第十四讲：
基于规格的实现正确性论证

OO2018课程组

计算机学院

本讲提纲

- 软件正确性内涵分析
- 类实现的正确性论证
- 方法实现的正确性论证
- 子类实现的正确性论证
- 实现正确性论证模板
- 作业

软件正确性内涵分析

- 正确性是软件最重要的一个特性
  - 软件正确实现用户需求
- 用户需求描述了用户对软件的期望
  - 功能期望
  - 数据期望
  - *性能期望*
- 开发人员根据用户需求进行分析，得到需求规格
  - 软件输入、输出数据定义（用户所关心的）
  - 输入输出约束定义
    - 值域约束
    - 时域约束
  - 输入输出机制
  - 软件对用户输入的处理流程

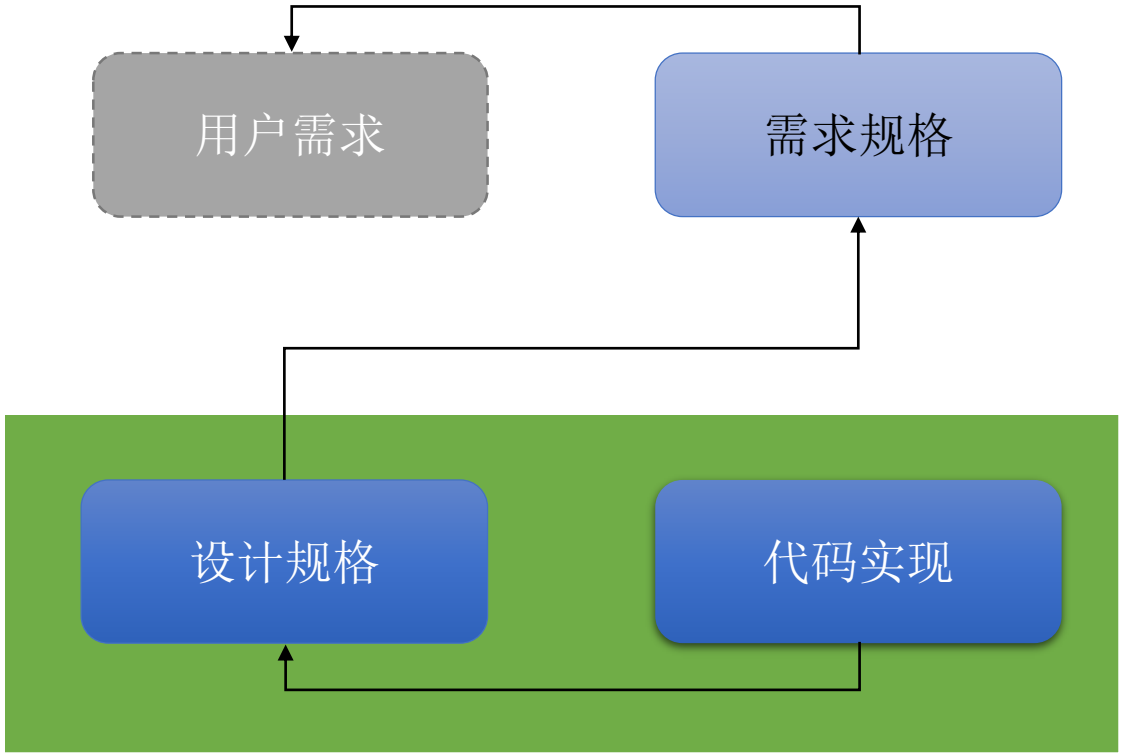
软件正确性内涵分析

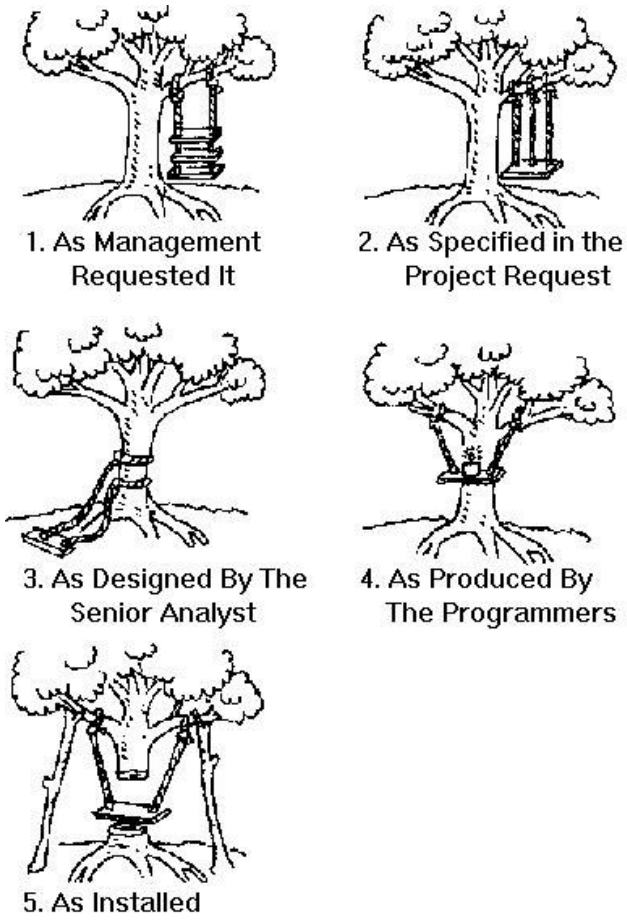
- 用户需求没有正确性问题
  - 可理解性问题
  - 实现成本问题
- 需求规格的主要问题
  - 是否准确捕捉了用户需求中的功能和数据
  - 是否准确识别了用户需求中隐含的环境/数据约束与限制
  - 是否准确整理了软件与用户的交互处理流程
  - 是否准确定义了软件运行环境及其约束
- 如果需求规格上述问题都能解决
  - 软件正确性可以定义为软件实现满足需求规格的特性

软件正确性内涵分析

- 软件设计的依据是需求规格
  - 用户需求：使用用户语言撰写的需求
  - 需求规格：使用技术语言撰写的需求
- 软件设计任务
  - 规划软件的模块（接口）及其依赖关系
    - 子系统
    - 组件
    - 类
  - 针对模块及其接口设计其规格
- 如果设计能够满足需求规格
  - 软件正确性可以定义为软件实现满足设计规格的特性

软件正确性内涵分析





逻辑追踪链(traceability link)是确保软件质量的关键

类实现的正确性论证

- 类的实现组成
  - 方法
  - 表示对象
- 类的规格组成
  - 抽象对象说明(overview)
  - 方法规格
  - *不变式*
- 正确性论证思路
  - 抽象对象都得到了有效实现
  - 方法实现都满足各自规格
  - 所有方法都不会导致不变式为假

类实现的正确性论证

- 抽象对象得到了有效实现
  - 为什么强调有效？
  - 使用抽象函数
    - 是不是抽象函数结果覆盖了overview中描述的抽象对象？
  - 有效性论证
    - 数据类型
