《面向对象设计与构造》
Lec11-类型层次规格与验证

2024

OO课程组

北京航空航天大学计算机学院

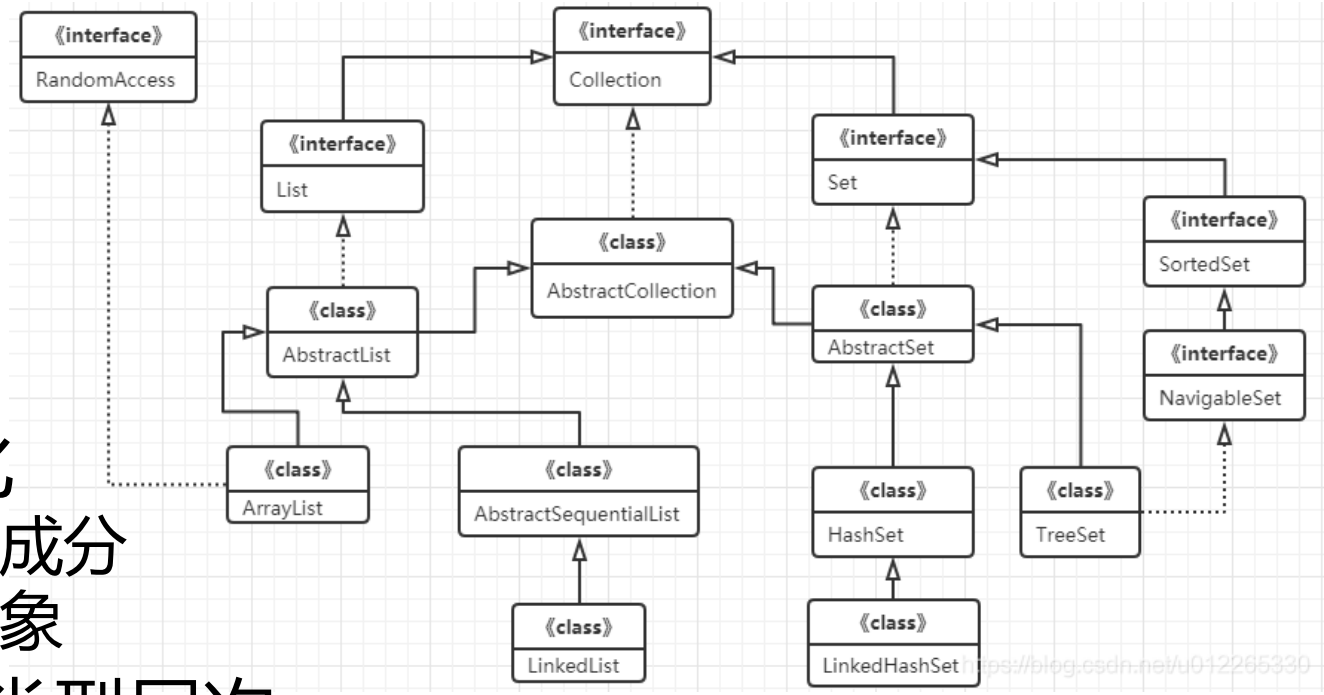
本讲提纲

- 回顾类型层次
- 类型层次下的规格关系
- 集合元素的迭代访问
- 基于规格的验证测试
- 作业

回顾类型层次

- 类型
  - 数据及其操作的规格化
  - 0001 --> (boolean)true; (integer)1;…
  - 在OO范畴下，规格化的不只是数据解析，还包括对数据的处理
    - 例：对集合有哪些处理?
- 类型层次是一种对数据及其处理的系统化抽象结果
  - 逐层往上抽取共性特征
  - 逐层往下增强细节描述能力
- 在Java语言中通过继承机制或接口机制来定义类型层次

