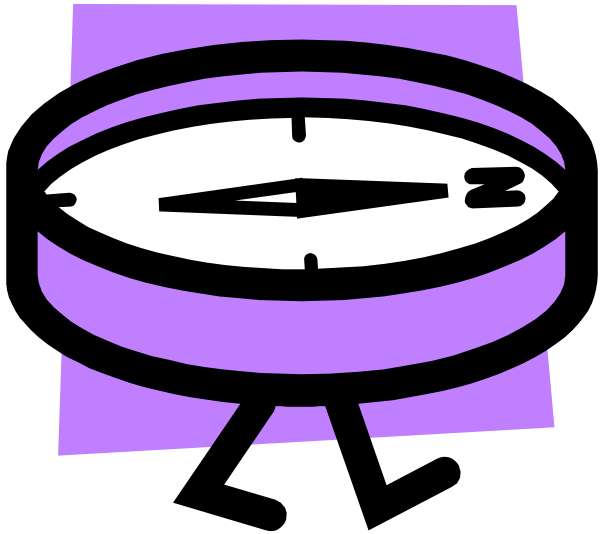
第十讲：类规格及其测试考虑



OO2019课程组

北航计算机学院

内容提要

- 数据抽象规格
- 方法规格的设计讨论
- 数据规格
- 如何使用数据抽象
- 数据抽象的实现
- 基于规格的测试
- 作业

数据抽象规格

- 数据抽象是关于类型的一种抽象
  - 数据抽象 = <数据内容，操作>
  - 数据内容定义了对象状态空间，即数据抽象的状态空间
  - 操作作用于对象状态空间，定义对象外部可见的规范化行为
- 类型
  - 从规格角度：由对象化的组成成分及其约定的操作方式组成
  - 从运行时角度：是对一块内存区域的模板化访问

数据抽象规格

- 用户看到的数据抽象规格整体包括三个部分
  - 数据抽象的内容描述
    - 例：JDK关于java类所管理数据内容的一段描述
    - *The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.*
  - 规格的构造函数
    - 定义了新对象如何初始化和初始状态
  - 数据抽象操作的描述
    - 例：JDK关于类操作的描述（返回值类型、方法名及参数的解释）
    - *public boolean isEmpty() Returns true if, and only if, length() is 0.*

数据抽象规格

- 需要存储哪些信息，如何有效操作相应的信息是设计数据抽象时首先需要回答的问题
  - 设想需要存储日期时间信息，year, month, day, hours, minutes, seconds, day of week
  - 需要使用某种数据结构来表示存储
  - 并提供相关的操作，如计算星期日期，比较日期先后顺序等
- 如果把数据的存储实现方式(即具体的数据结构)敞开给外部，则存储结构的变化会导致使用者程序的变化
  - 如在80s~90s，year的存储普遍采用2位十进制数字，引发了千年虫(Y2K)问题，很多软件的源代码都要进行修改
- 因此，我们强调封装机制
  - 把数据内容与数据的内部表示结构区分开
  - 对外公开的是数据内容（通过方法访问获得）
  - 内部表示结构不对外公开

数据抽象规格

- 高级编程语言提供了基本数据类型和构造抽象数据类型的手段
  - 基本数据类型
    - integer, boolean, characters, …
  - 抽象数据类型
    - list, table, array, 结构体…
- 为每一类数据类型提供了相应的操作
  - 如赋值、读取、比较等
  - 大部分抽象数据类型的内部表示都不允许直接访问
    - 如Java中的ArrayList，看不出其内部的具体存储结构
  - 有些语言允许访问某些抽象数据类型的内部表示
    - C语言允许使用指针访问数组的内部表示
    - 破坏信息封装

数据抽象规格

- 自定义数据类型可以提高信息封装程度，否则相关信息就散落在一堆变量中
- 数据操作的设计**取决于软件需求**
  - 银行账号: 打开、关闭、取款、存款、查询余额等
  - 图(graph): 初始化、增加节点、删除节点、检查节点间的连通性等
- 几乎所有高级语言都允许构造数据类型
  - 但是并不要求描述数据抽象，事实上开发人员必然要做出如下设计：
    - 我用这个类来管理/存储...数据
  - 也不对数据的内部结构是否公开进行检查

数据抽象规格

不失一般性，数据抽象操作包括四类：

- 构造操作(constructor)
  - 创建相应类型的新对象，满足初始状态要求
- 更新操作(mutator)
  - 更新对象的状态
- 观察操作(observer)
  - 观察对象的状态属性
- 生成操作(producer)
  - 根据当前对象生成新的对象，但不改变当前对象