      - 表示对象类型是否能够表示抽象对象的所有可能取值？
    - 数据存储规模
      - 表示对象是否能够存储抽象对象所有可能的数据量？

类实现的正确性论证

```
public class BinaryTree{
/*@overview: BinaryTree is a abstract tree to manage any type of elements. Each of its node can have at most one left node, and at most right node. It would be called a leaf node if it has neither left node nor right node.*/
//rep:
    private BinaryTree left;
    private BinaryTree right;
    private int numNodes;
}
```

```
public class DarkRoom{
/*@overview: DarkRoom manages all the communication tools defined in the configuration file. There are three types of communication tool: Sender, Receiver and Transmitter.*/
//rep:
    private Sender[] senders ;
    private Receive[] receivers
    private Transmitter[] transmitters;
    int numTools;
}
```

```
public class Sender{
/*@overview:…*/
}
```

```
public class Receiver{
/*@overview:…*/
}
```

```
public class Transmitter{
/*@overview:…*/
}
```

类实现的正确性论证

- 所有方法都不会导致不变式为假
  - 前提：确保所有rep都得到了私有保护
  - 查询方法
    - 确保不会对任何rep进行修改
    - 无需进行论证
  - 生成方法
    - 确保不会对任何rep进行修改
    - 确保使用构造方法来构造相应的生成对象
    - 无需进行论证
  - 变更方法
  - 构造方法

