哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

2021 年春季学期
计算学部《软件构造》课程

Lab 3 实验报告

姓名	李昆泽
学号	1190201018
班号	1936603
电子邮件	3343033352@qq.com
手机号码	15720828552

目录

1 实验目标概述

本次实验覆盖课程第 2、 3 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
- 语法驱动的编程、 正则表达式
- API 设计、 API 复用

本次实验给定了三个具体应用（值班表管理、 操作系统进程调度管理、大学课表管理），学生不是直接针对每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

实验环境基本上与 Lab2 完全一致。

在这里给出你的 GitHub Lab3 仓库的 URL 地址（HIT-Lab3-学号）。
https://github.com/ComputerScienceHIT/HIT-Lab3-1190201018

3 实验过程

3.1 待开发的三个应用场景

三个应用分别是值班表管理系统、进程调度系统和课表管理系统。

值班表管理系统是要在一个给定的时间段内，对员工进行排班，在给定时间段内的每一天都必须有人值班，并且一位员工只能在一个时间段值班。

进程调度系统需要对进程进行调度。在任何时刻，最多只有一个进程正在运行，每个进程执行的时间也是不一样的。我们需要做的就是对进程进行调度，最终执行完所有进程。

课表管理系统针对的则是课表的管理。相比于前两个应用，课表管理系统最大的不同之处在于它是周期性的，需要以周为单位进行安排。

这三个应用有一定的相似性。它们都涉及到对时间段的安排，并且每个时间段都可以对应到一个唯一的“label”。值班表管理系统需要对排班的时间段进行安排，每个时间段对应一个员工；进程调度系统则要对 CPU 的整个时间轴进行安排，每个时间段对应一个进程或者不执行进程；课表管理系统则要把时间段与课程关联起来。

与此同时，这三个应用也有各自的独特之处。例如：值班表管理系统需要覆盖整个时间段，不能留有空白，而其他两个应用都是可以的；又比如值班表管理系统和进程调度系统都是不允许时间段的重叠的，但是课程管理系统则是运行。

3.2 面向可复用性和可维护性的设计：IntervalSet<L>

该节是本实验的核心部分。

3.2.1 IntervalSet<L>的共性操作

为接口 IntervalSet<L>设计提供下列共性接口方法：

- （静态方法）创建一个空对象： empty()
- 在当前对象中插入新的时间段和标签： void insert(long start, long end, L label)
- 获得当前对象中的标签集合： Set<L> labels()
- 从当前对象中移除某个标签所关联的时间段： boolean remove(L

label)

- 返回某个标签对应的时间段的开始时间： long start (L label)
- 返回某个标签对应的时间段的结束时间： long end (L label)
- 返回按照起始时间升序排列的时间段列表：List<Interval<L>> all()

3.2.2 局部共性特征的设计方案

首先创建一个 Interval<L>的类，用以保存每个单独的时间段，而 IntervalSet 实际上就是很多 interval 的集合。而在设计 Interval<L>这个类的时候，实际上是委派给了 MultiInterval<L>进行实现，这个在后续会提到。

CommonIntervalSet<L>是 IntervalSet<L>的实现类，在这个类中，我用 List<Interval<L>> intervals 和 Set<L> labels 分别管理 intervals 和每个 interval 对应的 label 集合。下面是具体方法的实现。

（1）empty()

这个与 lab2 中的静态工厂方法类似。

```
/**
 * Create an empty interval set.
 *
 * @param <L> type of labels in this interval set, must be immutable
 * @return a new empty interval set
 */
public static <L> IntervalSet<L> empty(){
    return new CommonIntervalSet<>();
}
```

（2）insert(long start,long end, L label)

首先判断 intervals 之前是否插入过当前标签的 interval，如果插入过，则返回 false，并且不进行任何操作；否则把标签（label）、起始时间（start）、终止时间（end）组合成一个新的 interval 并插入到当前的 interval set 中。

（3）labels()

直接返回 labels 即可。

（4）remove(L label)

首先在 intervals 中查找是否有标签为 label 的 interval，如果有，则删除相应的 interval，并且在 labels 中将对应的 label 删除，并且返回 true；否则不进行任何操作，并且返回 false。

（5）start(L label)