不可变对象

- 不提供更新操作

Example：集合

构造操作：
- 创建空集
- 创建包含指定元素的集合

更新操作：
- 往集合中插入元素
- 从集合中删除元素

观察操作：
- 集合规模
- 判断集合是否相等
- 检查集合是否为空集

生成操作：
- 生成子集
- 生成与给定集合的并集
- 生成与给定集合的交集

整数集合规格

```
public class IntSet {
  //@ public model non_null int[] ia;
    //@ensures ia.length==0;
  public IntSet (){} //构造操作
    /*@assignable ia
      @ensures (\exists int j;0<=j<ia.length;ia[j]==x); @*/
  public void insert (int x){} //更新操作
    /*@assignable ia
      @ensures (\forall int j;0<=j<ia.length;ia[j]!=x); @*/
  public void delete (int x) {} //更新操作
    /*@ensures \result==(\exist int i; 0<=i<ia.length; ia[i]==x);@*/
  public /*@pure@*/ boolean isIn (int x){} //观察操作
    /*@ normal_behavior
      @ requires a!=null;
      @ assignable ?
      @ ensures ?
      @ also
      @ exceptional_behavior
      @ ??       @*/
  public IntSet intersection (IntSet a) throws NullPointerException{} //生成操作
}
```

把下面的操作加入到IntSet中：所属类别和规格

public IntSet union (IntSet a)

方法后置条件的规约

- 方法执行可能会产生三种不同的返回结果
  - 通过显式的return或throw来返回处理结果
    - 需区分正常情况和异常情况下的处理结果（返回机制不同）
    - 需要满足ensures子句和相应的signals子句
  - 通过修改所在this对象的状态来返回处理结果
    - 会改变this的状态
    - 需满足ensures子句和assignable子句
    - 需满足this对象的invariant和constraint
  - 通过修改方法输入参数对象的状态来返回处理结果
    - 会改变输入参数对象的状态
    - 需满足ensures子句和assignable子句
    - 需满足输入参数对象的invariant和constraint
  - 这三种不同的返回机制可能同时使用，但显然应确保KISS！

方法规格的设计

- 和实现代码一样，大部分情况下不能直接写代码，而是需要设计
  - 特别对于构造性规格
- 构造性规格：即需要构造{若干}中间数据表示来表达相应的逻辑
- Path. getDistinctNodeCount
  - 构造一个节点集合(int[])，其中无重复元素，且包含this中每个节点
  - \result==节点集合.length
- PathContainer. getDistinctNodeCount
  - 构造一个节点集合(int[])，其中无重复元素，且包含this中每个path中的每个节点
  - \result==节点集合.length

方法规格的设计

- 和实现代
  - 特别对
- 构造性规格：即需要构造{若干}中间数据表示来表达相应的逻辑
- P

int[] PathContainer.getReachableNodeSet(nodeId)
给出由给定nodeId可达的节点集合

而是需要设计

```
/*@ ensures (\exists int[] arr; (\forall arr元素i,j; arr[i] != arr[j]);
  @ (\forall arr元素; (this中存在一个Path p; p中有节点arr[i]))) &&
  @ (\forall this中路径p; (\forall p中节点node; (arr中\exists一个元素i;
  @ node == arr[i]))) &&(\result == arr.length));
  @*/
```

- PathContainer. getDistinctNodeCount
  - 构造一个节点集合(int[])，其中无重复元素，且包含this中每个path中的每个节点
  - \result==节点集合.length

方法间引用的规格

- 任何情况下，如果当前类或所依赖的类已经提供了相应pure方法，则应直接使用相应方法来构造当前的方法规格
  - 大家可以在作业中给定的规格中找到这样的示例
- 什么样的方法应标注为pure方法？
  - 无副作用的方法
  - 任何情况下的执行都会正常结束或者抛出异常
  - 规格逻辑较为简单的方法
- 方法规格中的引用
  - 可引用pure型方法
  - 可引用所依赖对象中public的规格数据内容(public model ***)

关于数据本身的规格

- 任何时刻对象实例数据所必须满足的要求
  - //@invariant (\forall int i,j; 0<=i&&i<j&&<a.length;a[i]!=a[j]);
  - 不变式是概括对象状态正确性的核心，也是形式化方法的核心概念
  - Daikon工具（Microsoft）甚至可以自动基于对象执行信息来归纳不变式
  - 一旦不满足不变式要求，对象任何方法的执行结果都可能无效，即便满足方法本身的后置条件
- 任何时刻**修改**对象实例数据所必须满足的要求
  - 修改后的状态与修改前的状态所必须满足的约束条件
  - //@constraint Math.abs(a.length -\old(a.length))<=1;
  - constraint是概括对象状态变化delta正确性的核心：delta的不变式