类实现的正确性论证

- 变更方法
  - 检查所有return点
    - 论证对rep的修改不会导致不变式为假（即repOK为假）
  - 检查所有return出去的变量
    - 论证不会对外暴露rep变量
- 构造方法
  - 检查所有执行结束点
    - 论证rep的初始化结果不会导致不变式为假
  - 检查所有构造输入参数
    - 论证不会直接共享传入的对象

```
public class IntSet {
/*@Overview: IntSets are unbounded, mutable sets of integers
@invariant: c.els <> null && all elements of
c.els[j].intValue*/
    private Vector els;  //the rep
public IntSet ( ) {
 /*@effects: \this.size==0*/
   els= new Vector( ); }
public void insert (int x) {
 /*@modifies: this
    @effects: \this.isIn(x)==tr
\this.isIn(\old(\this)[i])*/
   Integer y =new Integer(x) ;
   if(getindex(y) < 0) els . add(y); }
public void remove (int x) {
 /*@modifies: this
    @effects:\all int i;0<=i<\o
   int i= getindex(new Integer(x));
   if(i < 0) return;
   els.set(i, els.lastElement( ));
   els.remove(els.size() - 1); }
public int size ( ) {
   /*@effects: \result == \this.size */
   return els.size(); }
}
```

- 表示对象rep为Vector els，通过抽象函数可以映射为规模不限的整数集
- 构造方法：构造空的向量els，满足不变式
- insert：首先查询els中是否有x，如果没有则执行插入。不论els中是否有x，执行的结果都是els中有x，且原来集合中的元素仍然还在集合中，因此满足规格要求和不变式要要求。
- remove：首先查询els中是否有x，如果没有直接返回；如果有，且位置不是els中最后一个元素，则把最后一个元素复制到x位置以删除x，同时把最后一个元素删除保证最后一个元素不会重复出现；如果有且位置是els中最后一个元素，则把最后一个元素复制到x位置不对els做出任何改变，删除最后一个元素的结果则是删除x。不论如何，其执行结果都是els中没有了x，且凡是不等x的元素都仍然在集合中，因此满足规格和不变式要求。
- size：返回els的规模，已知构造方法、insert和remove都能确保不变式成立，即els中不会有重复元素，因此els的规模就是集合元素的个数。

方法实现的正确性论证

- 方法的规格组成
- 方法实现的正确性内涵
- 验证模板

方法实现的正确性论证

- 方法的规格组成
  - 标题声明
  - 前置条件
  - 副作用
  - 后置条件
- 正确性内涵
  - 前提：保证规格有效
    - 标题声明的异常必须在Effects中完整体现
    - 前置条件、后置条件可判定
    - 副作用范围确定
  - 任意满足前置条件的输入，方法执行的结果都能满足后置条件，同时不会修改副作用之外的非局部变量