回顾类型层次



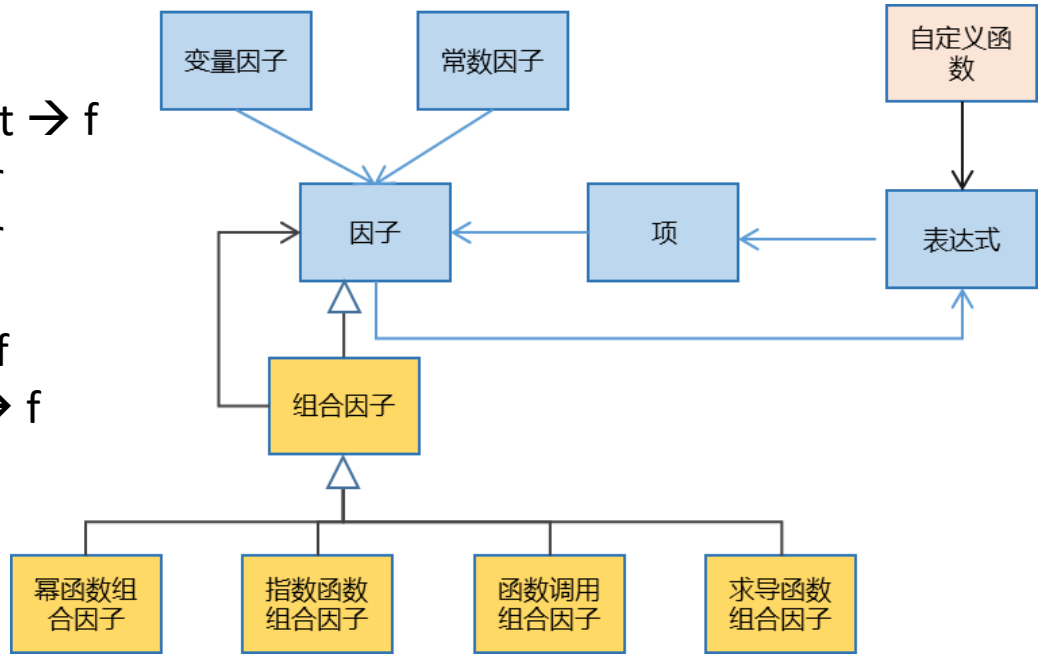
- 继承往下的本质是扩充+扩展
  - 扩充：增加新的属性和操作
  - 扩展：重写(override)已有的操作
- 继承往上的本质是提炼+一般化
  - 提炼：提取子类数据内容的共性成分
  - 一般化：子类具体行为的规格抽象
- 继承机制并不总能得到有效的类型层次
  - 给定集合类ArrayList，继承出CourseList类==》具有类型层次吗?
- LSP替换原则
  - 在任何父类型对象出现的地方使用子类对象都不会破坏user程序的行为

