rep` 就只有一个元素名称。元素名称要满足 `spec` 条件，字符长度最多为 2，如果是一个字符，那么该字符必须大写，如果是两个字符，那么第一个字符大写，第二个字符小写。

还可以对这个类重写一下 `toString` 方法，获得它的元素名称字符串表示。

3.5.2 Star

行星轨道系统中的中心物体是恒星，配置文件中给出了恒星的名称，半径和质量属性，那么就以这三样作为恒星的 `rep`。其中，名称是 `String` 类型，名称中只包含字母和数字而不包含任何空白符或符号。半径的默认单位是千米，质量的默认单位是千克。重写该类的 `equals` 和 `hashCode` 方法，规定两个有相同名称的恒星是相同的。

3.5.3 Person

社交关系网络的中心物体是人，配置文件中给出了人的姓名，年龄和性别，应该以这三样作为 `rep`，但是注意到社交关系网络中的中心物体和轨道物体的性质是相同的，即 `rep` 应该相同（除了轨道物体需要确定位置而有的角度 `rep`），所以这里不妨使 `Person` 类继承 `Friend` 类（轨道物体类），令这两个类都设计成 `IMMUTABLE` 的，还可以重写 `Person` 类的 `equals` 和 `hashCode` 方法，规定两个有相同姓名，年龄相同并且性别相同的人是相同的。

3.6 面向复用的设计：PhysicalObject

首先，`PhysicalObject` 也被要求设计成 `IMMUTABLE` 的类，而对于轨道物体来说，不同的轨道系统的轨道物体都不同，对此要求做面向复用的设计，对此可

以将 `PhysicalObject` 设计成接口，然后再设计具体轨道物体类应用接口。注意因为是 IMMUTABLE 的，所以接口中绝不应该有 `mutator` 方法声明。

另外，和 `CircularOrbit<L, E>` 接口所遇到的麻烦相同，所有具体类中的方法，不管是不是共用的方法，都需要在接口中声明才能被正常调用。

因为在可视化中，轨道物体是要画在画布上特定位置的，所以所有的 `PhysicalObject` 具体类中都应该有 `degree` 属性，`degree` 是以 x 正半轴开始，逆时针旋转到轨道物体与中心物体连线处所转过的最小角度， $\in [0, 360)$ 。

3.6.1 Electron

`Electron` 是原子结构中的轨道物体类型。原子结构中，所有的电子都是一样的，没有独特的属性，所以只有一个 `degree rep`。除了获取 `degree` 的 `observer` 方法以外，其它接口中的方法在此都应该被断言为假。

3.6.2 Planet

`Planet` 是行星轨道模型中的轨道物体类型。按照配置文件给出的数据，一个行星应该有名称，状态，颜色，半径，公转速度，公转方向，角度（位置）等几个属性，除了这些属性的 `observer` 方法，没有其它方法。重写 `equals` 和 `hashCode` 方法，规定只有当两个行星的名字是相同的时候，两个行星才是相同的。

3.6.3 Friend

`Friend` 是社交关系网络中的轨道物体类型。只有姓名，年龄，性别和角度（位置）四个属性，这个类还是被社交关系网络中的中心物体 `Person` 继承的。该类中除了各 `rep` 的 `observer` 方法没有其它方法了。重写 `equals` 和 `hashCode` 方法，规定只有当两个朋友的姓名、年龄和性别都是相同的时候，这两个朋友才是相同的。

3.7 可复用 API 设计

可复用的 API 都包含在类 `CircularOrbitAPIs` 中。

`CircularOrbitAPIs` 包括客户端可在具体应用程序中使用的所有操作。当客户端想要使用这些函数操作时，需要实例化一个 `CircularOrbitAPIs` 对象。并非 `CircularOrbitAPIs` 中的所有函数操作都可用于各种轨道系统。中心和轨道物体的组合应由客户确定，也就是 `L` 和 `E` 的类型组合应由客户端确定，建议此组合适合特定的应用场景。

3.7.1 `public double getObjectDistributionEntropy(CircularOrbit<L, E> c)`

计算轨道系统中物体的信息熵。轨道物体越分散，熵越大。计算方法描述如下：如果这个轨道系统有 `n` 条轨道，第 `i` 个轨道上有 `ai` 个对象，假设轨道物体的总数是 `m`，那么分布熵由 $\sum ((a_i / m) * \log(a_i / m))$ 计算而得。

3.7.2 public int getLogicalDistance(CircularOrbit<L, E> c, E e1, E e2)

计算轨道系统中两轨道物体之间的逻辑距离。设两轨道物体为 $e1$ 和 $e2$ ，要计算从 $e1$ 到 $e2$ 的逻辑距离，即计算在轨道系统 c 中 $e1$ 到 $e2$ 的最短路径长度。该方法中修改了 Lab2 中 Graph 里面的计算最短路径距离的代码并拿来使用求最短路径长度。该方法应该只在有关系的轨道系统中被调用。

值得一提的是，这个方法还可以兼容获取中心物体和轨道物体之间的逻辑距离。虽然广泛地说，中心物体和轨道物体的类型是不相同的，而这个方法的第二个和第三个参数类型都是 E ，但是，可以传入一个伪中心物体。该伪中心物体具有较为独特的特征，规定伪中心物体是轨道物体类型 E ，在社交关系网络中，它使一个姓名为 `center`，年龄为 1，性别为 `M` 的朋友对象。

之所以设计一个伪中心物体，是因为在该方法的具体实现中，内置了一个伪中心物体，取代了关系网络中的真正的中心物体的位置，这么设计是便于处理在中心物体和轨道物体的类型不相同的情况下，仍然可以做到由一轨道物体，经过中心物体而到达另一轨道物体的路径遍历。如果遍历的对象类型不相同的话，是很麻烦的。

3.7.3 public static Map<PhysicalObject, Double> targets(PhysicalObject source, List<Edge<PhysicalObject>> physicalEdges)

该方法被声明为静态方法，主要目的是在该类中被调用，来获取一个邻接关系映射，该映射是从轨道物体到亲密度度的映射，该映射表的 `key` 是和轨道物体 `source` 有关系的其他轨道物体，在有向关系网中，和轨道物体 `source` 有关系，指的是从 `source` 到其他物体的关系。事实上，该方法在 Lab2 中的 Graph 中已经被实现过，但这里和那里的 `targets` 方法参数不同，具体实现稍有不同，稍加改动，在这里就可以使用了。该方法应该只在有关系的轨道系统中使用，并且建议通过静态方法调用。

3.7.4 public double getPhysicalDistance(CircularOrbit<L, E> c, E e1, E e2)

计算轨道系统中两轨道物体之间的物理距离。设两轨道物体分别为 $e1$ 和 $e2$ 。物理距离的计算方法是计算两物体在平面坐标系中两点坐标之间的距离，坐标可以根据物体所在轨道半径和物体的 `degree` 属性唯一确定。该方法应仅在具有精确轨道物体位置的轨道系统中被调用。在恒星系统中，物理距离的单位是千米。

3.7.5 public double getPhysicalDistanceFromCentralToObject(CircularOrbit<L, E> c, E e)

计算轨道系统中某轨道物体到中心物体之间的物理距离。设该轨道物体为 e 。事实上，该物理距离的计算方法和 `getPhysicalDistance` 无异，但考虑到中心物体和轨道物体的参数可能不同，故分成两个方法来求轨道系统中任意两个物体之间

的物理距离。该方法应仅在具有精确轨道物体位置的轨道系统中被调用。在恒星系统中，如果轨道是圆形的，那么该方法实际是在求轨道物体 *e* 所在轨道的半径，物理距离的单位是千米。

3.7.6 **public Difference getDifference(CircularOrbit<L, E> c1, CircularOrbit<L, E> c2) throws Exception**