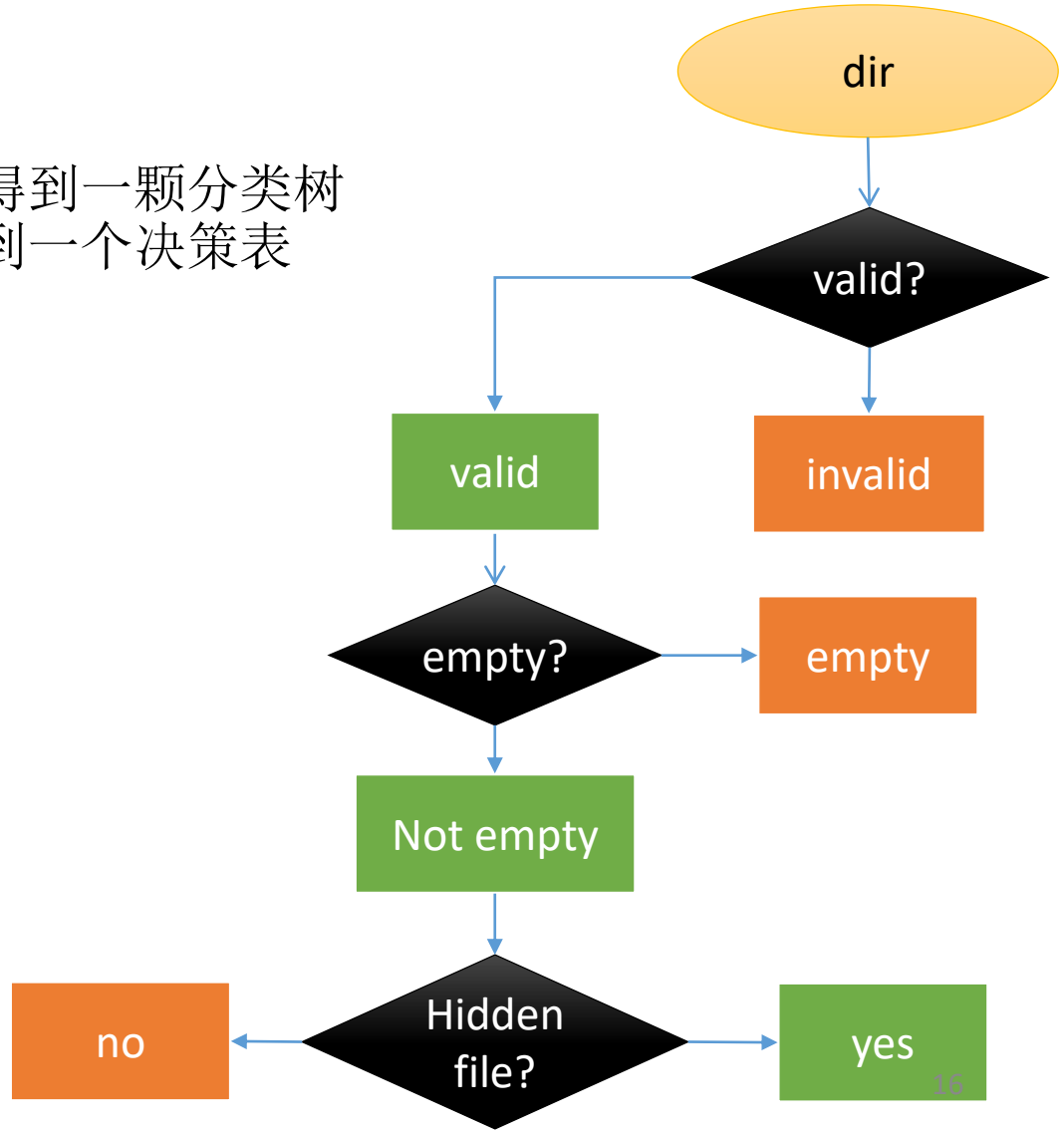
方法规格是设计的产物

方法实现的正确性论证

- 采用论证的方法(argument)，而不是形式化验证(formal verification)和执行测试(execution testing)
  - 测试：基于前置条件和后置条件对输入划分并抽样产生测试数据，检查方法执行输出是否满足后置条件
    - 优点：易于实施，工程广为采用
    - 缺点：无法确保正确性
  - 形式化验证：针对实现和规格构造形式化模型，通过模型检查或定理证明确认是否在所有可能的输入下，方法实现（即形式化模型）都能够给出满足后置条件的结果
    - 优点：严格验证
    - 缺点：应用成本高，形式化模型本身的正确性难以保证
  - 论证：针对格式化的规格和代码实现，人工方式对代码逻辑进行分析，确认是否所有满足前置条件的输入都能产生满足后置条件的结果
    - 优点：折衷，形式验证与自然语言层次逻辑推理相结合
    - 缺点：无法确保自然语言层次逻辑推理的严谨性