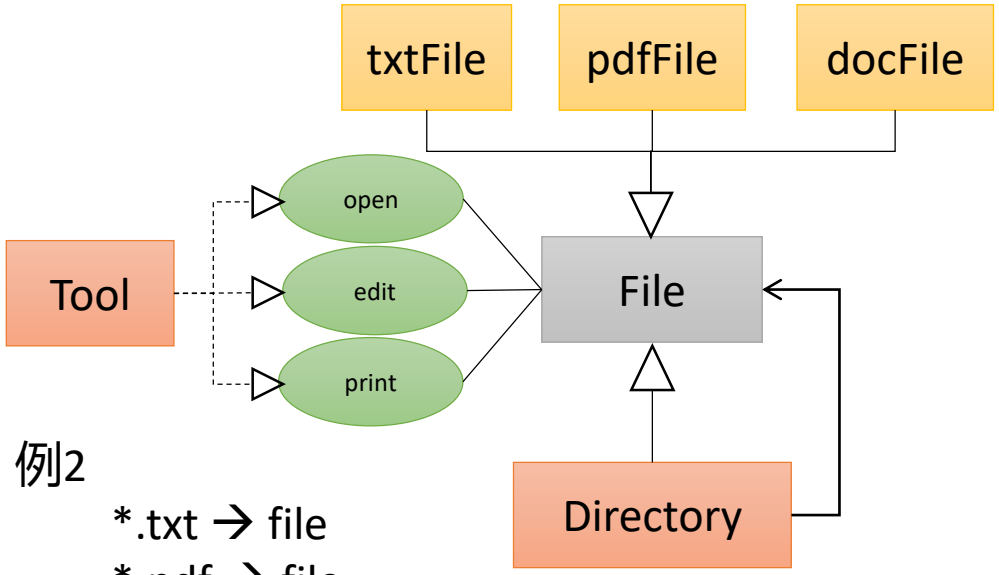
回顾类型层次

- 通过继承或接口实现都能得到归一化的表示与处理能力，区别在于继承具有严格的数据抽象层次约束，而接口可以自由跨越多个数据抽象层次

例1

Var|Const → f
f+f→expr
f*f→expr
f-f→expr
f(expr)→f
e(expr) → f
g(f)→f
d(f)→f





例2

*.txt → file
*.pdf → file
*.doc → file
directory → file

类型层次设计

- 子类继承了父类的规格
- 子类可以重写父类的方法规格
- 子类可以扩充父类的方法规格和数据规格
  - 子类的数据抽象：父类的数据抽象+新增的数据抽象
  - 子类的数据规格：父类的数据规格+扩展的数据规格+新增的数据规格
  - 子类的方法规格：父类的方法规格+重写的方法规格+新增的方法规格
- 为了满足LSP
  - 子类重写父类方法时不能与父类的方法规格相冲突
  - 子类新增和扩展的数据规格不能与父类数据规格相冲突

类型层次设计：示例分析

```
public class A{
//public model non_null T[] ta;
/*@ assignable \nothing;
   @ ensures contains(x)==>ta[\result] ==  x ;
   @ signals (NotFoundException e) !contains(x);
   @*/
   public int find(T x) throws NotFoundException
}
```

B1是错的：因为！contains(x)时父类A要报出异常，而B1则让result为-1

B3：假如otherException和notfoundException有关系，那么是对的；否则是错的

```
public class B1{
/*@assignable \nothing;
   @ensures contains(x)==>ta[\result]== x;
   @ensures !contains(x) ==>\result=-1;
   @*/
public int find(T x) throws NotFoundException
}
```

```
public class B2{
//public model T lastFound;
/*@assignable this;
   @ensures contains(x)==>ta[\result]==x &&
lastFound ==x;
   @signals (NotFoundException e) !contains(x);
   @*/
   public int find(T x) throws NotFoundException
}
```

B2是对的，增加了新的东西，但是没有改变父类规格

```
public class B3{
/*@assignable \nothing;
   @ensures contains(x) ==>ta[\result]==x ;
   @signals (OtherException e) !contains(x);
   @*/
  public int find(T x) throws OtherException
}
```

类型层次设计

- 子类可以对父类进行扩充，但需要保持父类的规格仍然成立

```
public class IntSet {
  //@public model non_null int[] ia;
  //@public invariant (\forall int i,j;0<=i&&i<j&&j<ia.length;ia[i]!=ia[j]);

  //@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 \nothing;
      @ ensures (\forall int ele; this.isIn(ele);a.isIn(ele)<==>\result.isIn(ele));
      @ also
      @ exceptional_behavior
      @ signals (NullPointerException e) a==null;      @*/
  public IntSet intersection (IntSet a) throws NullPointerException{} //生成操作
}
```

类型层次设计

- 子类可以对父类进行扩充，但需要保持父类的规格仍然成立

```
public class IntSet {
  //@public model non_null int[] ia;
  //@public invariant (\forall int i,j;0<=i&&i<j&&j<ia.length;ia[i]!=ia[j]);

  //@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]!=
  public void delete (int x) {} //更新操作
    /*@ensures \result==(\exist int i; 0<=i<ia.length
  public /*@pure@*/ boolean isIn (int x){} //观察操作
    /*@ normal_behavior
      @ requires a!=null;
      @ assignable \nothing;
      @ ensures (\forall int ele; this.isIn(ele);a.is
      @ also
      @ exceptional_behavior
      @ signals (NullPointerException e) a==null;
  public IntSet intersection (IntSet a) throws NullPo
}
```

```
public class MaxIntSet extends IntSet {
//@public model int max;
//@public invariant (\forall int ele;this.isIn(ele);max>=ele);
//@public invariant (ia.length>0)==> this.isIn(max);

  //@ensures ia.length == 0 && max == MINIMAL_INT;
  public MaxIntSet ( ){ }

  //@ensures \result == max;
  public/*@pure@*/ int max () {}

  /*@assignable this;
    @ensures this.isIn(x) && (x>\old(max)==>(max == x));
    @*/
  public void insert (int x){ }
}
```

抽象对象存在多种表示

- 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）
    - 具体实现(表示对象)可以映射到数据抽象
    - 具体实现的变化满足在数据抽象上定义的相关要求(invariant, constraint)
- Poly对象的抽象与实现
  - 数据抽象： int[] cof, int[] deg
  - 不变式: (cof.length == deg.length) && (\forall int i; 0<=i && i<cof.length;cof[i]!=0)
- Poly的紧凑实现(DensePoly)
  - int[] terms, int degree
- Poly的稀疏实现(SparsePoly)
  - Vector<Term> terms
- 开发者/实现者如何判断数据实现是否满足规格?
  - 需要在具体数据实现(rep)与数据抽象之间建立明确的映射
  - 通过规格层次的invariant和constraint来进行静态检查
  - 通过repOK来进行运行时检查