首先在 intervals 中查找是否有标签为 label 的 interval，如果有，则返回相应的 interval 的 start 属性；否则抛出异常。

（6）end(L label)

首先在 intervals 中查找是否有标签为 label 的 interval，如果有，则返回相应的 interval 的 end 属性；否则抛出异常。

（7）all()

这个方法实际上是为了后续的应用设计的。首先按照 Interval 的 start 的升序对 intervals 中的所有 interval 进行排序，并且返回排序好的 List。这里需要对 Java 库中的 Comparator 进行 override，具体实现如下。

```
intervals.sort(new Comparator<Interval<L>>() {
    @Override
    public int compare(Interval<L> o1, Interval<L> o2) {
        return (int) (o1.getStart()-o2.getStart());
    }
});
```

3.2.3 面向各应用的 IntervalSet 子类型设计（个性化特征的设计方案）

暂无面向各应用的IntervalSet子类型,后续子类型均为MultiIntervalSet的相关委派（Delegation）。

3.3 面向可复用性和可维护性的设计：MultiIntervalSet<L>

3.3.1 MultiIntervalSet<L>的共性操作

为接口 MultiIntervalSet<L>设计提供下列共性接口方法：

- （静态方法）创建一个空对象： empty()
- 创建一个空对象： empty()或不带任何参数的构造函数
- 创建一个非空对象：构造函数 MultiIntervalSet(IntervalSet<L> initial)，利用 initial 中包含的数据创建非空对象
- 在当前对象中插入新的时间段和标签： void insert(long start, long end, L label)
- 获得当前对象中的标签集合： Set<L> labels()
- 从当前对象中移除某个标签所关联的所有时间段： boolean remove(L label)
- 从当前对象中获取与某个标签所关联的所有时间段：IntervalSet<Integer> intervals(L label)，返回结果表达为IntervalSet<Integer>的形式，其中的时间段按开始时间从小到大的次序排列。

- 返回按照起始时间升序排列的时间段列表：List<Interval<L>> all()

3.3.2 局部共性特征的设计方案

首先创建一个 MultiInterval<L>的类，对于其中的每个 label，对应多个时间段。其中提供了 add，getStart，getEnd，getLabel 等方法，用于往 MultiInterval 中插入 interval 或者获取 MultiInterval 的相关属性。

CommonMultiIntervalSet<L>是 MultiIntervalSet<L>的实现类，在这个类中，我用 Set<MultiInterval<L>> multiIntervals 和 Set<L> labels 分别管理 MultiIntervals 和每个 MultiInterval 对应的 label 集合。下面是具体方法的实现。

（1）empty()

这个与 lab2 中的静态工厂方法类似。

（2）insert(long start,long end, L label)

首先判断 intervals 之前是否插入过当前标签的 interval，如果插入过，则找到相应的 MultiInterval，并且调用 add 方法进行插入；否则把新建一个标签为 label 的 MultiInterval，并把起始时间和终止时间分别为 start 和 end 的这个 interval 插入进去。

```
@Override
public void insert(long start, long end, L label) throws Exception {
    if(!labels.contains(label)) {
        MultiInterval<L> multiInterval = new MultiInterval<>(label);
        multiInterval.add(start, end);
        multiIntervals.add(multiInterval);
        labels.add(label);
        checkRep();
        return;
    }else {
        for(MultiInterval<L> multiInterval : multiIntervals) {
            if(multiInterval.getLabel().equals(label)) {
                multiInterval.add(start, end);
                checkRep();
                return;
            }
        }
    }
}
```

（3）labels()

直接返回 labels 即可。

（4）remove(L label)

首先在 intervals 中查找是否有标签为 label 的 interval，如果有，则删除相应的 MultiInterval，并且在 labels 中将对应的 label 删除，并且返回 true；否则不进行任何操作，并且返回 false。

（5）intervals(L label)

首先在 intervals 中查找是否有标签为 label 的 MultiInterval，如果有，

则把相应的 MultiInterval 转化为标签分别为 0,1,2……的 IntervalSet，并返回；否则，就返回一个空的 IntervalSet。

（6）all()