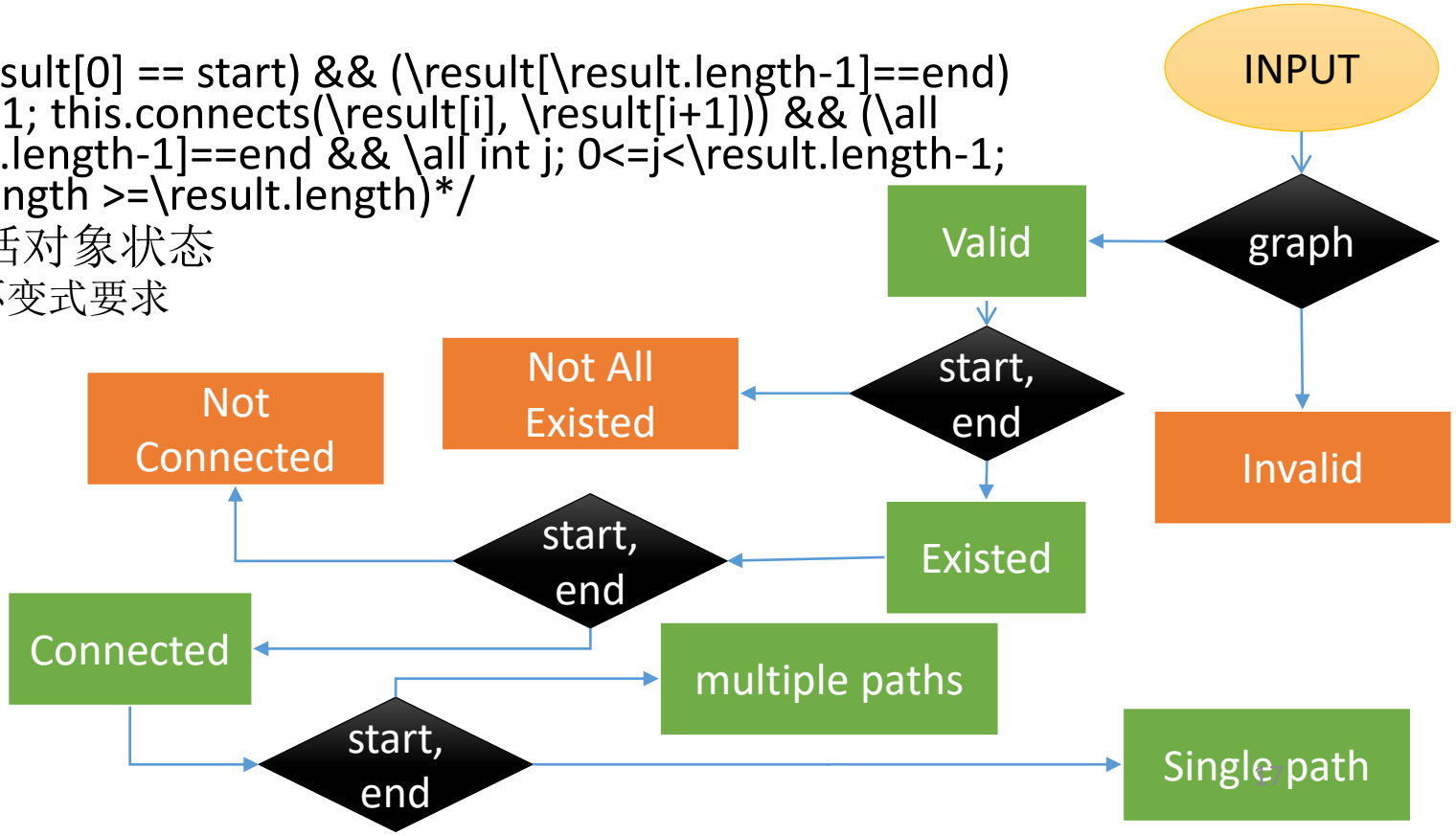
方法实现的正确性论证

- 准备工作
  - Step1：依据前置条件和后置条件划分输入，得到一颗分类树
  - Step2：依次分析每个划分对应后置条件，得到一个决策表
- 输入分析
  - public Vector<String> scan4subs(String dir)
  - /*@requires: Files.isDirectory(dir) == true
  - @effects: \all String p; \result.contains(p) ==> p.substring(dir).equals(dir) && Files.isFile(p) */
  - 输入划分(考虑输入本身和前置条件约束)
    - dir无效
    - dir有效
      - dir下无文件
      - dir下有文件
        - dir下有非隐含文件
        - dir下有隐含文件



方法实现的正确性论证

- 输入分析
  - public List<Node> Shortestpath(Node start, Node end) throw NoPathFoundException
  - /*@requires: this.contains(start) && this.contains(end)
  - @modifies: this
  - @effects: \result!=null ==> (\result[0] == start) && (\result[\result.length-1]==end) && (\all int i; 0<=i<\result.length-1; this.connects(\result[i], \result[i+1])) && (\all List<Node> p; p[0]==start && p[p.length-1]==end && \all int j; 0<=j<\result.length-1; this.connects(p[j], p[j+1]) && p.length >=\result.length)*/
  - 输入不只是方法参数，还可包括对象状态
    - 对象状态要求可以不同于对象不变式要求
  - 输入划分
    - Graph无效
    - Graph有效
      - (Start, end)不在图中
      - (Start, end)在图中
        - (Start, end)不连通
        - (Start, end)连通
          - 单一路径
          - 多路径