抽象函数

- 针对数据抽象要存储和管理的数据，实现者在选择具体的数据表示方式时，需检查和确认所选择的表示是否满足要求
- 抽象函数(Abstraction Function)：数据实现到数据抽象的映射规则
  - AF: C→A, AF(c) = …, C为‘数据实现’空间，A为‘数据抽象’空间
- AF_DensePoly(c)
  - {<**cof[i],** c.terms[i]>, <**deg[i]**, c.i>|0<=i<c.terms.length&&*c.terms[i]!=0*}
- AF_SparsePoly(c)
  - {<**cof[i]**,c.terms[i].cof>,<**deg[i]**,c.terms[i].deg>|0<=i<c.terms.length&&*c.terms[i].cof!=0*}
- 为什么需要定义抽象函数?
  - 分析和检查相应的表示能否满足类型规格的要求(即数据抽象规格)
  - 明确表示对象(即类中属性定义)的语义(即含义)
  - 使得类的方法在对表示对象进行访问时能够准确检查对象当前的状态

类型层次设计

- 子类可以对父类的实现数据进行扩充
- IntSet抽象函数
  - 数据抽象： int[] ia
  - 数据实现： Vector els
  - AF_IntSet(c) = {<**ia[i]**, c.els[i].intValue()>|0<=i<c.els.size()}
- MaxIntSet抽象函数
  - 数据抽象： (*int[] ia*), int max
  - 数据实现： (*Vector els*), int biggest
  - AF_MaxIntSet(c) = AF_IntSet(c)U{<**max**, biggest>}

类型层次设计

- 子类新增数据抽象和相应的数据实现
  - IntSet (int[] ia, Vector<Integer> els), ComplexSet(int[] ca, Vector<Integer> mels)
  - AF_IntSet(c) = {<**ia[i]**,c.els[i].intValue()>|0<=i<c.els.size}
  - AF_ComplexSet(c) = AF_IntSet(c)U{<**ca[i]**,c.mels[i].intValue()>|0<=i<c.mels.size}
- 通过抽象函数把类的数据实现映射到规格层次的数据抽象
  - 可以应用规格层次的invariant和constraint来检查数据实现是否满足要求

注意：JML比Liskov教材中的规格描述提供了更加严谨和细致的表达方式，相应抽象函数也有差异

类型层次下的规格关系

- 不变式应满足蕴含关系：*I_sub(c)* implies *I_super(c)*
  - I_IntSet(c): (ia!=null) && (\forall int i,j; 0<=i&&i<j&&j<ia.length;ia[i]!=ia[j])
  - I_MaxIntSet(c): I_IntSet(c) && (\forall int ele;this.isIn(ele);max>=ele) && ((ia.length>0)==> this.isIn(max))
- 子类新增的不变式一般**不应**对父类数据抽象施加新的约束
- 不变式满足蕴含关系是基础，同时也要注意数据的保护
  - 子类一般不应直接对父类所实现的数据进行修改，而是通过父类所定义的方法

讨论：不变式蕴含关系如何构造?

- IntSet (int[] ia, Vector<Integer> els), ComplexSet(int[] ca, Vector<Integer> mels)
- AF_IntSet(c) = {<**ia[i]**,c.els[i].intValue()>|0<=i<c.els.size}
- AF_ComplexSet(c) = AF_IntSet(c)U{<**ca[i]**,c.mels[i].intValue()>|0<=i<c.mels.size}
- I_IntSet(c): (ia!=null) && (\forall int i,j; 0<=i&&i<j&&j<ia.length;ia[i]!=ia[j])
- I_ComplexSet(c): I_IntSet(c) && ??
- 如果无法构造出满足蕴含关系的不变式
  - 不满足LSP
  - 继承关系建立的不合理

类型层次下的规格关系

- 子类与父类在不变式方面具有紧密的关联
- 子类的repOK应该调用父类的repOK来检查父类rep是否满足父类的不变式要求，并增加专属于子类的不变式检查逻辑

```
MaxIntSet{
    public boolean repOK(){
        boolean existed = false;
        if (!super.repOK())return false;
        for(int i=0;i<size();i++){
            if(biggest < getAt(i)) return false;
            if(biggest == getAt(i)) existed = true;
        }
        if(i==0) return true;  //must be an empty set
        return existed;
    }
}
```