这个方法实际上是为了后续的应用设计的。把 MultiIntervalSet 里的所有时间段全部取出，首先按照 start 的升序对所有时间段进行排序，并且返回排序好的 List。这里需要对 Java 库中的 Comparator 进行 override，具体实现如下。

```
@Override
public List<Interval<L>> all()  {
    List<Interval<L>> temp = new ArrayList<>();
    Iterator<MultiInterval<L>> iter = multiIntervals.iterator();
    while(iter.hasNext()) {
        MultiInterval<L> multiInterval = iter.next();
        L label = multiInterval.getLabel();
        List<Long> start = multiInterval.getStart();
        List<Long> end = multiInterval.getEnd();
        for(int i=0; i<start.size(); i++) {
            try {
                temp.add(new Interval<L>(start.get(i), end.get(i), label));
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    temp.sort(new Comparator<Interval<L>>() {
        @Override
        public int compare(Interval<L> o1, Interval<L> o2) {
            return (int) (o1.getStart()-o2.getStart());
        }
    });
    checkRep();
    return Collections.unmodifiableList(temp);
}
```

3.3.3 面向各应用的 MultiIntervalSet 子类型设计（个性化特征的设计方案）

（1）NoBlankIntervalSet

这个是针对不允许出现空白的 MultiIntervalSet 子类型设计。NoBlankIntervalSet 和 MultiIntervalSet 并不是继承关系，而是委派关系。

NoBlankIntervalSet 类中一共有三个方法：checkNoBlank、insert 和 remove。

关于 checkNoBlank 的实现思路是：首先把每个时间段映射到一个集合上（例如(0,5)就映射 0,1,2,3,4,5），然后在调用 checkNoBlank 时，获取集合中的最大和最小值，如果集合是没有 blanks 的，那么集合中应该会包含最大与最小间的所有数。我们利用这个规律来检查当前的 MultiIntervalSet 是否是 NoBlank 的。需要注意的是 checkNoBlank 只需要在所有操作均完成后调用，没有必要在 insert 或者 remove 方法中调用。

关于 insert 和 remove 的实现较为简单，基本上就是对 MultiIntervalSet 中相应方法的一个委派。

（2）NonOverlapIntervalSet

这个是针对不允许出现重叠的 MultiIntervalSet 子类型设计。与上个子类型一样，NonOverlapIntervalSet 和 MultiIntervalSet 也不是继承关系，而是委派关系。

NonOverlapIntervalSet 类中一共有四个方法：checkNoOverlap、insert、remove 和 all。

关于 checkNoOverlap，要分两种情况，一种是不允许相同标签的时间段重叠，但允许不同标签的时间段重叠；另一种是不允许所有时间段重叠。针对第一种情况，首先对每个标签利用 MultiIntervalSet 中的 Intervals 方法，取出某个标签对应的全部时间段，再利用 all 方法对它们进行排序，通过比较相邻两个时间段的起始时间和终止时间的相对大小关系判断是否有时间段重叠。如果是第一种情况，到这里应该就结束了。而如果是第二种情况，则直接对 MultiIntervalSet 调用 all 方法，然后通过比较相邻两个时间段的起始时间和终止时间的相对大小关系判断是否有时间段重叠。

关于 insert、remove 和 all 的实现较为简单，基本上就是对 MultiIntervalSet 中相应方法的一个委派。

（3）PeriodIntervalSet

这个接口的设计思路其实和前面两个不太一样，我发现更重要的是每个时间段安排了什么事件，而不是某个事件被安排在了哪些时间段上。而且这些时间段往往是固定的，我们甚至可以把它们抽象成时间点，对应每个点保存相应事件的集合。

所以我使用了一个 List<Set<L>> records 用来保存每个时间点对应的事件集合。除此之外，period 表示周期长度，times 表示每天的时间段数，这看起来是个二维数组，但是我们可以把它映射到一个一维数组上（针对第 i 天，第 j 个时间段，index = (i-1)*times + j）。这样，records 中的每个元素表示的都是一个周期上的某个时间段对应的事件集合。

而在 insert 的时候，用 interval 的标签表示 i,start=end=j，这样传入一个 interval 类型的参数，在此基础上传递真实的 label（因为这里的 interval 的 label 不再表示真实的 label），然后在函数里把这个事件加入 records。

```
@Override
public void insert(L label, Interval<Integer> interval) throws Exception {
    int i = interval.getLabel();
    int index = (i-1)*times+(int)interval.getStart();
    records.get(index).add(label);
}
```