invariant和constraint的示例

- IntSet
  - 规格数据内容：int[] ia
  - invariant ia != null && (\forall int i,j; 0<=i&&i<j&&j<ia.length; ia[i] != ia[j]);
  - constraint Math.abs(ia.length-\old(ia.length))<=1;
- Poly
  - 规格数据内容：int[] cof, int[] deg
  - invariant cof != null && deg!= null&&cof.length == deg.length && (\forall int j;0<=j &&j<cof.length;cof[j]!=0);
  - constraint cof.length = \old(cof.length);

invariant和constraint的示例

- 电梯类
  - 规格数据内容：int infloor, STA status, boolean closed, int target_floor, int top_floor

```
invariant (status != DOCK)==>closed == true && 1<= infloor <= top_floor &&
                0<=target_floor <=top_floor;
invariant (target_floor == infloor) ==> status == DOCK;
invariant (target_floor == 0) ==> status == IDLE;
constraint Math.abs(infloor-\old(infloor))<=1;
```

完整理解类的规格

- 当使用者调用一个方法时
  - 对象必须有效，否则不太可能获得正确的处理结果。
    - 是否应该要求使用者保证对象有效？
  - 要求使用者所提供输入满足前置条件
- 当方法执行结束时
  - 必须满足方法的后置条件
  - 必须确保对象有效
    - invariant为真
    - constraint为真

关于整数集合规格的讨论

- 关于集合的数学知识支撑我们设计出集合的行为操作要求
  - 集合的插入、删除
  - 集合的交集和并集
- 集合状态的要求
  - 不能有重复元素
  - 每次只能增加或删除一个元素

讨论：对涉及大规模数据存储的软件而言，不可能使用单一集合来管理所有的整数。假设要求每个整数集合能够自动根据所管理数据的规模和大小情况，分裂出新的整数集合，且满足this集合< split集合。这样的集合如何设计其规格？

如何使用数据抽象

- 用来声明一个数据抽象的数据内容
  - //@ public model non_null **Path**[] pList;
- 用来实现一个数据抽象中的数据结构
  - private HashMap<Integer, **Path**>;
- 使用方法规格来定义一个方法的规格
  - //@ requires p!=null && p.**isValid**();
  - 注意被引用的方法规格必须在JML中声明为/*@pure@*/
- 使用方法规格来实现一个方法
  - try{…myset.**removePath**(p)…}catch(**PathNotFoundException** e){…}

和直接使用类型实现的最大区别在于：可以基于规格进行推理。使用者的规格建构在被使用的规格之上。

如何使用数据抽象

- 数据抽象规格为使用者和实现者定义了一份**契约(contract)**，采用规约的方法
  - 使用者无需关心一个类如何实现数据，只需要了解这个类管理的数据内容和对数据的管理行为
  - 实现者关心一个类如何保存数据，确定相应数据的类型和存储结构(即数据结构) ---->从而能够有效、高效率和健壮的实现所承诺的契约！
- 使用者的职责与权益
  - 确保清楚契约对于方法规格和状态约束的要求
  - 确保按照契约所规定的方式进行操作，包括提供相应的输入数据
  - 完全拥有实现者所提供能力的使用权

数据抽象的实现

- 基于规格的实现
  - 数据实现：使用具体数据结构来实现model所定义的数据内容(representation, 简称rep)
  - 方法实现：基于数据实现来实现方法规格
- 数据表示
  - 需要存储哪些数据？
  - 使用何种方式存储？
  - 方法如何高效率的访问数据？
  - 数据状态需要满足哪些要求？
- 方法实现
  - 如何按照给定输入提供相应输出？---算法流程
  - 如何确保不会违背数据要求？---契约保证

数据抽象的实现

- 数据表示
  - 需要存储哪些数据？
    - IntSet：规模未知的一组无重复整数
    - Poly：多项式的所有项(项数未知)
  - 如何存储这些数据？
    - IntSet：使用[静态/动态]数组/链表/向量
    - Poly：使用[静态/动态]数组/链表/向量
  - 使用什么类型来表示？
    - IntSet：向量不能存储int，采用Integer
    - Poly：两个int数组(系数、幂)

数据抽象的实现

- 数据表示
  - 对数据的访问效率
    - IntSet
      - Java的向量提供了丰富的访问方式。增加：add(.);获取指定索引的数据get(.);获取向量规模size(); 获取最后一个元素lastElement()
    - Poly
      - 如果采用两个向量，同上可以使用丰富的访问方式
      - 注意：确保对两个向量的访问是对齐的
      - 多项式的degree如何获得？

```
public class Poly{

    public Poly()
    public Poly(int c, int n)

    public int degree()
    public int coeff(int d)

    public Poly add(Poly q)
    public Poly sub(Poly q)
    public Poly mul(Poly q)
}
```