层次化职责划分
类规格
类实现
状态有效性检查

类型层次下的规格关系

- 如果一个方法未被重写，则其正确性只由父类保证
- 子类新增方法的正确性不会影响父类的正确性
  - 前提：父类隐藏了所有数据实现
- 被子类重写的方法
  - LSP要求对父类型对象的方法**调用**可以**替换**为对子类型对象的相应调用
    - 调用前：Requires_super(f) ***implies*** Requires_sub(f)
    - 调用后：(Requires_super(f) && Ensures_sub(f)) ***implies*** Ensures_super(f)
  - 如果调用者违背父类型方法规格，但满足子类型方法规格
    - 父类型对象方法无法有效处理
    - 但子类型方法可能进行处理：Requires_sub(f) ***implies*** Ensures_sub(f)

类型层次下的规格关系

- 子类重写方法可以**减弱**父类方法规定的Requires,或者**加强**父类方法规定的Ensures
  - 对比分析IntSet的insert方法和MaxIntSet的insert方法
    - Requires: IntSet.insert(x) ---(none, i.e. true); MaxIntSet.insert(x)---(?)
    - Ensures: IntSet.insert(x) --- (this.isIn(x)); MaxIntSet.insert(x) ---(?)

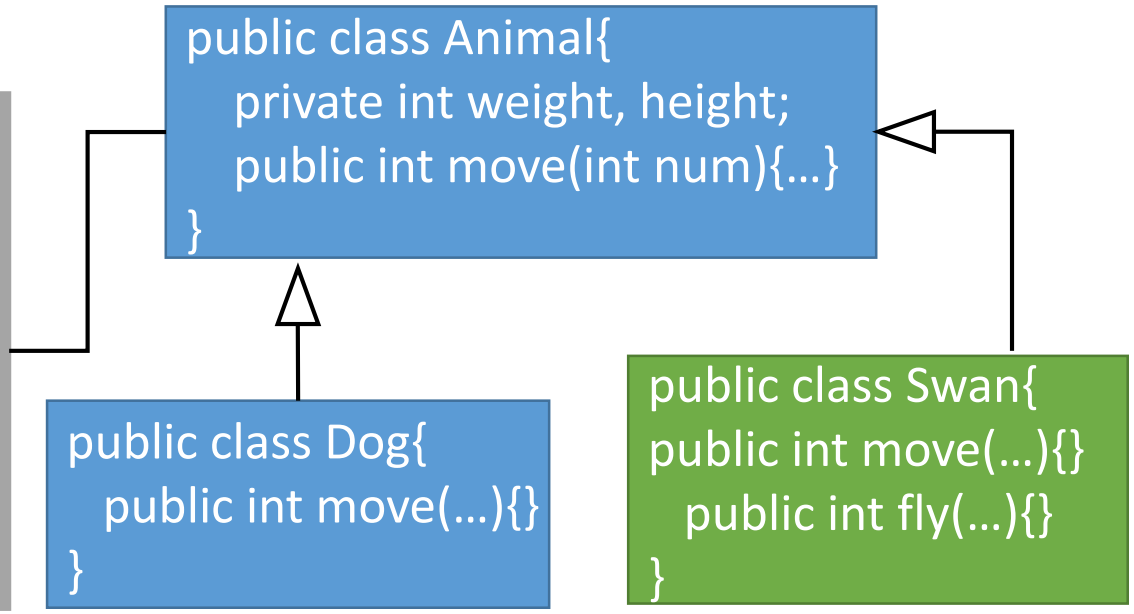
```
this.isIn(x) && (x>\old(max)==>(max == x))
```

- 子类重写方法的副作用范围可以不同，但不能影响父类方法的副作用范围规定
  - IntSet::insert, assignable this
  - MaxIntSet::insert, assignable this

类型层次的规格概括能力

- 在设计类型层次时，要考虑未来的扩展能力
  - 扩展可能发生在未来，且由别人来完成
- 父类型对未来子类型的概括能力是保持架构稳定的关键
  - 如果子类型扩展导致不满足LSP，就可能会导致user code修改
  - user code的修改代价可能非常高！

```
public class Some{
    public void action (Animal a){
        int distance = a.move(x);
        if(a instanceof Swan){
            if((Swan)a.isFly())distance = (Swan)a.fly(x);
        …
    }
}
```