有了这个 records，我们其实就对整个 PeriodIntervalSet 了如指掌了，因为我们知道任意时间段发生的事件。我还加入了 ratioOfFree 和 ratioOfOverlap 方法，我们只需要对 records 中 size==0 和 size>1 的元素个数进行统计即可。

```java
@Override
public double ratioOfFree() {
    int freeNum = 0;
    for(int i=1; i<records.size(); i++) {
        if(records.get(i).isEmpty()) {
            freeNum++;
        }
    }
    return (double)freeNum/(records.size()-1);
}

@Override
public double ratioOfOverlap() {
    int overlapNum = 0;
    for(int i=1; i<records.size(); i++) {
        if(records.get(i).size() > 1) {
            overlapNum++;
        }
    }
    return (double)overlapNum/(records.size()-1);
}
```

3.4 面向复用的设计：L

对于所有的接口，我们都保留了 L 的设计，L 可以被任何的 immutable 类型代替，这就增加了代码的灵活性。当需要指定类型时，也可以直接指定。

3.5 可复用 API 设计

3.5.1 计算相似度

double Similarity(MultiIntervalSet<L> s1, MultiIntervalSet<L> s2)

这里只要知道两个数据即可。一是两个 MultiIntervalSet 的总的时间长度大小(size)，二是两个 MultiIntervalSet 中相同部分的时间总长度(fit)。

同样，我们还是利用 MultiIntervalSet 的 all 方法，首先取出 s1 和 s2 中的所有时间段，找到其中 start 最小值和 end 最大值，二者作差即为总的时间长度大小。

然后，对于 s1 中的每个 label，我们首先取出某个 label 对应的所有的时间段，将经历的时间保存下来，再去 s2 中找 label 对应的所有时间段，二者进行比较，记录重合的时间数。最后把每个 label 的重合时间进行累加，就可以得到 s1 和 s2 中相同部分的时间总长度。最后返回(double)fit/size。

3.5.2 计算时间冲突比例

double calcConflictRatio(IntervalSet<L> set)

double calcConflictRatio(MultiIntervalSet<L> set)

第一个方法是第二个方法的特例，所以只需要完成第二个方法的实现即可。

与之前的思路类似，首先计算时间总长度 size。

```java
long size = set.all().get(set.all().size()-1).getEnd() - set.all().get(0).getStart();
```

然后创建一个负责计数的数组，只要某个时间段中包含了某个时间，该时间的计数就加 1，最后统计数量超过 1 的时间长度 fit。

最后返回(double)fit/size。

3.5.3 计算空闲时间比例

double calcFreeTimeRatio(IntervalSet<L> set)

double calcFreeTimeRatio(MultiIntervalSet<L> set)

第一个方法是第二个方法的特例，所以只需要完成第二个方法的实现即可。

与之前的思路类似，首先计算时间总长度 size。

```
long size = set.all().get(set.all().size()-1).getEnd() - set.all().get(0).getStart();
```

然后创建一个负责计数的数组，只要某个时间段中包含了某个时间，该时间的计数就加 1，最后统计数量仍未 0 的时间长度 fit。

最后返回(double)fit/size。

3.6 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。

3.6.1 排班管理系统

首先是排班起始日期、终止日期的确定以及员工信息的录入。这里采用文件读入、正则表达式解析的方案。

通过下列函数把文件转化为字符串。

```
private static String fileToString(String filename) throws IOException {
    File file = new File(filename);
    if(!file.exists()){
        return null;
    }
    FileInputStream inputStream = new FileInputStream(file);
    int length = inputStream.available();
    byte bytes[] = new byte[length];
    inputStream.read(bytes);
    inputStream.close();
    String str = new String(bytes, StandardCharsets.UTF_8);
    return str ;
}
```

然后针对 Period 和 Employee 的不同格式构造正则表达式对字符串进行解析。解析之后，把相应的起始日期、终止日期信息以及员工信息提取并保存下来。

如果使用手动排班，那么排班的信息也可以通过文件+正则表达式的方式进行录入，而“排班”这一操作，其实就是把 Employee 对象、起始时间、终止时间当作 interval 插入 MultiIntervalSet 的过程，直接利用 MultiIntervalSet 的子类型 NonOverlapIntervalSet 可以实现。