获得两个轨道系统之间的差异。这两个轨道系统必须是同一类型，不然也就没有比较的意义了。在这里有一个新的 ADT **Difference**。**Difference** 类规定了轨道差异都包含了哪些内容。差异包括轨道数量的差异，相同轨道位置中的对象数量差异，如果这种轨道系统中轨道物体之间是有差异的，那么还包括同一轨道位置中两组对象之间的差异，**Difference** 类中有一个重写的 **toString** 方法，该方法将这些差异以字串的形式输出，可读性高。

轨道数量的差异可以通过获取轨道 **list** 的大小相减得到。而各轨道上的物体数量差异，需要用两部分遍历来得到。假设 *c1* 轨道数量为 *t1*，*c2* 轨道数量为 *t2*，而 *t1*<*t2*，那么第一部分遍历应该是从第 1 条轨道到第 *t1* 条轨道，之后在第二遍遍历从第 *t1*+1 条轨道到第 *t2* 条轨道。实际上，第二次遍历只考虑了 *c2* 中的轨道物体，因为这部分轨道在 *c1* 中是不存在的，但如果不这么做，就会出现空指针异常。

如果轨道物体是有差异可分的，那么 **Difference** 中还要包括这些差异。这部分的差异表示是，在某条轨道上，如果 *c1* 有的物体而 *c2* 没有，则加入集合 *A* 中，如果 *c2* 有的物体而 *c1* 没有，则加入集合 *B* 中，结果以 *A-B* 的形式输出。

3.7.7 **public void planetMovingSimulate(CircularOrbit<L, PhysicalObject> c)**

模拟恒星系统中的行星运动。此方法应仅在行星轨道系统中被调用。该方法中还实例化了一个 **private** 类 **VisualPanel**，这个 **VisualPanel** 类是绘制动画面板的。

3.8 图的可视化：第三方 API 的复用

这部分代码参考了个人在假期时学习《Java 轻松学》中的可视化动画代码。主要使用了 Java 中的 **java.awt** 和 **javax.swing** 中的一些库。本实验中可视化有两种，一个是三个轨道系统共用的静态可视化，一个是行星轨道模拟运动的动态可视化。静态可视化由 **visualize** 函数全部承担，放在 **CircularOrbitHelper** 类中。

CircularOrbitHelper 类中有一个 **visualize** 函数用于可视化，还有两个封装的内部方法 **drawRelationBetweenCentralAndObject** 和 **drawRelationBetweenObjects** 用来在画布上绘制关系边，这两个方法都是静态的，且对外不可见的，仅用来辅助 **visualize**，提高可读性。

在 **visualize** 中要画中心物体，轨道，轨道上的物体，如果轨道系统中有关系的话，还要画出这些关系。按照参数确定比例，很容易可以画出中心物体和轨道，要画轨道上的物体的话，需要获取轨道物体的 **degree** 参数，由 **degree** 和轨道半径来唯一确定坐标，确保物体绘制在轨道上。至于关系的绘制，首先要判断轨道

系统是否为社交关系网络（在本实验中），如果是，才能调用两个 `private static` 方法来绘制。如果不是，那就不应该绘制。这两个 `private static` 中包含了对关系操作的调用，这些调用在除社交关系网络的轨道系统中是被断言为假的，所以在这些轨道系统中是不能被调用的，否则会报 `AssertionError` 错误。

至于行星运动可视化模拟，是 `planetMovingSimulate` 方法，我把它放在了 `CircularOrbitAPIs` 类中，作为一个具体应用 API。该方法只有在行星轨道系统中调用。原理是设置一个定时器，每隔一段延迟时间，就按照经过的时间计算一下当前位置，同时绘制当前帧的图像。

3.9 设计模式应用

3.9.1 Factory method

在构造 `Track` 和 `PhysicalObject` 对象时，使用工厂模式，可以避免这些对象暴露给客户端，实际上就是用一个间接的类来封装 `Track` 和 `PhysicalObject` 的构造器。拿 `Track` 的构造为例，要构造一个 `Track` 类型的 `NormalTrack` 对象，只需设计一个工厂类 `TrackFactory`，并设计一个 `produce` 方法，该方法返回一个 `Track` 类型对象即可。

```
/**
 * Produce a track.
 *
 * @param radius radius of track
 * @return a Track instance
 */
public Track produce(double radius) {
    return new NormalTrack(radius);
}
```

3.9.2 Abstract factory

在构造 `ConcreteCircularOrbit` 对象时，可以采用抽象工厂方法。工厂方法可以避免构造对象直接暴露给客户端。另一方面，因为 `ConcreteCircularOrbit` 下属有三个子类，若设计成简单工厂，则三个子类的工厂生产方法都在同一个工厂类下，那么如果今后某一个子类或其构造发生改变，那么需要访问这个工厂类去变化。如果采用抽象工厂方法，把 `ConcreteCircularOrbit` 的生产工厂设计成抽象工厂类，而三个子类的生产工厂设计成具体工厂类，可以有针对性地应对今后的改变。

这个抽象工厂类是这样的：

```

public abstract class circularOrbitFactory<L, E> {

    /**
     * Constructor
     */
    public circularOrbitFactory() {

    }

    /**
     * Produce a circular orbit in certain class. This should be override in
     * particular classes.
     *
     * @return a circular orbit
     */
    public abstract CircularOrbit<L, E> produce();
}

```

而三个具体工厂类继承了抽象工厂类，以原子结构模型工厂为例：

```

public class AtomStructureFactory extends circularOrbitFactory<Nucleus, PhysicalObject> {

    /**
     * Constructor
     */
    public AtomStructureFactory() {

    }

    /**
     * Create a atom structure.
     */
    @Override
    public CircularOrbit<Nucleus, PhysicalObject> produce() {
        return new AtomStructure();
    }

}

```

3.9.3 Iterator method

在 ConcreteCircularOrbit 中使用迭代器模式，希望得到的效果是，可以用 iterator 方法或 for-each 来迭代一个 CircularOrbit 对象，得到的迭代结果是遍历轨道系统中的所有轨道对象，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由小到大的次序，若不考虑绝对位置，则随机遍历。

为此，只要是 ConcreteCircularOrbit 应用 Iterable 接口，并重写 iterator 方法即可。在重写 iterator 方法中，返回一个 MyIterator 对象，而 MyIterator 类是应用了 Iterator 接口的，也就是说，MyIterator 对象中具有重写过的 next 和 hasNext 方法，如果需要其它功能，还可以重写 remove 等父类的方法。如果需要明确表示不能调用 remove 方法，那么最好重写 remove，并断言为假：

```

@Override
public void remove() {
    assert false : "doesn't support remove";
}

```

因为轨道物体被轨道分为若干个小集合，在 MyIterator 要想实现重写 hasNext

方法，可以使用两个游标索引 `trackIndex` 和 `objectIndex`。前者是当前遍历的轨道数游标，后者是当前正在遍历的轨道中的物体遍历游标。每当一组轨道上的物体被遍历完成，转入下一个轨道遍历时，`objectIndex` 应该置 0，并且 `trackIndex` 加 1。如此一来，`hasNext` 的重写规则就可以确定了。

```
@Override
public boolean hasNext() {
    if (trackIndex >= tracks.size() || objectIndex >= objectsInTrack.get(tracks.get(trackIndex)).size()) {
        return false;
    } else {
        return true;
    }
}
```

而 `next` 的重写规则，只要按序拿出相应的轨道物体就好了，并适时更新两个游标。

```
@Override
public E next() {
    List<E> objects = objectsInTrack.get(tracks.get(trackIndex));
    E e = objects.get(objectIndex);
    objectIndex++;
    if (trackIndex < tracks.size() - 1 && objectIndex >= objects.size()) {
        trackIndex++;
        objectIndex = 0;
    }
    return e;
}
```