方法实现的正确性论证

- 输入分析得到输入划分树
- 检查后置条件对于不同分类的处理情况
  - 确保都有明确处理
- public Vector<String> scan4subs(String dir) throws InvalidDIRException
- /*@requires: Files.isDirectory(dir) == true
- /* @effects: \all String p; \result.contains(p) ==> p.substring(dir).equals(dir) && Files.isFile(p) */

  (Files.isDirectory(dir) == false) ==> *exceptional_behavior*(InvalidDIRException)*/

方法实现的正确性论证

- 梳理后置条件中明确的执行效果
  - <effect Y1> with <input partition X1>
  - <effect Y2> with <input partition X2>
  - …
  - public Vector<String> scan4subs(String dir) throws InvalidDIRException
- /*@effects: \all String p; \result.contains(p) ==> p.substring(dir).equals(dir) && Files.isFile(p)
- (Files.isDirectory(dir) == false) ==> *exceptional_behavior*(InvalidDIRException) */
  - <InvalidDIRException thrown> with < Files.isDirectory(dir) == false >
  - <\result.size==0> with <empty dir>
  - <\result.size=F(dir).num && \result.contains(*f*.name) for any *f* in F(dir)> with <no hidden file in F(dir)>
  - <\result.size=F(dir).num && \result.contains(*f*.name) for any *f* in F(dir)> with <with hidden file in F(dir)>

方法实现的正确性论证

- 梳理后置条件明确的执行效果
  - public List<Node> Shortestpath(Node start, Node end)
  - /*@requires: this.contains(start) && this.contains(end)
  - @modifies: this
  - @effects: \result!=null ==> (\result[0] == start) && (\result[\result.length-1]==end) && (\all int i; 0<=i<\result.length-1; this.connects(\result[i], \result[i+1])) && (\all List<Node> p; p[0]==start && p[p.length-1]==end && \all int j; 0<=j<\result.length-1; this.connects(p[j], p[j+1]) && p.length >=\result.length);
  - \result ==NULL ==>\all List<Node> p; \all int I;0<=i<p.length-1;this.contains(p[i]) && this.connects(p[i], p[i+1]);(p[0]==start)==>(p.include(end)==false)*/
    - <\result==NULL> with <this does not have valid graph>
    - <\result==NULL> with <start or end is not valid node in the graph>
    - <\result==NULL> with <start and end do not connect in the graph>
    - <(\all int i; 0<=i<\result.length-1; this.connects(\result[i], \result[i+1])) && (\all List<Node> p; p[0]==start && p[p.length-1]==end && \all int j; 0<=j<\result.length-1; this.connects(p[j], p[j+1]) && p.length >=\result.length)> with <start and end connect in the graph>

方法实现的正确性论证

- 针对输入的划分和处理分析完毕
  - 每种输入情况都能被后置条件覆盖
  - 对每种输入情况的处理都有明确的效果定义
- 对照代码实现进行论证
  - 检查代码中所有的**return**点，回溯形成输出支撑路径
  - 检查每个输出支撑路径的输出结果集合和输入划分
    - 是否能够映射到后置条件中的某个<effect Y> with <partition X>
      - No, 论证结束，实现不能满足规格
    - 在此路径中是否修改了超出modifies列表之外的非局部对象/变量
      - No, 论证结束，实现不能满足规格
  - 检查所有<effect Y> with <partition X>都已被实现
    - No, 论证结束，实现不能满足规格

方法实现的正确性论证

```
public int choose ( ) throws EmptyException{
/*@Effects: (this.size==0)==>exceptional_behavior(EmptyException);
  this.contains(\result)==>(this.size>0) */
   if(els.size( ) == 0) throw new EmptyException(”IntSet.choose”);
   return els.lastElement( ).intValue;
}
```

```
public int choose ( int n, Vector<Integer> set ) throws EmptyException {
/*@Effects: (this.size==0)==>exceptional_behavior(EmptyException);
   (\result == set.size) ==> \all int x; x<=n; this.contains(x) && set.contains(x)*/
   if(els.size( ) == 0) throw new EmptyException(”IntSet.choose”);
   for(int i = 0; i<els.size();i++)
        if(els.get(i).intValue <= n) set.insert(els.get(i));
   return set.size();
}
```