而随机排班的思想是按照时间顺序，从起始日期排起，一直到终止日期。每

次从员工表里随机取出一名员工（可以重复，其实解决了 change 的变化 1），然后从当前的起始日期一直排到终止日期前的某一天（随机生成），然后更新起始日期，直至全部排完为止。

关于排班表的可视化，其实主要是利用 NonOverlapIntervalSet 的 all 方法。我们需要获取排班表 TimeTable 中的所有时间段，然后依照时间顺序，依次输出每个时间段对应的 Employee。

除此之外，还需要在任意时刻检查是否排满。我们依然可以利用 NonOverlapIntervalSet 的 all 方法，但这次是反向获取空闲的时间段。如果空闲时间段为空，那么则已排满；否则就输出空闲的时间段。

但是因为最后要设计一个客户端程序，我们一方面要让客户端尽可能简单，但是也要实现应用所有的功能。我的想法是构建一个 process 方法，把客户端需要的菜单、选择都在这个函数里实现，根据用户的选择，进行相应的操作。具体的 process 方法如下。

```
public void process() throws ParseException, Exception {
    int choice = meau();
    while(choice != 0) {
        switch(choice) {
        case 1:
            System.out.println("请输入员工姓名：");
            String name = s.next();
            while(!hasEmployee(name)) {
                System.out.println("请输入员工姓名：");
                name = s.next();
            }
            System.out.println("请输入值班开始时间：");
            String startTime = s.next();
            System.out.println("请输入值班结束时间：");
            String endTime = s.next();
            set(name, startTime, endTime);
            choice = meau();
            break;
        case 2:
            TimeTable = new NonOverlapIntervalSetImpl<>();
            List<Employee> temp = new ArrayList<>();
            Iterator<Employee> iter = EmployeeTable.iterator();
            while(iter.hasNext()) {
                Employee employee = iter.next();
                temp.add(employee);
            }
            long start = 0;
            while(start < period) {
                long end;
                if(temp.size() == 1) {
                    end = period;
                }else {
                    end = (long) (Math.random() * (period - start + 1) + start);
```

```
            }
            Employee employee_t = temp.get(0);
            temp.remove(0);
            Calendar c = Calendar.getInstance();
            c.setTime(startDay);
            c.add(Calendar.DATE, (int) start);
            String startDate = sdf.format(c.getTime());
            c.setTime(startDay);
            c.add(Calendar.DATE, (int) end);
            String endDate = sdf.format(c.getTime());
            set(employee_t.getName(), startDate, endDate);
            start = Math.min(end+1, period);
            if(end+1 == period) {
                c.setTime(startDay);
                c.add(Calendar.DATE, (int) period);
                set(temp.get(0).getName(), sdf.format(c.getTime()), sdf.format(c.getTime()));
            }
        }
        choice = meau();
        break;
    case 3:
        checkFull();
        choice = meau();
        break;
    case 4:
        System.out.println("当前排班情况：");
        System.out.println(toString());
        choice = meau();
        break;
    default:
        choice = meau();
        break;
    }
```

最后的客户端程序极为简单。

```
public class DutyRosterApp {

    public static void main(String[] args) throws Exception {

        DutyIntervalSet DutyRoster = new DutyIntervalSet("./txt/duty/test.txt");

        DutyRoster.process();

    }
}
```

3.6.2 操作系统的进程调度管理系统

首先是进程信息的录入。这里依然采用文件读入、正则表达式解析的方案。

然后针对 Process 的不同格式构造正则表达式对字符串进行解析。解析之后，把相应的进程信息提取并保存下来（保存在 processes 里）。

如果使用随机选择进程，其实选择进程的过程就是把 process 以及它执行的时间段插入到 NonOverlapIntervalSet<String> ProcessTable 的过程，这里是把 insert 操作委派给了 NonOverlapIntervalSet。这里对随机数的大小有一定要求：随机数必须小于等于该进程的最长执行时间，如果大于等于最短执行时间，则认为该进程执行完毕。我们每次选择一个进程（或者不选择，可以通过在选择进程时增加一个随机数的方法实现）进行执行，直至所有进程均执行完毕。