这里，遍历顺序的确定实际上没有在迭代器中规定。而是在 `ConcreteCircularOrbit` 中的 `addTrack` 和 `addPhysicalObjectToTrack` 方法中规定的。在这两个方法中，添加轨道和轨道物体时，已经在相应的 `list` 中做好了排序。那么如果按照游标索引的增序拿出相应的轨道物体的顺序，也就是我们期望的顺序了。

3.9.4 Façade method

外观设计模式就是把功能性函数都封装在一个类里，需要使用的时候，实例化一个这个封装类的对象，然后用这个对象调用方法。既然我们的所有的具体应用的 API 都放到了 `CircularOrbitAPIs` 中，那么外观模式也算是做到了吧。

3.9.5 Decorator method

其实给行星加卫星的需求并没有在具体应用中调用，而且 `spec` 也没有给这个需求太多说明。不过既然要在这里用到修饰模式，那就要把行星的修饰属性卫星提炼出来，设计一个卫星修饰抽象类 `SatelliteDecorator` 应用 `PhysicalObject` 接口，表示具有卫星的行星仍然具有 `PhysicalObject` 的特性。这个类中有一个 `rep` 是用来做 `delegation` 的，

```
protected final PhysicalObject planet;
```

在构造时指派 `delegation` 关系，

```

/**
 * Constructor.
 *
 * @param planet delegation from PhysicalObject
 */
public SatelliteDecorator(PhysicalObject planet) {
    this.planet = planet;
}

```

然后设计具体修饰类 PlanetWithSatellite, 该类继承 SatelliteDecorator 抽象类, 在构造时, 调用父类方法以传入的参数指派 delegation 关系, 另外, 因为这个类表示具有一组卫星的行星, 所有它的 rep 有一个卫星的 list, 每构造一次应该向该 list 中加入一个卫星, 如果需要加入多个卫星, 按照 decorator 设计模式, 应该多层嵌套使用 decorate。

```

/**
 * Constructor. Extends it's super-class.
 *
 * @param planet
 */
public PlanetWithSatellite(PhysicalObject planet) {
    super(planet);
    addSatellite();
}

```

剩下的就是用 delegation 来实现接口中的各方法, 例如:

```

@Override
public double getDegree() {
    return this.planet.getDegree();
}

```

3.9.6 Memento method

我的设计是使用 memento 设计模式, 仅管理电子跃迁的历史状态, 而对其它的轨道系统的改变, 不保存历史状态。比如, 如果轨道系统先删除了某个电子, 再执行一次电子跃迁, 那么如果此时利用 memento 回溯一个历史版本的话, 轨道系统将恢复到删除电子之前的状态。

ConcreteCircularOrbit<L, E>作为需要保存的状态类, 它的 rep 是需要保存的状态数据。该类中应该有 save 和 restore 方法。save 方法用来将当前状态存入一个 Memento 类对象, 并返回这个对象。restore 方法会传入一个 Memento 对象参数, 用来将 ConcreteCircularOrbit 对象的 rep 置为 Memento 中的状态数据。而 Memento 类应该具有和 ConcreteCircularOrbit 相同的 rep, 并且获取这些 rep 的 observer 需要做防御式拷贝, 否则, 在 restore 时会发生表示泄露, 会引起表示泄

露相关的 bug，这点要尤其注意。

然后还要做一个 `Caretaker<L, E>` 类，用来保存历史状态，也就是有一个 `Memento` 的 `list` 作为 `rep`。该类中有一个 `mutator` 方法，用来把一个 `Memento` 参数加入到 `rep` 中。还有一个方法用来获取历史状态，因为历史状态存储在一个 `list` 中，而加入 `list` 时默认是在后面加入 `list`，所以按照 `list` 索引从小到大，是由旧到新的历史状态。获取历史状态方法中传入一个 `int i` 参数，可以通过 `list` 的 `get` 方法，获取相应位置的 `Memento`，比如，如果想要回溯 `i` 次版本，那么应该调用 `list.get(list.size() - i - 1)`。

```
/**
 * Observer. Get a memento in order to roll back to a historical version.
 *
 * @param i the number of version rolled back
 * @return the memento after i versions rolled back
 * @throws Exception the number of versions rolled back is larger than the
 *                  version total.
 */
public Memento<L, E> getMemento(int i) throws Exception {
    if (mementos.size() - 1 < i) {
        throw new Exception("Cannot rollback so many back!");
    }
    return this.mementos.get(this.mementos.size() - i - 1);
}
```

如此，用 `Caretaker` 来管理历史状态，在需要记录当前状态的时刻调用

```
caretaker.addMemento(atomStructure.save());
```

在需要回溯版本（撤销）时调用

```
atomStructure.restore(caretaker.getMemento(countdownVersion));
```

`Caretaker` 类和被管理版本的 `ConcreteCircularOrbit` 类通过 `Memento` 类互相传递调用，完成状态版本的记录与恢复功能。

3.10 应用设计与开发

首先，给用户的具体应用开发全在 `Main.java` 中，该文件中有一个 `main` 函数，是整个应用的开始，还有一个内部的菜单显示函数。用户通过选择 1、2 或 3，来分别选择执行哪个应用。为了提高可读性，将具体场景应用代码封装成类，用静态方法的形式来调用。

```

/**
 * main
 *
 * @param args
 * @throws Exception deal with some exception when user's input disobey
 *                   scientific notation rules.
 */
public static void main(String[] args) throws Exception {
    menu();
    int choose = in.nextInt();
    switch (choose) {
        case 1:
            AtomStructureAPP.application();
            break;
        case 2:
            StellarSystemAPP.application();
            break;
        case 3:
            SocialNetworkCircleAPP.application();
            break;
        default:
            break;
    }
}

```

3.10.1 StellarSystem

有关行星轨道系统的代码在 StellarSystemAPP 中，该类中有一个 public static 的 application 函数，在 main 中被调用，执行应用代码，还有两个内部的菜单显示函数。大体上，行星轨道系统的应用功能如菜单显示如下，共 12 个功能：

```

/**
 * Menu in stellar system application which indicates users' choices
 */
private static void menu() {
    System.out.println("1. Read from file to generate a atomic structure.");
    System.out.println("2. Visualize.");
    System.out.println("3. Add a track.");
    System.out.println("4. Add an object to a track.");
    System.out.println("5. Delete a track.");
    System.out.println("6. Delete an object in a track.");
    System.out.println("7. Calculate the information entropy of the system.");
    System.out.println("8. Calculate every planet's position at time t.");
    System.out.println("9. Calculate the physical distance between the star and another planet.");
    System.out.println("10. Calculate the physical distance between two planets.");
    System.out.println("11. Simulate movement in GUI.");
    System.out.println("12. Check legality.");
}

```

3.10.1.1 读取文件

功能一：读取文件，并根据文件数据构造一个轨道系统。

由于我们有三种不同数据规模的文件，可以让用户选择读取哪一个，读取功能直接调用轨道系统的方法即可。


```

case 1: // Read from file to generate a atomic structure
    readMenu();
    int choose1 = in.nextInt();
    switch (choose1) {
        case 1:
            stellarSystem.readFromFile(new File("input/StellarSystem.txt"));
            break;
        case 2:
            stellarSystem.readFromFile(new File("input/StellarSystem_Medium.txt"));
            break;
        case 3:
            stellarSystem.readFromFile(new File("input/StellarSystem_Larger.txt"));
            break;
        default:
            System.out.println("Wrong input");
            break;
    }
    break;