数据抽象的实现

```
public class IntSet {
  //@ public model non_null int[] ia;
      private Vector<Integer> els;
    //@ensures ia.length==0;
  public IntSet (){els = new Vector<Integer>();} //构造操作
    /*@assignable ia
      @ensures (\exists int j;0<=j<ia.length;ia[j]==x); @*/
  public void insert (int x){} //更新操作
    /*@assignable ia
      @ensures (\forall int j;0<=j<ia.length;ia[j]!=x); @*/
  public void delete (int x) {} //更新操作
    /*@ensures \result==(\exist int i; 0<=i<ia.length; ia[i]==x);@*/
  public /*@pure@*/ boolean isIn (int x){} //观察操作
    /*@ normal_behavior
      @ requires a!=null;
      @ assignable ?
      @ ensures ?
      @ also
      @ exceptional_behavior
      @ ??       @*/
  public IntSet intersection (IntSet a) throws NullPointerException{} //生成操作
}
```

数据抽象的实现

```
public class IntSet {
  //@ public model non_null int[] ia;
      private Vector<Integer> els;
    //@ensures ia.length==0;
  public IntSet (){els = new Vector<Integer>
    /*@assignable ia
      @ensures (\exists int j;0<=j<ia.length
  public void insert (int x){} //更新操作
    /*@assignable ia
      @ensures (\forall int j;0<=j<ia.length
  public void delete (int x) {} //更新操作
    /*@ensures \result==(\exist int i; 0<=i<
  public /*@pure@*/ boolean isIn (int x){} /
    /*@ normal_behavior
      @ requires a!=null;
      @ assignable ?
      @ ensures ?
      @ also
      @ exceptional_behavior
      @ ??       @*/
  public IntSet intersection (IntSet a) throws NullPointerException{} //生成操作
}
```

```
{
    Integer e = new Integer(x);
    int i;
    for(i=0; i<els.size();i++)
      if(els.get(i).equals(e)){
          els.set(i, els.lastElement());
          els.remove(els.size()-1);
          return;
      }
}
```

数据抽象的实现

- Poly
  - 数据存储： int [] terms; int deg;
    - 使用下标i表示多项式的幂(隐含存储)，terms[i]表示对应的系数
    - deg为各个项中最大的幂
  - 可能会带来大量的存储冗余
    - $x+2x^{100}$

```
public Poly add (Poly a) throws NullPointerException{
    if(a==null)throw new NullPointerException("Poly add(Poly)");
    int newdeg;
    Poly lg,sm;
    if(this.deg > a.deg) {lg = this;sm=a;} else {lg = a;sm=this;}
    newdeg = lg.deg;
    if(this.deg == a.deg){
       for(int i=this.deg;i>0;i--)
         if(this.terms[i]+a.terms[i]!=0)break;else newdeg = newdeg-1;
    }
    Poly p = new Poly(newdeg);
    int i;
    for(i=0;i<sm.deg &&i<=newdeg;i++)
       p.terms[i] = this.terms[i] + a.terms[i];
    for(int j=i;j<=newdeg;j++)
       p.terms[j] = lg.terms[j];
    return p;
}
```

数据抽象的实现

- Poly
  - 数据存储
    terms; in
    - 使用
      项式的
      储)，t
      对应的
    - deg为
      大的
  - 可能会
    的存储
    - x+

```
/*@ normal_behavior
  @ requires a!=null;
  @ assignable \nothing;
  @ ensures (\forall int i; 0<=i&&i<a.deg.length||i<this.deg.length);
              \result.coeff(i)==this.coeff(i)+a.coeff(i));
  @ also
  @ exceptional_behavior
  @ requires a==null;
  @ assignable \nothing;
  @ signals_only NullPointerException;
  @*/
```

```
/*@ ensures (\exists int i;0<=i&&i<deg.length;deg[i]==d &&\result == coeff[i]);
  @ ensures (\forall int i;0<=i&&i<deg.length;deg[i]!=d) ==> \result == 0;
  @*/
  public /*@pure@*/ int coeff(int d){}
```

```
                p.terms[j] = lg.terms[j];
            return p;
        }
```

数据抽象的实现

- 引入稀疏存储格式
  - 只存储多项式中存在的项
  - terms[i]与i之间的隐含关系不成立了！
  - 采用两个数组：int[] terms; int[] coeff;
  - cx^n: terms[i] = n; coeff[i] = c;
  - 问题1：构造器如何编写？
  - 问题2：在做运算开始时，不知道新的多项式会有多少项！

```
/*@normal_behavior
  @ensures coeff(n)==c && cof.length==1;
  @ also
  @ exceptional_behavior
  @ signals (NegativeExponentException e)(n<0);
  @*/
public Poly (int c, int n) throws
NegativeExponentException
{
   if(n<0) throw new
     NegativeExponentException("Poly(int, int)");
   if(c == 0){
      terms = new int[1];
      deg = 0;
      return;
   }
   terms = new int[n+1];
   terms[n] = c;
   deg = n;
}
```