而最短进程优先的思想则是优先执行距离其最大执行时间差距最小的进程，这里就不用随机选择进程了，但是每次执行的时间依然是不确定的。

关于排班表的可视化，其实主要是利用 NonOverlapIntervalSet 的 all 方法。我们需要获取排班表 ProcessTable 中的所有时间段，然后依照时间顺序，依次输出每个时间段对应的进程 ID。

但是因为最后要设计一个客户端程序，我们一方面要让客户端尽可能简单，但是也要实现应用所有的功能。我的想法是构建一个 process 方法，把客户端需要的菜单、选择都在这个函数里实现，根据用户的选择，进行相应的操作。具体的 process 方法如下。

```
public void process() throws Exception {
    int choice = meau();
    while(choice != 0) {
        switch(choice) {
        case 1:
            ProcessTable = new NonOverlapIntervalSetImpl<>();
            randomSet();
            choice = meau();
            break;
        case 2:
            ProcessTable = new NonOverlapIntervalSetImpl<>();
            bestSet();
            choice = meau();
            break;
        case 3:
            System.out.println("请输入当前时刻数（自然数）：");
            int time = s.nextInt();
            show(time);
            choice = meau();
            break;
        default:
            choice = meau();
            break;
        }
    }
    System.out.println("已退出！");
    s.close();
    checkRep();
}
```

最后的客户端程序。

```
public class ProcessScheduleApp {

    public static void main(String[] args) throws Exception {

        ProcessIntervalSet ProcessSchedule = new ProcessIntervalSet("./txt/process/test.txt");

        ProcessSchedule.process();

    }
}
```

3.6.3 课表管理系统

首先是课程信息的录入。这里依然采用文件读入、正则表达式解析的方案。

与前两个应用相比，这个 ADT 里反而没有太多功能实现的方法，这是因为在 PeriodIntervalSet<L>这个接口里，已经基本实现了这里所需要的所有方法。

我们依然要设计一个客户端程序，为了让客户端尽可能简单，我们把所有的功能选择放在 process 方法里。针对每个选择，把具体的实现委派给 PeriodIntervalSet 里的方法。

3.7 基于语法的数据读入

在上一节中已经使用基于语法的数据读入，即将文件转化为字符串之后，利用正则表达式对字符串进行解析，以获取想要的数据。

例如，利用下图所示的代码即可完成对“yyyy-MM-dd”格式日期的匹配。

```
String pattern_t = "\\d{4}-\\d{2}-\\d{2}";
Pattern r_t = Pattern.compile(pattern_t);
Matcher m_t = r_t.matcher(temp);
```