```

3.10.1.2 可视化

功能二：行星轨道可视化。

直接调用 CircularOrbitHelper 类中的 visualize 方法，传入构造好的行星轨道系统参数即可。

```

case 2: // Visualize
    CircularOrbitHelper.visualize(stellarSystem);
    break;

```

3.10.1.3 增加轨道

功能三：增加一条轨道。

对于用户来说，要给定增加轨道的半径。行星轨道的半径需要时 double 类型以尽可能精确。然后调用轨道系统的方法即可。

```

case 3: // Add a track
    System.out.println("What's the radius(decimal) of the added track?");
    double addTrackRadius = in.nextDouble();
    stellarSystem.addTrack(trackFactory.produce(addTrackRadius));
    break;

```

3.10.1.4 向特定轨道上增加物体

功能四：向特定轨道上增加物体。

对于用户来说，向特定轨道上增加物体需要给定轨道半径，以及要增加物体的相关信息。这里的轨道必须是一条已存在的轨道。如果这条轨道不存在，或者这个物体已存在，那么都将添加失败，但是并没有给出提示失败的信息。

物体信息要求用户以严格的语法输入，代码中会对这句输入做和读取文件时

相同的语法解析。

最后添加时调用轨道系统的方法即可。

3.10.1.5 删除一条轨道

功能五：删除一条轨道。

对于用户来说，要给定删除轨道的半径。行星轨道的半径需要时 `double` 类型以尽可能精确。然后调用轨道系统的方法即可。

3.10.1.6 删除特定轨道上的某个物体

功能六：删除特定轨道上的某个物体。

对于用户来说，删除特定轨道上的某个物体需要给定轨道半径，以及要删除物体的相关信息。这里的轨道必须是一条已存在的轨道。如果这条轨道不存在，或者这个物体不存在，那么都将删除失败，但是并没有给出提示失败的信息。

物体信息要求用户以严格的语法输入，代码中会对这句输入做和读取文件时相同的语法解析。

最后删除时调用轨道系统的方法即可。

3.10.1.7 计算信息熵

功能七：计算轨道系统的信息熵。

这点功能是三个轨道系统所共有的，具体执行是调用的 `CircularOrbitAPIs` 类中的方法。计算方法见 3.7.1 节中的

```
public double getObjectDistributionEntropy(CircularOrbit<L, E> c)。
```

3.10.1.8 计算每个行星在时刻 t 时的位置

功能八：计算每个行星在时刻 t 时的位置。

实际上，这个位置是由行星所处轨道半径，和行星的 `degree` 参数唯一确定的。在圆轨道上的行星运动，假定行星是以 `speed` 参数（单位 `km/s`）做匀速运动，那么根据轨道半径 `radius` 和经历时间 `time`，可以求出偏转角 `theta`。再根据行星的 `direct` 参数，可以确定公转方向，从而在 `degree` 参数上加减偏移量 `theta`，得出新的 `degree`。

考虑到这是行星运动，输入的时间 t 的单位被认为是天数，但由于配置文件中的数据除了地球全是瞎编的，所以也只有地球测试数据是有实际参考价值的了。

3.10.1.9 计算恒星和某行星之间的物理距离

功能九：计算恒星和某行星之间的物理距离。

因为本实验中的轨道是圆轨道，所以恒星和行星之间的距离就是轨道半径了。

3.10.1.10 计算两行星之间的物理距离

功能十：计算两行星之间的物理距离。

同样，这个物理距离是在平面坐标系上的两点直线距离，而两点坐标是由行星所在轨道半径 `radius` 和行星参数 `degree` 唯一确定的。利用三角函数知识不难由 `radius` 和 `degree` 求出坐标 (x, y) 。由于行星轨道系统中相邻轨道的半径长度比往往是 10:1 以上，甚至是 $1e9:1$ 的这种巨大比例，所以求得的物理距离其实和外圈轨道的轨道半径相差不多。

3.10.1.11 模拟行星运动动态可视化

功能十一：模拟行星运动动态可视化。

直接调用 `CircularOrbitAPIs` 中的 `planetMovingSimulate` 方法。

3.10.1.12 检查轨道合法性

功能十二：检查轨道合法性。

事实上，一部分轨道合法性已经在代码层面中 RI 检查 `checkRep` 时完成了一部分。按照要求，再检查以下几点：

- 中心点必须有一颗恒星
 - 一个轨道上只能有一个行星且不能没有行星
 - 相邻轨道的半径之差不能小于两颗相应行星的半径之和
- 对此，将这三点的检查做成一个功能，供用户调用。

```
case 12: // Check legality
    if (stellarSystem.getCentralObject() == null) {
        throw new Exception("The central object is null!");
    }
    for (Track track : stellarSystem.getTracks()) {
        if (stellarSystem.getObjectsInTrack().get(track).size() != 1) {
            throw new Exception("There must be one single planet in one track!");
        }
    }
    for (int i = 0; i < stellarSystem.getTracks().size() - 1; i++) {
        Track track = stellarSystem.getTracks().get(i);
        Track track_ = stellarSystem.getTracks().get(i + 1);
        double trackRadiusSum = track.getRadius() + track_.getRadius();
        double planetRadiusSum = stellarSystem.getObjectsInTrack().get(track).get(0).getRadius()
            + stellarSystem.getObjectsInTrack().get(track_).get(0).getRadius();
        if (planetRadiusSum >= trackRadiusSum) {
            throw new Exception(
                "The sum of radius of two neighbouring tracks should be more than the sum of
            }
        }
    }
    break;
```

3.10.2 AtomStructure

有关原子结构模型的代码在 `AtomStructureAPP` 中，该类中有一个 `public static` 的 `application` 函数，在 `main` 中被调用，执行应用代码，还有两个内部的菜单显示函数。大体上，原子结构模型的应用功能如菜单显示如下，共 9 个功能：

```

/**
 * Menu in atom structure application which indicates users' choices.
 */
private static void menu() {
    System.out.println("1. Read from file to generate a atomic structure.");
    System.out.println("2. Visualize.");
    System.out.println("3. Add a track.");
    System.out.println("4. Add an object to a track.");
    System.out.println("5. Delete a track.");
    System.out.println("6. Delete an object in a track.");
    System.out.println("7. Calculate the information entropy of the system.");
    System.out.println("8. Transition.");
    System.out.println("9. Restore transition.");
}

```

3.10.2.1 读取文件

功能一：读取文件。根据文件数据构造一个原子结构模型。

由于我们有三种不同数据规模的文件，可以让用户选择读取哪一个，读取功能直接调用轨道系统的方法即可。

3.10.2.2 可视化

功能二：可视化。

直接调用 CircularOrbitHelper 类中的 visualize 方法，传入构造好的原子结构模型参数即可。

3.10.2.3 增加轨道

功能三：增加一条轨道。

对于用户来说，要给定增加轨道的半径。原子结构中的轨道没有实际意义的半径参数，在读取文件信息时，第 i 条轨道的半径就设置为 i 。所以这里用户给定的轨道参数是 `int` 就好了，可以认为 `addTrackRadius` 是第几条轨道的索引。

```

case 3: // Add a track
    System.out.println("What's the radius(integer) of the added track?");
    int addTrackRadius = in.nextInt();
    in.nextLine();
    atomStructure.addTrack(trackFactory.produce((double) addTrackRadius));

```

3.10.2.4 向特定轨道上增加物体

功能四：向特定轨道上增加物体。

对于用户来说，向特定轨道上增加物体需要给定轨道半径，而因为所有的电子都相同，物体的相关信息是不需要的。这里的轨道必须是一条已存在的轨道。如果这条轨道不存在，那么将添加失败，但是并没有给出提示失败的信息。

最后添加时调用轨道系统的方法即可。

3.10.2.5 删除一条轨道

功能五：删除一条轨道。

对于用户来说，要给定删除轨道的半径。给定的轨道半径参数的实际意义应该是第几条轨道的索引。但是，假设本来有 5 条轨道，增加一条半径为 7 的轨道后，虽然这条新轨道其实是在索引为 5 的位置上（最内层轨道索引为 0），但要想删除这条轨道，还是应该传入一条半径为 7 的轨道才行，然后调用轨道系统的方法即可。

3.10.2.6 删除特定轨道上的某个物体

功能六：删除特定轨道上的某个物体。

对于用户来说，删除特定轨道上的某个物体需要给定轨道半径。这里的轨道必须是一条已存在的轨道。如果这条轨道不存在，那么将删除失败，但是并没有给出提示失败的信息。

最后删除时调用轨道系统的方法即可。

3.10.2.7 计算信息熵

功能七：计算轨道系统的信息熵。

见 3.10.1.7.