数据抽象的实现

- 引入稀疏存储格式
  - 让terms和coeff能够根据需要自动增长
  - 无需通过new来提前申请所需的空间
  - Poly(int c, int n)的存在必要性就不大
  - 取代为Poly(int [] c, int[] n)，便于用户一次性构造所需的多项式

```
/*@ requires c!=null && n!= null && c.length == n.length;
  @ ensures (\forall int i;0<=i&&i<n.length;cof[i]==c[i] && deg[i]==n[i]);
  @*/
public Poly(int [] c, int[] n){…}
```

数据抽象的实现

- 引入稀疏存储格式

```
public Poly add(Poly a) throws NullPointerException
/*@ensures:…
p.terms[i] = this.terms[i] + a.terms[i]; -->
{p.coeff.add(this.coeff.get(i)+a.coeff.get(i));
p.terms.add(this.terms.get(i));}
```

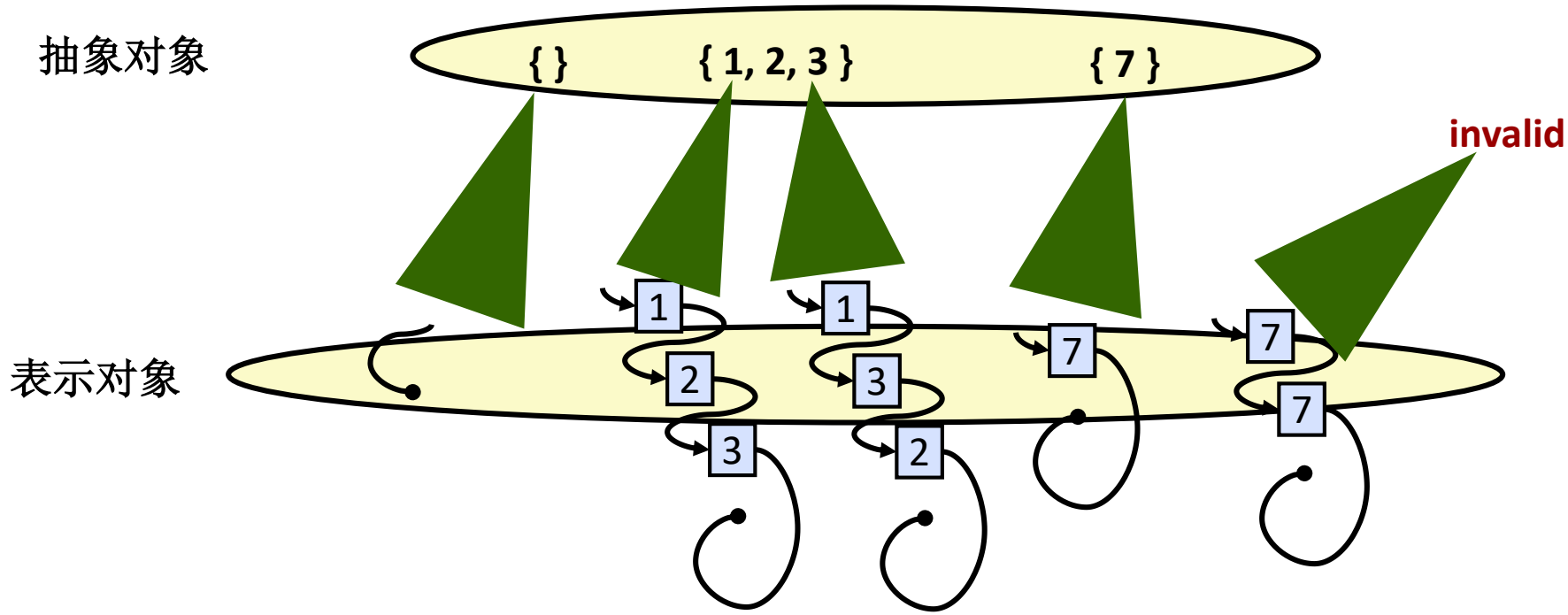
  - 对齐对terms和coeff这两个向量的访问不方便，甚至可能会出现潜在的问题，我们希望把这两个数据整合在一起

```
class Term{
    private int coe; private int deg;
    public Term(int c, int n){coe = c; deg = n;}
    public int coeff(){return coe;}
    public int degree(){return deg;}
}
```

这时如何修改Poly类的表示和add方法？

数据抽象的实现

- 一个数据抽象可以有多种实现，对用户透明
  - 任何一种实现都必须满足规格要求（兑现承诺）
  - 表示对象可以映射到抽象对象(多个表示可以映射到同一个抽象对象)