3.8 应对面临的新变化

3.8.1 变化 1

由于之前在排班管理系统里使用的是 MultiIntervalSet 进行排班表的安排，其实隐含着允许了变化 1 的发生。所以其实这里不用做任何修改。

3.8.2 变化 2

这里应对变化依然比较容易。

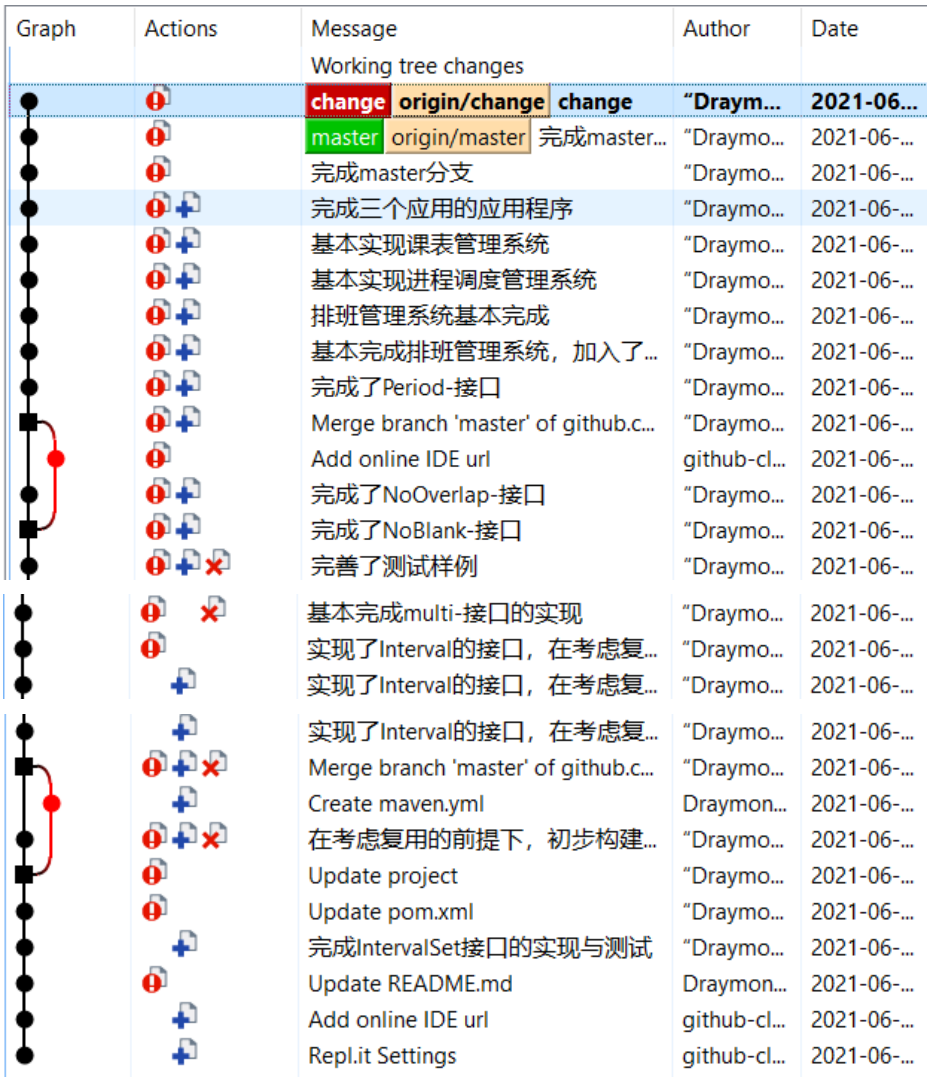
之前的 PeriodIntervalSet 接口里的 insert 方法是允许时间段重复的，如果不允许重复，只需要再增加一个不允许重复的 insert 方法即可。我们再回到 PeriodIntervalSet 的实现，类里面是用一个 List<Set<L>>类型的 records 记录各个时间段的安排的。如果要确保时间段的不重叠，只要每次试图往 records 里添加元素之前检查一下这个位置之前是否已经有元素存在即可。相关方法如下。

```
@Override
public boolean nonOverlapInsert(L label, Interval<Integer> interval) throws Exception {
    int i = interval.getLabel();
    int index = (i-1)*times+(int)interval.getStart();
    if(records.get(index).isEmpty()) {
        records.get(index).add(label);
        return true;
    }
    return false;
}
```

该方法是有返回值的，如果用户输入的安排时间与之前的时间段冲突，那么则插入无效，否则则是有效插入。

3.9 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 `change` 分支和 `master` 分支所指向的位置。



4 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

2021-06-08	13:45-16:30	完成 IntervalSet 接口的实现与测试	按计划完成
2021-06-09	18:30-23:10	完成 MultiIntervalSet 接口的实现与测试	在考虑复用的前提下，初步构建了 multi-类的框架
2021-06-10	9:00-10:00	完成 MultiIntervalSet 接口的实现与测试	初步实现 Multi-接口的功能
2021-06-12	18:30-21:00	完成 MultiIntervalSet 接口的实现与测试	按计划完成
2021-06-15	14:00-16:00	完成 NoBlank-接口	按计划完成
2021-06-22	14:00-15:00	完成 NoOverlap-接口	按计划完成
2021-06-25	13:30-17:50	完成 Period-接口	按计划完成
2021-06-27	18:00-23:30	完成排班管理系统 ADT	在按计划完成的基础上，加入了正则表达式的解析
2021-06-28	14:00-16:00	完成进程调度管理系统 ADT	按计划完成
2021-06-29	15:00-19:40	完成课表管理系统 ADT	按计划完成
2021-06-29	20:00-23:50	完成三个应用程序的客户端	按计划完成
2021-06-30	12:00-13:30	完善注释，完成 master 分支的所有功能	按计划完成
2021-06-30	13:30-14:00	完成 change 分支	按计划完成