3.10.2.8 模拟电子跃迁

功能八：模拟电子跃迁。

模拟电子跃迁没有一个专门的封装方法实现，只要给定跃迁的始轨道和末轨道，就可以调用增删电子的方法，来实现始轨道的一个电子，跃迁到末轨道上的一个效果。

根据 Memento 设计模式，电子跃迁之后是要存储一次当前状态的。

```
case 8: // Transition
    System.out.println("What's the radius(integer) of the source track?");
    int sourceRadius = in.nextInt();
    System.out.println("What's the radius(integer) of the target track?");
    int targetRadius = in.nextInt();
    atomStructure.deletePhysicalObjectFromTrack(electronFactory.produce(),
        trackFactory.produce((double) sourceRadius));
    atomStructure.addPhysicalObjectToTrack(electronFactory.produce(),
        trackFactory.produce((double) targetRadius));
    caretaker.addMemento(atomStructure.save());
    break;
```

3.10.2.9 撤销跃迁操作（恢复历史跃迁状态）

功能九：撤销若干次跃迁操作。

正如 3.9.6 中介绍的 Memento 设计模式，这中撤销操作只限于对跃迁的历史状态恢复，而不能恢复除跃迁操作之外的时间点。

3.10.3 SocialNetworkCircle

有关社交关系网络的代码在 SocialNetworkCircleAPP 中，该类中有一个 public static 的 application 函数，在 main 中被调用，执行应用代码，还有两个内部的菜单显示函数。大体上，社交关系网络的应用功能如菜单显示如下，共 11 个功能：

```
/**
 * Menu in social network circle application which indicates users' choices.
 */
private static void menu() {
    System.out.println("1. Read from file to generate a social network circle.");
    System.out.println("2. Visualize.");
    System.out.println("3. Add a track.");
    System.out.println("4. Add an object to a track.");
    System.out.println("5. Delete a track.");
    System.out.println("6. Delete an object in a track.");
    System.out.println("7. Calculate the information entropy of the system.");
    System.out.println("8. Calculate information diffusivity of someone in the first track.");
    System.out.println("9. Add/Delete a social relationship.");
    System.out.println("10. Calculate logical distance of two friends.");
    System.out.println("11. Check legality.");
}
```

3.10.3.1 读取文件

功能一：读取文件，并根据文件数据构造一个轨道系统。

由于我们有三种不同数据规模的文件，可以让用户选择读取哪一个，读取功能直接调用轨道系统的方法即可。

3.10.3.2 可视化

功能二：社交关系网络可视化。

直接调用 CircularOrbitHelper 类中的 visualize 方法，传入构造好的社交关系网络参数即可。

3.10.3.3 增加轨道

功能三：增加一条轨道。

对于用户来说，要给定增加轨道的半径。社交关系网络中的轨道的半径参数的实际意义，是该轨道上的物体，到中心物体的最短路径的逻辑距离。想要增加一条轨道的话，给定的轨道半径没有任何意义，但是这条轨道半径不应该是已存在的任何一条轨道半径，否则增加失败，并且不提供失败信息。

事实上，在检查合法性时，检查的是每条轨道上的物体到中心物体的逻辑距离是否和这条轨道的位置索引是否相同，而没有利用轨道的半径。在求逻辑距离功能时，也没有利用物体所在的轨道半径。所以，这里给定的增加的轨道半径是没有实际意义的。

3.10.3.4 向特定轨道上增加物体

功能四：向特定轨道上增加物体。

对于用户来说，向特定轨道上增加物体需要给定轨道半径，以及要增加物体的相关信息。这里的轨道必须是一条已存在的轨道。如果这条轨道不存在，或者这个物体已存在，那么都将添加失败，但是并没有给出提示失败的信息。

物体信息要求用户以严格的语法输入，代码中会对这句输入做和读取文件时相同的语法解析。对于轨道半径的输入是无关紧要的，见 3.10.3.3 的分析。

最后添加时调用轨道系统的方法即可。

3.10.3.5 删除一条轨道

功能五：删除一条轨道。

对于用户来说，要给定删除轨道的半径。这里给定轨道的半径必须和删除轨道的 `radius` 参数完全相同，否则删除失败，并且不给出失败信息。

3.10.3.6 删除特定轨道上的某个物体

功能六：删除特定轨道上的某个物体。

对于用户来说，删除特定轨道上的某个物体需要给定轨道半径，以及要删除物体的相关信息。这里的轨道必须是一条已存在的轨道。如果这条轨道不存在，或者这个物体不存在，那么都将删除失败，但是并没有给出提示失败的信息。

物体信息要求用户以严格的语法输入，代码中会对这句输入做和读取文件时相同的语法解析。这里给定轨道的半径必须和删除轨道的 `radius` 参数完全相同，否则删除失败，并且不给出失败信息。

最后删除时调用轨道系统的方法即可。

3.10.3.7 计算信息熵

功能七：计算轨道系统的信息熵。

见 3.10.1.7.

3.10.3.8 计算信息扩散度

功能八：计算中心物体（人）的信息扩散度。

信息扩散度的定义是，中心物体（人）经过与他/她有直接关系的某个朋友，把这个中介朋友认为是媒介，可以结识到其它朋友的个数，这里的其他朋友本应该与中心物体（人）没有关系。并且这里规定，只有中心物体（人）到媒介的亲密度不小于媒介到其它无关朋友的亲密度，才可以认为，这个无关朋友可以被中心物体（人）结识。

所以，用户的输入只是某一个与中心物体（人）有直接关系的朋友。然后遍历朋友列表，获取这个媒介 `intermediary`。

```

-----,
pattern = Pattern.compile(
    "Friend\\s*:=\\s*<" + LabelRegex + commaRegex + "(\\d*)" + commaRegex + "([M|F]{1})" + ">");
matcher = pattern.matcher(intermediaryData);
while (matcher.find()) {
    String name = matcher.group(1);
    int age = Integer.valueOf(matcher.group(2));
    char sex = matcher.group(3).charAt(0);
    intermediary = friendFactory.produce(name, age, sex);
}
for (PhysicalObject friend : socialNetworkCircle.getRelationBetweenCentralAndObject()[0].keySet()) {
    if (friend.equals(intermediary)) {
        existIntermediary = true;
        break;
    }
}
if (!existIntermediary) {
    System.out.println("ERROR! This intermediary doesn't exist!");
    break;
}

```

然后，在关系网中遍历，寻找符合条件的邻接边，每当有一个符合条件的边，就说明找到了一个符合条件的可结识的朋友。最后输出 diffusivity 即可。

```

intimacy = socialNetworkCircle.getRelationBetweenCentralAndObject()[0].get(intermediary);
for (Edge<PhysicalObject> edge : socialNetworkCircle.getRelationBetweenObjects()) {
    if (edge.getSource().equals(intermediary)
        && !socialNetworkCircle.getRelationBetweenCentralAndObject()[0].keySet()
            .contains(edge.getTarget())
        && edge.getWeight() <= intimacy) {
        diffusivity++;
    }
}
System.out.println("The diffusivity via " + intermediary.getName() + " is: " + diffusivity);
break;