针对不变式的检查

- 不变式本质上是对表示对象是否有效的判定
  - 不变式成立==》对象有效==》对象方法能够满足规格要求
  - 我们希望把不变式实现为一个判定方法
  - public boolean repOK()
  - /*@ensures: \result==invariant(this).

```
IntSet:
public boolean repOK(){
        if(els == null) return false;    //els <> null
        for (int i=0; i<els.size();i++){
              Object x = els.get(i);
              if(!(x instanceof Integer)) return false;   //els[i] is an Integer
              for(int j = i+1; j<els.size();j++) if(x.equals(els.get(j)))return false;  //els[i] <>els[j] for i<j
        }
        return true;
}
```

针对不变式的检查

- 如果一个对象c的rep不能支撑规格数据内容，则该对象不可能有效
- 如果一个对象c的表示不变式成立，意味着对象一定有效
- 用户可以随时调用一个对象的repOK，检查一个对象的表示状态是否有效c.repOK()<==>invariant(c)
- 对象的状态更新方法可以在更新状态之前调用repOK来检查对象是否有效，如果无效可以通过throw InvalidStateException来提醒用户当前对象的状态无效
- 测试程序可以通过调用repOK来判断程序是否出现了问题
- 在实现一个类时，repOK应该与不变式在早于任何其他方法之前实现

类的设计与实现策略

- 定义类的规格
  - 类的目标(数据内容定义)
  - 类的方法及其规格
- 类的设计实现
  - 实现类的属性
  - 实现repOK
  - 实现类的构造器
  - 实现类的观察方法
  - 实现类的生成方法
  - 实现类的更新方法

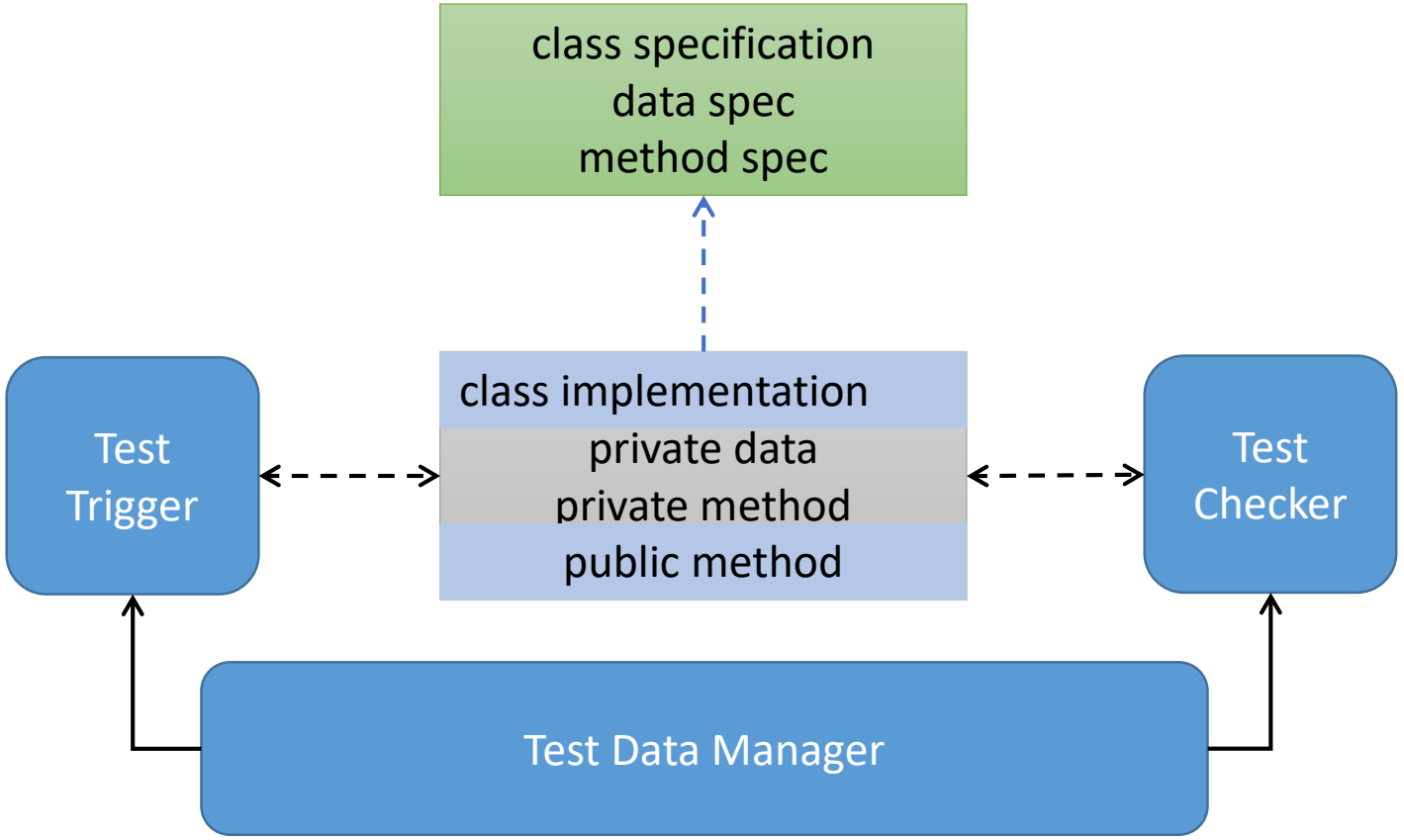
IntSet:

```
public boolean repOK(){
        if(els == null) return false;
        for (int i=0; i<els.size();i++){
            Object x = els.get(i);
            if(!(x instanceof Integer)) return false;
            for(int j = i+1; j<els.size();j++) if(x.equals(els.get(j)))return false;
        }
        return true;
}
/*@assignable this
  @ensures isIn(x);
  @ensures (\forall int i; 0<=i&&i<\old(ia.length);
  @     \old(isIn(\old(ia[i])))==>isIn(\old(ia[i])));
  @*/
public void insert(int x){
      els.add(new Integer(x));
}
```