类型层次下的正确性检查问题

- 类正确：实现**满足**其**规格**
  - 对象状态保持有效：所有方法都不会导致不变式和修改约束失效
  - 方法实现满足其规格要求
- 类型层次下的正确性检查
  - 检查父类的正确性
  - 检查子类对象是否始终有效
  - 检查子类新增方法是否正确
  - **检查子类重写方法是否正确**

类型层次下的正确性检查问题

- 子类对象的有效性
  - 构造器不能导致**父类不变式**失效
  - 构造器保证**子类不变式**有效
  - 子类重写方法
    - 与父类相应方法是否满足LSP原则
  - 针对所有子类更新方法
    - 如果更新了**父类数据实现**，需要检查是否导致**父类不变式**无效
    - 如果更新了**子类数据实现**，需要检查是否导致**子类不变式**无效
  - 如果子类不能直接访问父类数据实现，则简单许多
    - 子类只能调用父类方法来更新父类rep---->父类所有方法都不会导致repOK为假！

讨论：规格满足蕴含，但实现正确吗？

```
public class Rectangle {
   //@public model long width, height;
   //@invariant width > 0 && height > 0;
   private long width;
   private long height;
   public void setWidth(long width) {
      this.width = width;
   }
   public long getWidth() {
      return this.width;
   }
   public void setHeight(long height) {
      this.height = height;
   }
   public long getHeight() {
      return this.height;
   }
}
```

```
public class Square extends Rectangle {
   //@invariant width == height
   public void setWidth(long width) {
      super.setHeight(width);
      super.setWidth(width);
   }

   public void setHeight(long height) {
      this.setWidth(height);
   }
}
```

```
public class Test
{
   public void resize(Rectangle r)
   {
      while (r.getHeight() <= r.getWidth() )
         r.setHeight(r.getHeight() + 1);
   }
}
```

请思考：

1. **Test类的resize方法存在什么潜在的问题？**
2. **如果有问题，原因是什么？**
3. **从规格角度来分析原因**

集合元素的迭代访问

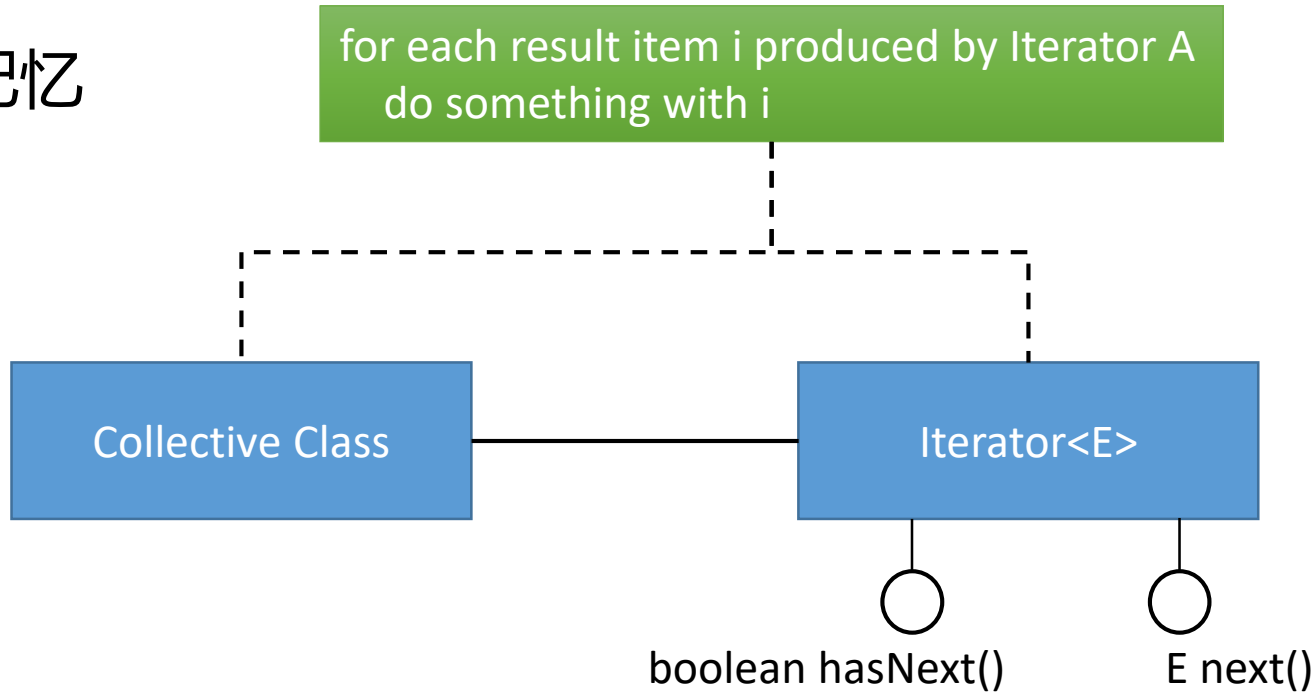
- 使用类来封装和管理数据
- 集合是一类重要的数据封装和管理类型
  - 具有动态性
  - 实现方式多样化：Array, Vector, List, HashSet, Tree, Graph,…
- 集合类型的操作
  - 查看操作：元素个数、是否为空、是否存在某个元素、查找某个元素…
  - 更新操作：排序、插入、删除…
- User的期望
  - 管理集合元素
  - 遍历集合元素进行特定的处理(User的业务逻辑)