```

3.10.3.9 增加/删除一条关系

功能九：增加或删除一条关系。

这里的关系比较复杂。首先要用户输入是增加（Y/y）还是删除（N/n）关系。其它输入被认为是非法的。

```

-----
System.out.println("Add(Y/y) or delete(N/n)?");
String c = in.nextLine();
System.out.println(c);
if (!c.equalsIgnoreCase("Y") && !c.equalsIgnoreCase("N")) {
    System.out.println("Wrong input!");
    break;
}

```

然后需要按照严格的语法，输入要增加/删除的关系。代码中对此进行解析，同时要检查这条输入关系的相关的两个朋友，必须是关系网络中已经存在的两个朋友。不接受输入一条连接关系网络中不存在的朋友的关系。


```

while (matcher.find()) {
    double infimacy = Double.valueOf(matcher.group(3));
    String name1 = matcher.group(1);
    String name2 = matcher.group(2);
    boolean exist1 = false, exist2 = false;
    PhysicalObject friend1 = null, friend2 = null;
    for (Track track : socialNetworkCircle.getObjectsInTrack().keySet()) {
        for (PhysicalObject friend : socialNetworkCircle.getObjectsInTrack().get(track)) {
            if (friend.getName().equals(name1)) {
                friend1 = friend;
                exist1 = true;
            }
            if (friend.getName().equals(name2)) {
                friend2 = friend;
                exist2 = true;
            }
        }
    }
    if (!exist1 || !exist2) {
        System.out.println("Both friends must have existed!");
    } else if (c.equalsIgnoreCase("Y")) {
        socialNetworkCircle.addRelationshipBetweenPhysicalAndPhysical(friend1, friend2, infimacy);
    } else {
        socialNetworkCircle.deleteRelationshipFromPhysicalToPhysical(friend1, friend2);
    }
}

```

如果输入 Y/y, 那么调用增加关系, 如果输入的是 N/n, 那么调用删除关系方法。

在增删关系之后, 有可能关系网络中的逻辑距离发生变化, 此时需要一次重构关系网络。先获取此时的关系网络的所有关系相关信息, 然后重置关系网络, 并调用重构方法。

```

List<PhysicalObject> friends = new ArrayList<PhysicalObject>();
for (PhysicalObject friend : socialNetworkCircle) {
    friends.add(friend);
}
List<Edge<PhysicalObject>> physicalEdges = socialNetworkCircle.getRelationBetweenObjects();
@SuppressWarnings("unchecked")
Map<PhysicalObject, Double>[] centralEdges = new Map[2];
centralEdges[0] = socialNetworkCircle.getRelationBetweenCentralAndObject()[0];
centralEdges[1] = socialNetworkCircle.getRelationBetweenCentralAndObject()[1];
socialNetworkCircle.resetObjectsAndTrack();
socialNetworkCircle.constructSocialNetworkCircle(friends, physicalEdges, centralEdges);
break;

```

3.10.3.10 计算两朋友之间的逻辑距离

功能十：计算两朋友之间的逻辑距离。

只要用户给定两个朋友的信息, 就可以在社交关系网络中获得这两个朋友的对象, 然后调用 API 中的 `getLogicalDistance` 方法即可。

3.10.3.11 检查轨道合法性

功能十二：检查轨道合法性。

事实上, 一部分轨道合法性已经在代码层面中 RI 检查 `checkRep` 时完成了一部分。按照要求, 再检查以下几点:

- 如果某个人出现在第 n 条轨道上, 那么他和中心点的人之间的最短路径是 n 。






- 如果某个人与中心点用户不连通，则不应出现在轨道系统中。

事实上，在检查功能中，只要检查第一点即可。因为在 API 中获取最短路径长度的方法中规定，如果两个人之间没有关系，那么两者之间的逻辑距离为-1。又因为在 API 中 `getLogicalDistance` 方法的参数是两个轨道物体对象，并没有提供某轨道物体到中心物体的逻辑距离计算。但是，在这个方法内部，存在一个伪中心物体，这个伪中心物体是一个轨道物体对象，并且具有较为独特的特征，他是一个姓名为 `center`，年龄为 1，性别为 M 的轨道物体。所以，如果 `getLogicalDistance` 方法传入的第二个参数是姓名为 `center`，年龄为 1，性别为 M 的轨道物体的话，那么得到的结果，就是从中心物体到达目标轨道物体（第三个参数）的逻辑距离了。

3.11 应对应用面临的新变化

3.11.1 StellarSystem

按照实验要求，行星轨道变成符合实际的椭圆形轨道（意即轨道的属性从“半径”变化成描述椭圆的多个参数）。因此，出现了一种新的轨道，行星椭圆轨道。因为之前我们在设计 `Track` 时是面向接口编程，很容易想到把这个新的行星椭圆轨道也应用到 `Track` 接口，所以，添加一个 `StellarTrack` 类作为行星椭圆轨道类，添加一个 `StellarTrackFactory` 作为这个椭圆轨道的工厂类。

```
>  NormalTrack.java
>  StellarTrack.java
>  StellarTrackFactory.java
>  Track.java
>  TrackFactory.java
```

因为之前也是对接口编程，所以，添加这个类之后，应该不会有静态检查错误。

在 `StellarTrack` 类中加入两个 `rep`，分别表示长半轴和短半轴：

```
private final double longRadius;
private final double shortRadius;
```

然后再加入这两个 `rep` 的 `observer` 方法。再重写这个类的 `equals` 和 `hashCode`，只有当两个 `StellarTrack` 对象的长短半轴都分别相等时，两条轨道才相同。

这样的话，接口里面应该有三个 `observer` 方法，一个是获取圆轨道的半径，另外两个是椭圆轨道的 `observer`。根据场景需要，在 `StellarTrack` 里面将 `getRadius` 断言为假，同样在 `NormalTrack` 里面，把 `getLongRadius` 和 `getShortRadius` 也断言为假。

这样，因为限制了两个具体类能用的 `observer` 方法，虽然这里仍没有静态类型检查的错误，但是在运行时，现在代码中所有的 `Track` 接口对象调用的都是 `getRadius`，而且也都是实例化的 `NormalTrack`。现在将和行星轨道系统有关的所有轨道生成都改为 `StellarTrackFactory` 生产的对象。比如在读文件时的轨道构造：

```

    this.addPhysicalObjectToTrack(
        planetFactory.produce(planetName, planetState, planetColor, planetRadius,
            revolutionSpeed, revolutionDirection, originDegree),
        stellarTrackFactory.produce(trackLongRadius, trackShortRadius));

```

在修改完所有的轨道工厂生产方式之后,再运行就会抛出 `AssertionError`,这是因为我们调用的方法还都是 `getRadius`,而这个方法在 `StellarTrack` 中被断言为假。为了迎合需求,将所有的半径都改为长半轴和短半轴,比如在可视化时,要画出椭圆轨道,就需要获取长半轴和短半轴,按照比例来规定绘图规则:

```

for (int i = 1; i <= tracks.size(); i++) {
    double scale = tracks.get(i - 1).getLongRadius() / tracks.get(i - 1).getShortRadius();
    int drawLongRadius = Integer.parseInt(df.format(radiusScale * i * scale));
    int drawShortRadius = radiusScale * i;
    g.drawOval(centralCoordinate - drawLongRadius, centralCoordinate - drawShortRadius,
        2 * drawLongRadius, 2 * drawShortRadius);
}