方法实现的正确性论证

- 调用了无源代码的方法怎么办？
  - 类库
  - 引用的库
- 调用了有源代码的方法怎么办？
  - 无参数输入的调用
  - 有参数输入的调用
- 出现递归调用怎么办？
  - 单个方法的递归调用
  - 多个方法间的递归调用

非递归调用的方法实现正确性论证

- 三种情形
  - y = a.f(args);
  - a.f(args);
  - try{…a.f(args);…}catch(Exception e){…}
- 如果被调用方法实现不正确（或无法确认是否正确）
  - 一般的，无法证明当前方法的实现正确性
- 如果已经证明f的实现正确
  - 排除运行时args不可能处于的input partitions
  - 针对运行时args可能处于的每个<partition X>，获得相应的<effect Y>
    - 在被论证方法逻辑空间中使用<effect Y>进行推理分析

非递归调用的方法实现正确性论证

```
public int countPlainTextFiles (String dir)
/*@Requires: Files.isDirectory(dir) == true
  @Effects: (\result > 0) ==> \exist String[] plainFiles; plainFiles.size ==\result && \all String fname;
plainFiles.contains(fname) ==> scan4subs(dir).contains(fname) && getType(fname)==FileType.plain*
    Vector<String> files = null;
    String fname; int count = 0;
    try{ files = scan4subs(dir);}catch (InvalidDIRException e) {return 0;}
    for(int i = 0; i<files.size;i++){
        fname = files.get(i);
        if(getType(fname) == FileType.plain)count++;
    }
    return count;
}
```

```
public enum FileType{
    plain,
    multimedia,
    office,
    binary,
    unknown
}
```

```
public FileType getType(String file)
<\result==unknow> with <invalid file>
<\result==plain> with <plain text file>
<\result==office> with <office file>
<\result==multimedia> with <image file>
<\result==multimedia> with <audio file>
<\result==multimedia> with <video file>
<\result==binary> with <executable file>
<\result==binary> with <zipped file>
<\result==unknown> with <any other file>
```

```
<\result == 0> with < invalid directory>
<\result == 0> with <empty directory>
<\result == 0> with <no plain text file in dir>
<\result == plainFiles(dir).size> with <plain text files in dir)>
```

<InvalidDIRException thrown> with <invalid dir>
<\result.size=0> with <empty dir>
<\result.size=F(dir).num && \result.contains(*f*.name) for any *f* in F(dir)> with <no hidden file in F(dir) and no sub-dir in dir>
<\result.size=F(dir).num && \result.contains(*f*.name) for any *f* in F(dir)> with <with hidden file in F(dir) and no sub-dir in dir >
<\result.size=F(dir).num+∑scan4subs(S(dir)).size && \result.contains(*f*.name) for any *f* in F(dir) and U(scan4subs(S(dir))>
with <with sub-dir S(dir) in dir>

```
public Vector<String> scan4subs(String dir) throws InvalidDIRException
/*@effects: \all String p; \result.contains(p) ==> p.substring(dir).equals(dir) && Files.isFile(p)
    (Files.isDirectory(dir) == false) ==> exceptional_behavior(InvalidDIRException) */
     Vector<String> files, subs, subfiles;      String fname;
     if(!valid(dir)) throw new InvalidDIRException();
     files = new Vector<String>(); subs = new Vector<String>();subfiles = new Vector<String>();
     fname=getFile(dir);
     while(fname!=null){
         if(!Files.isDirectory(fname))files.add(fname); else subs.add(fname); fname = getFile(dir);
     }
     for(i=0;i<subs.size();i++){
            fname =  subs.get(i);
            try{ subfiles = scan4subs(fname);}catch (InvalidDIRException e) {subfiles.clear();}
            files.addAll(subfiles);
     }
     subs.clear();   return files;
}
```