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
正则表达式的解析与匹配	通过上网上查找 java 正则表达式的资料，学会了正则表达

	式解析字符串的一般流程以及相关语法。
客户端设计不够简洁	最开始我想在让客户在客户端界面调用 ADT 内的各类方法，但是最后发现这样做客户端有点过于复杂了。解决的办法是把这一整套过程放在 ADT 内部的一个方法里（我用的是 process）。
周期性 MultiIntervalSet 子类型设计	在 PeriodIntervalSet 这个接口里，如果一味想使用现有的 MultiIntervalSet 接口是比较复杂的，也不太好实现，所以我干脆把时间段当作下标，保存对应的事件。
关于日期的转化	主要利用 Date 类，还有 SimpleDateFormat 类进行日期的快速转化以及格式化输出。在有些地方，还使用了 Calendar 类进行计算，比如计算某个日期后第 n 天的日期。

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和教训

通过这次实验，我感觉我既体会到了设计复用的复杂性，也体会到了如果设计得当给后续编程带来的便利性。但是，如果一开始不去考虑复用，不去考虑代码的可拓展性，可能依然可以实现功能，但在后续需要修改时则需要修改大量代码，效率不高。

考虑复用的开发其实也是一个“折中”的过程，如果我们认为这部分代码日后不会有太大变化，那么复用可能不是那么必要；但如果现有功能拓展性很强，那我们在编程时就要事先预见到这种情况，不要吝啬增加一些接口、(抽象)类，尽管现在看很多类都没有太大的必要，但在日后拓展的时候会节省大量的工作量。

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？

直接面向应用场景编程的难度要远大于面向 ADT 的编程。如果别人已经规定好了 ADT 的 spec 的话，那就只需要根据 spec 进行实现就好了。但是如果是面向应用场景编程，我们需要设计 ADT，设计 spec，我们需要考虑各个 ADT 间的关系，让他们尽可能高效地运作。

如果 ADT 设计得当，充分考虑复用的话，那么当它被移植到不同场景时，会非常容易，只需要做少量的修改。

(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？

确保当前 ADT 的正确性、表示独立性等等，并且保证在调用过程中不会给 ADT 造成任何影响。这样在今后调用 ADT 时会非常方便，并且 ADT 本身不会出任何问题，如果出现 bug，可以直接排除 ADT 的问题。

我愿意在以后的编程中坚持这么做。

(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？

使用别人的 API 时，我会发现有些我需要的功能是没有的。但是当自己开发 API 时，也同样很难把所有的需求都考虑到。当然，在编程中调用自己实现的 API 也是一件很开心的事情。

(4) 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？

这样实际上是非常方便的，并且十分便于修改。如果语法规则变了，只需要修改正则表达式以及相关的解析语句即可。而且用正则表达式去表示一个语法是准确的。

(5) Lab1 和 Lab2 的工作都不是从 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？

设计 ADT 的难度主要体现在确定各个类、各个接口之间的关系，以及每个类实现功能的区分。通过老师上课讲解的一些基本设计原则，以及在编程、设计中摸索出的一些经验来克服。

(6) “抽象”是计算机科学的核心概念之一，也是 ADT 和 OOP 的精髓所在。本实验的三个应用既不能完全抽象为同一个 ADT，也不是完全个性化，如何利用“接口、抽象类、类”三层体系以及接口的组合、类的继承、委派、设计模式等技术完成最大程度的抽象和复用，你有什么经验教训？

尽可能地找到不同应用之间的相同之处，构建一个较为通用的接口。而对于一些个性化的功能，可以在这个接口的基础上实现一些子类型。然后在客户端把具体的功能实现委派给这些子类型。

(7) 关于本实验的工作量、难度、deadline。

工作量还是挺大的，难度也是不小。尤其因为快到期末了，没有太多时间进行大量的编程、设计，不然这次的实验应该会更有意思一些。

(8) 下周就要进行考试了，你对《软件构造》课程总体评价如何？

其实总的感觉还是课时不太够吧，不然多编一些程序应该会对软件的构造有一个更深的认识，也会更有意思一些。