```

全部修改之后,还可以调整一下配置文件,每组数据添加一个参数。

```

Stellar ::= <Sun,6.96392e5,1.9885e30>
Planet ::= <Earth,Solid,Blue,6378.137,1.99e8,1.49e8,29.783,CW,0>
Planet ::= <Mercury,Solid,Dark,1378.137,1.99e7,1.49e7,69,CW,20.085>
Planet ::= <Saturn,Liquid,Red,2378,1.99e6,1.49e6,2.33e5,CCW,39.21>
Planet ::= <Jupiter,Gas,Blue,1637.007,2.8e8,2e8,30,CW,70>
Planet ::= <Mars,Solid,Red,637.137,1.19e11,9.99e10,1000.93,CCW,110>
Planet ::= <Neptune,Liquid,Yellow,6627.137,1.99e5,1.49e5,9293.05,CCW,359>
Planet ::= <Uranus,Gas,Blue,637.137,1.99e11,1.49e11,1e5,CW,359>
Planet ::= <Venus,Solid,Red,6378.137,1.99e20,1.49e20,203.24,CCW,181.23>

```

最后再修改一下测试文件中的变化就完成了。

3.11.2 AtomStructure

按照实验要求,原子核需要表达为多个质子和多个中子,即处于中心点的物体可以是多个物体构成的集合。

对此,要实现这个新需求,可以在原子核中加一个 `list` 的 `rep`,表示物体集合,然后用 `mutator` 方法来修改这个 `rep`;还可以在 `constructor` 里面直接构造这 `list`;还可以用修饰模式,原子核每多一个质子或中子,就用修饰器套一层原子核。为了简单,就加一个 `list` 的 `rep`,然后再设计一个 `mutator` 来操作这个 `list` 就好了。

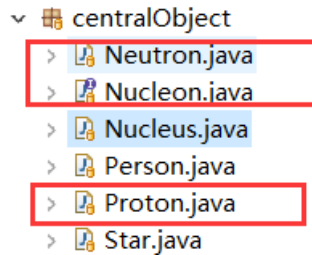
```

private final List<Nucleon> nucleons = new ArrayList<Nucleon>();

/**
 * Mutator. Add a nucleon into this nucleus.
 *
 * @param nucleon a Nucleon instance
 */
public void addNucleon(Nucleon nucleon) {
    this.nucleons.add(nucleon);
}

```

`Nucleon` 是一个表示核子的接口,有两个具体实现类,一个质子,一个是中子,并没有要求核子有什么属性或具体操作,所以就当作一个空 ADT。



3.11.3 SocialNetworkCircle

按照实验要求，为社交关系增加方向性。这一变化我在之前就已经考虑好并设计相关的 ADT 模式了。拿 ConcreteCircularOrbit 类的 rep 来说，

```
protected L centralObject;
protected List<Track> tracks = new ArrayList<Track>();
protected Map<Track, List<E>> objectsInTrack = new HashMap<Track, List<E>>();
@SuppressWarnings("unchecked")
protected Map<E, Double>[] relationBetweenCentralAndObject = new Map[2];
protected List<Edge<E>> relationBetweenObjects = new ArrayList<Edge<E>>();
```

relationBetweenCentralAndObject 被设计成一个长度为 2 的 Map 数组，其中维护着 2 个 Map。两个 Map 的 key 都是轨道物体，value 都是邻接边的权值，也就是亲密度。Map[0]表示从中心物体发出的邻接边，Map[1]表示指向中心物体的邻接边。

relationBetweenObjects 中是 Edge 的集合。Edge 是来自 Lab2 中的一个 ADT，是表示有向边的 ADT，有起始点，终止点和权值。

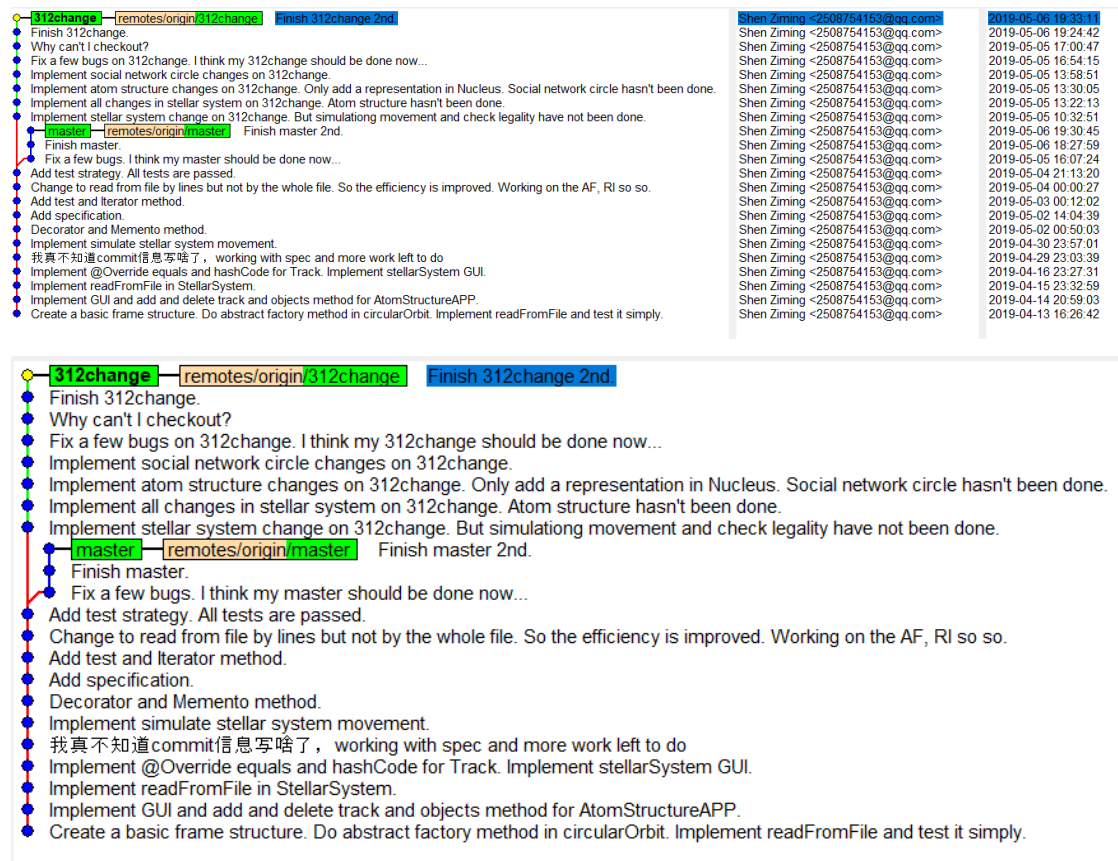
所以，这两个 rep 都可以适应有向关系的需求。只是之前调用关系相关的方法时，都是调用两次，双向捆绑，现在只需调用一次，表示单向关系。

```
@Override
public void addRelationshipBetweenPhysicalAndPhysical(E physicalObjectA, E physicalObjectB, d
    this.relationBetweenObjects.add(new Edge<E>(physicalObjectA, physicalObjectB, weight));
// this.relationBetweenObjects.add(new Edge<E>(physicalObjectB, physicalObjectA, weight));
    checkRep();
}
```

此外，在这个变化中，我的 spec 相比实验要求是增强的。我的新轨道系统支持从外到中心点的社交关系，也支持外层到内层的关系。

3.12 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。



在 `commit -m "Add test strategy. All tests are passed."` 之后，建立 `312change` 分支，开始 3.12 节的工作。3.12 节工作结束后，后来又在 `master` 上修改了 `bug` 和 `spec` 提交了 3 次，最终结束。