类的设计与实现策略

- 避免暴露对象表示
  - public Vector allEls(){ return els;//els is a Vector)
- 避免外部直接对对象表示进行操作
  - public IntSet(Vector v){els = v;}
- 如果某种情况下用户一定要获得对象中所保存的所有对象数据
  - 返回其拷贝，而不是原始数据
  - public Vector allEls(){ return (Vector)(els.clone());//els is a Vector)
  - 如果Vector中存储的对象是自定义的类型
    - 需要实现Cloneable接口，否则els.clone()会触发抛出CloneNotSupportedException

基于规格的测试：Framework



- Test Trigger
  - 使用Test Data来构造被测对象(测试准备)
  - 使用Test Data来发起测试动作(调用被测对象方法)，并获得方法执行结果
- Test Checker
  - 使用Test Data直接对方法执行效果进行检查
  - 使用Test Data和被测对象的查询方法来对方法执行效果进行检查
- Test Data Manager
  - 针对被测类的data spec和method spec所设计的针对性数据
  - 提供数据访问和更新接口

基于规格的测试

- 规格为Test Trigger, Test Checker和Test Data Manager提供了设计依据
  - test a *class implementaion* according to its *specification*
- 测试目标
  - 每个方法是否都满足规格？
  - 是否在任何使用场景下，类都能确保状态正确？
- 测试有效性问题
  - 需要多少组测试数据？
  - 测试覆盖了多少代码成分？

基于规格的测试：准备数据

- 前置条件涉及的数据+方法输入参数
  - 大的划分：满足前置条件、不满足前置条件
  - 细致划分：针对每个数据项，按照约束条件和数据特征来划分
- 后置条件涉及的数据
  - 用以判断执行效果的参考数据
  - 特点：常常与输入数据和对象状态有关，动态性
- 不变式和修改约束涉及的数据
  - 如何通过方法调用序列获得相应的对象状态？
  - 用以判断状态是否正确的参考数据

数据准备Example

```
/*@ ensures (\exists int[] arr; (\forall int i, j; 0 <= i && i < j && j < arr.length; arr[i] != arr[j]);
  @             (\forall int i; 0 <= i && i < arr.length;this.containsNode(arr[i]))
  @          && (\forall int node; this.containsNode(node); (\exists int j; 0 <= j && j < arr.length; arr[j] == node))
  @          && (\result == arr.length));
  @*/
public /*pure*/ int getDistinctNodeCount();
```

对象状态数据
(1,1), (1,2), (1,2,1)
(1,2,2,1), …
(1,2,2,1,1)…
(1,2,2,1,1,2)…

```
/*@ also
  @ public normal_behavior
  @ requires obj != null && obj instanceof Path;
  @ assignable \nothing;
  @ ensures \result == (((Path) obj).nodes.length == nodes.length) &&
  @                    (\forall int i; 0 <= i && i < nodes.length; nodes[i] == ((Path) obj).nodes[i]);
  @ also
  @ public normal_behavior
  @ requires obj == null || !(obj instanceof Path);
  @ assignable \nothing;
  @ ensures \result == false;
  @*/
public boolean equals(Object obj);
```