补充说明

- 全部过程与部分过程对正确性论证的影响
  - 全部过程规定了对所有可能输入的处理
  - 对全部过程的调用不存在违背前置条件的可能性
  - 对部分过程的调用需要检查输入参数是否满足前置条件，否则可能产生未被后置条件规约的状态
    - 导致无法论证正确性
  - 解决办法：通过引入异常处理，把部分过程变成全部过程
    - 一方面让前置条件和后置条件的逻辑闭合
    - 另一方面迫使调用者必须要明确使用一个控制流程来捕捉相应的异常

子类的正确性论证

- 类正确性
  - 抽象对象得到了有效实现
  - 所有方法不会导致不变式为假
  - 类方法实现满足其规格要求
- 子类的正确性论证思路
  - 前提：父类的正确性已经得到了论证
  - step1：论证子类抽象对象得到了有效实现
  - step2：论证子类新增方法的正确性
  - step3：论证子类重写方法的正确性
  - step4：论证子类所有方法不会导致其不变式为假

类型层次下的正确性论证

- step1：子类抽象对象得到了有效实现
  - 如果子类抽象函数结果和父类抽象函数一致，自动获证
  - 如果不一致，则需论证
    - 新增表示对象类型能够有效表示新增抽象对象的所有可能取值
    - 新增表示对象能够有效存储新增抽象对象的所有可能数据

```
public class Elevator{
/*@overview: Elevator is a carrier to take passengers between different floors that can be visited. The door must be closed when it is moving. It must stop exactly at a floor to serve passengers (i.e. go in or out).*/
//rep:
    private int status;  //0: serving; 1: up moving; 2: down moving; 3:idle
    private int infloor;
    private boolean doorStatus;
}
```

```
public class BothSideElevator extends Elevator{
/*@overview: BothSideElevator is a special Elevator. It has doors on both sides, but only one side can be opened when serving at any floor.*/
//rep:
    private boolean side;  //false/true: left/right side door used
}
```

类型层次下的正确性论证

- step2:论证子类新增方法的正确性
  - 前提：父类所有的rep都是private保护
  - 由于不能直接修改父类rep，可按照一般的方法正确性进行论证
- step3: 论证子类重写方法的正确性
  - 子类方法的前置条件不能强于父类方法
    - 父类前置条件分类树中真分支 **蕴含** 子类前置条件分类树中真分支
  - 子类方法的后置条件不能弱于父类方法
    - 子类方法执行效果 **蕴含** 父类方法执行效果
  - 子类实现满足其规格

子类的正确性论证

```
public class MaxIntSet extends IntSet {
/*@OVERVIEW: MaxIntSet is a subtype of IntSet with an additional method, max, to determine the maximum element of the set.*/
     private int biggest;  //holds the maximal integer in the set
     public MaxIntSet ( ){
     /*@EFFECTS: this.size ==0.*/
     …}
     public int max ( ) throws EmptyException{
     /*@EFFECTS: this.size > 0 ==> \result == \max(this); this.size==0 ==> exceptional_behavior(EmptyException).*/
     …}
     public void insert(int x){/*@Effects: this.contains(x) && \all int i;
     0<=i<\old(this).size;this.contains(\old(this)[i]); (x>max(\old(this)))==>max(this) ==x */
          super.insert(x);
          if (x > biggest) biggest = x;
     }
     public void remove(int x){ /*@Effects: \all int i; 0<=i<\old(this).size; (\old(this)[i]!=x)==> this.contains(x)*/
          super.remove(x);
          int i,j;  biggest = Integer.MIN_VALUE;
          for(i=0; i<size();i++){
               j = getAt(i);
               if (j > biggest) biggest = j;
          }
     }
}
```

子类的正确性论证

- step4：论证子类所有方法不会导致不变式为假
  - 子类所有方法通过调用父类方法来更新父类rep
    - 包括构造方法
  - 子类所有方法对rep的更新不会导致相应的不变式为假
    - I_Sub(c) = I_Super(c) && I_Sub_Local(c)

实现正确性论证模板

- 类实现正确性推理模板.docx

作业

- 针对第十三次作业
  - 修复测试发现的bug和重构代码
  - 补充或完善规格
  - 进行正确性论证，按照提供模板来撰写论证过程
- 论证要求
  - 按照要求的步骤进行正确性论证
  - 论证文档务必与代码实现保持一致
- 测试要求
  - 本次作业将由老师来对论证进行结果检查
  - 发现一个论证逻辑错误，报告一个incomplete类型的bug。