4 实验进度记录

日期	时间段	计划任务	实际完成情况

5 实验过程中遇到的困难与解决途径

遇到的难点	解决途径
equals 和 hashCode 的重写方法。equals 重写不难，主要是 hashCode	一位网友给出了很有总结性的答案： https://blog.csdn.net/zzg1229059735/article/details/51498310 原文在这里： https://www.mkymong.com/java/java-how-to-overrides-equals-and-hashcode/

<p>的重写，因为要满足自反性，对称性和传递性，而如果 equals 涉及到多个 rep 的相等关系的话，不知道 hashCode 怎么写。</p>	<p>我在本实验中采用的是 17-31 散列码和 java.util.Objects 来重写 hashCode 方法。</p>
<p>各种设计模式难以下手，简单工厂和抽象工厂混淆不清。</p>	<p>本次实验应用了多种设计模式，比较好做的算是 Memento 设计模式和 façade 设计模式，这两个照着课件写下来就好了。不太能分清的是简单工厂和抽象工厂。后来查了很多博客，发现这两个其实区别不大，抽象工厂算是对简单工厂的一次升级，抽象工厂常常可以和 build 模式在一起讨论，而这次实验中，针对三个具体的轨道系统，又没有太多的相关部件可以放到工厂中组装，所以看起来简单工厂和抽象工厂的区别有但不是很大。迭代器模式最开始没明白实验要求，后来在 piazza 上提问才明白，是要对轨道系统这个对象迭代，得到一个轨道物体的迭代结果。应用接口 Iterable 再重写 iterator 方法，设计一个私有的类继承 Iterator 类，再重写 next 和 hasNext 方法就好了。至于 decorator 模式也是比较不容易理解，因为之前都是把一个事物的细小属性放到 rep 里，现在是把这个属性提炼出来，反过来去修饰套在事物外面。这其中还要用到很关键的 delegation 关系。主要是设计出一个修饰类和一个修饰器类，修饰类继承修饰器类。修饰器类中有一个 rep 作为 delegation，构造器中构造这个 delegation 关系，而修饰类中才加入了我们以前加入到事物中的属性 rep。凡是和之前事物相同的地方，都用 delegation 直接继承，而有变化的地方，才使用新的属性 rep 操作。</p>
<p>一个 bug 就是当在用 Pattern 和 Matcher 时可能会发生不匹配，这有可能是字符串格式不正确或者正则表达式格式不正确导致的。也许你有的时候会像我一样试试，打</p>	<p>调用 matcher.find()会导致这个 boolean 量改变，它会自动检索有没有下一次匹配并更新它的值。</p>

<p>印一下 <code>matcher.find()</code> 是否为真，来确定到底有没有进入到 <code>while</code> 循环，那么如果你在调试完没有删除这一句，就会出现找了半天也找不到的 bug</p>	
<p>关于泛型的坑，很多时候，什么地方使用泛型，什么地方使用接口类型，什么地方使用具体类型，混淆不清。</p>	<pre>public static <L, PhysicalObject> void visualize(CircularOrbit<L, PhysicalObject> orbit, PhysicalObject obj)</pre> <p>这是一个错误示例。PhysicalObject 应该改为泛型 E。这是对静态方法的泛型声明，不应该用一个实际的类型声明。还有在参数和返回值都为泛型的方法中，如果要涉及到具体类型的操作，一定要做好参数变量的类型检查，并加以强制转换。</p>

6 实验过程中收获的经验、教训、感想

6.1 实验过程中收获的经验教训

设计模式真的是很强大的技能，做完实验三之后，感觉自己的项目目录非常清晰，类之间的关系也很清晰，就像办公桌抽屉里的文件夹一样。虽然还并没有体会到工厂模式避免了对象类型直接暴露给 client 的是什么意思，还没有体会到修饰器模式具体好用在哪，不过迭代器模式和外观模式还有备忘录模式是真的好用。

还有面向接口编程，这一点大大增强了可变化性，如果没面向接口编程，那么 3.12 节新的变化将很麻烦。有点体会到软件的接口就像硬件的主板上的插口一样了，需要什么东西，找来符合要求的那块板子插上去就好了。

语法驱动的正则表达式这个东西，练习一次之后就能记住了，上手难使用易，熟练之后就不会忘了。不过有没用过的地方还是得查。

还有 delegation 这个东西好多地方都用到了，现在的体会还不是很形象，其实之前的编程里可能也用到 delegation 了，只是当时还不知道叫这个玩意。方便是很方便的，就是还有点抽象。

重写的使用主要体现在面向接口编程，和具体 ADT 中的 `equals`, `hashCode`, `toString` 方法重写上。当然迭代器里面也涉及到了重写。重写给予开发者很大的权力，可以自己定义一些默认的方法操作。这样一来，和 Java 本身的方法之间就建立了联系，比如说 list 的 `contain` 操作，重写 `equals` 之后，就方便得多了。

再说复用代码，反正就是凡是需要复用的，都封装成 `public` 方法，和对象关系不大的，就写成 `static` 方法静态调用它，和对象关系大的，就仿照外观模式，实例化一个对象来调用它，总之条理很清晰。为了提高可读性，避免冗长的应用程序代码，可以把一些功能性强的代码片段封装成一个 `private` 函数，在内部调用它，可以提高可读性，使代码更整洁。

[本次实验的总结博客，总结真的没时间写了。。。。](#)

6.2 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？

A: 面对 ADT 编程的项目目录更加清晰，文件具有相应场景的实际意义，而直接面向应用场景编程只是写了几个没有灵魂的文件，而它们可以运行而已。复用虽好，但还是要注意不同场景下应用的差异，不然，一旦设计冲突了，那就麻烦大了。

- (2) 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？

A: 我赞同老师的观点。这些东西应该写，养成习惯，虽然我还是不太会写 `RI` 和 `AF`，不过 `safety from rep exposure` 和 `checkRep` 这两个东西还是非常有用的，开发责任交接时，有这些东西会更顺利一些。至于 `representation`，最重要的了。

- (3) 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？

A: 说实话，难是难倒也没有难的做不上来的地步，也谈不上什么乐趣。就是希望自己设计的 ADT 能适应更多的变化吧，也没指望谁能用这个轨道系统。

- (4) 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？

A: 设计模式真香。永远不要嫌类多，文件结构越清晰越好，尽可能地和实际应用场景相关联。

- (5) 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？

A: 起码以后对文本配置文件解析不愁了。之前觉得实验最难的部分之一就是解析实验给的文本配置文件了。另外我觉得这种对语法的复杂要求

仅仅是给开发者看的，要是给用户限制太多的语法要求，那就太头疼了。

- (6) 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？

A: 很难从一开始就确定非常清晰而且正确的思路一直走下去。先在纸上画画关系图，然后就拿一个最简单的应用场景一直做下去，主要还是得动手做，不能怕做错，也不能一股脑地全做完，一边做一边考虑其它场景，这样可以即时地调整 ADT 的设计。如果最后真的发现一个致命错误而不得不推倒重来，那就推倒重来罢……

- (7) 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？

A: 最开始看了，后来就没看了。没怎么考虑后面两个，只是按照之前的编程习惯，尽可能地提高健壮性和性能。有些地方对健壮性和性能要求的过于苛刻，就放下不去管他，等到后面的实验再说吧。

- (8) 关于本实验的工作量、难度、deadline。

A: 工作量巨大。难度较高。Deadline 合理，虽然合理，但还是受到了其他科目的冲击，能理解老师的良苦用心，但是现实它不给我们留情面啊~

- (9) 到目前为止你对《软件构造》课程的评价。

A: 听学长说这门课可能就是以后前途不济之时，不得不做一个小码农时的必备吃饭技能。别说了，我懂了。都是工程规范，我做就完事了:~)。