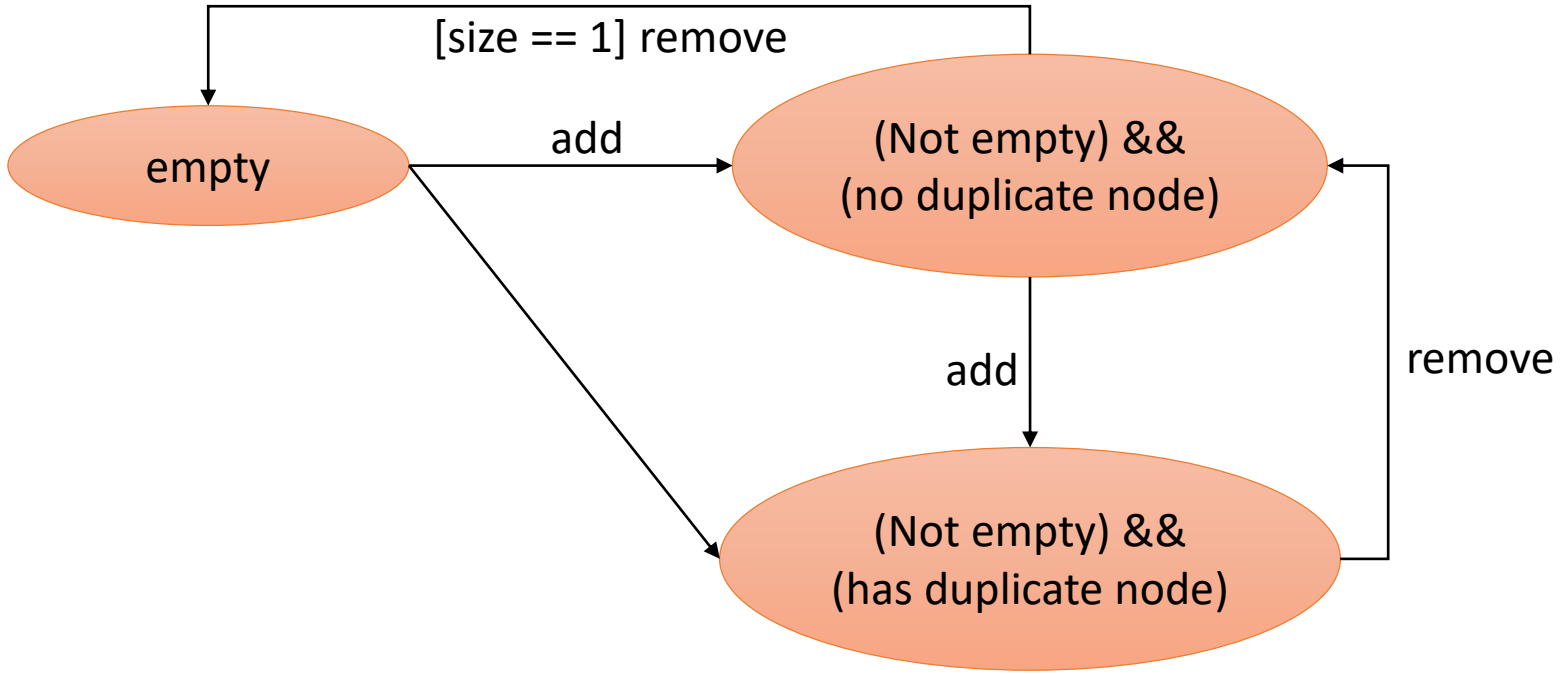
前置条件数据
(obj == null), (obj != null)
(obj as Integer), (obj as Path), (obj as Object)
(obj != this), (obj ~= this), (obj == this)
(obj与this的部分相同pattern)

基于规格的测试：准备场景

- 模拟使用者对象与被测对象的交互
  - 通过被测对象提供的方法
  - 始终注意检查对象的状态
- 测试场景往往具有一定的实际意义
  - 往往对应着功能场景
  - 比如： new file; open file; append; append; remove; close file
- 测试场景的发现错误能力一般会显著高于单一的方法调用
  - 源自于对象状态的更强覆盖能力

给定一个对象状态，如何快速获得这样的对象？

场景准备Example



以对象状态为目标，构造状态迁移操作

在具体状态中执行观察操作和进行判断

int pid1=addPath(p1); int pid2=addPath(p2);

removePathById(pid1);Path p = getPathById(pid2);

check (p.equals(p2));…

基于规格的测试：自动化

- 可以使用Java语言来实现这样的测试，具有可扩展性。数据独立于测试用例。
- 也可以基于junit来实现这样的测试
- 这样获得的好处
  - 测试可以自动化，只要代码发生变化，可以自动回归
  - 通过维护规格和测试代码的一致性，软件质量水平得到了保持
  - 其实有针对JML的工具可以自动生成测试用例

作业

- 在PathContainer基础上实现Graph
  - 给出了相应的接口规格定义
  - 注意：规格数据内容未变
- Graph特征
  - 无向图: (nodei,nodej) <==>(nodej,nodei)
  - 默认(node,node)是一条连通边
  - 由后验观察到的{path}来定义其连通性
- Graph的实现
  - 数据结构的选择很关键
- 强烈建议同学们使用Junit来实践本次课程所介绍的测试！！！