集合元素的迭代访问

- 方法1：无保护访问
  - 直接把一个对象的内部rep返回给用户进行操作
  - 无法保证invariant始终得到满足
- 方法2：有保护的克隆访问
  - 直接返回对象rep的克隆，对象数据得到保护
  - 集合规模大时?
- 方法3：不规范的有保护访问
  - 循环调用查找操作来获得相关的元素：for(i=0;i<o.size();i++){…;o.get(i)…}
  - 讨论：每个集合类型都要提供类似的方法，是否可以规范化?
- 方法4：规范的有保护访问(迭代访问机制)
  - 规范一致的User Code

集合元素的迭代访问

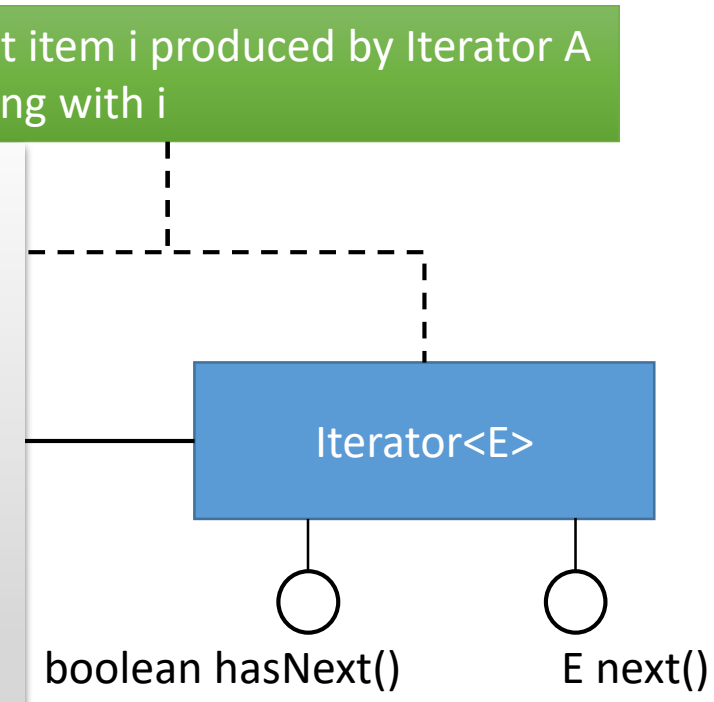
- 迭代访问机制
  - 通过模板化的迭代器接口规范元素访问
    - hasNext()判断是否还有元素来迭代访问
    - next()返回下一个元素
  - 一次只返回一个元素，且有记忆



集合元素的迭代访问

- 迭代访问机制
  - 通过模板化的迭代器接口规范元素访问
    - hasNext()判断是否还有元素来迭代访问
    - next()返回下一个元素
  - 一次只返回一个元素，且有记忆

```
public interface Iterator<E>{
     //@ public model int index;
     //@ public model E[] els;
     //@ ensures \result==(index<els.length)
     Public /*@pure@*/ boolean hasNext ( );
     /*@ normal_behavior
       @ requires index < els.length;
       @ assignable this;
       @ ensures \result == els[\old(index)] && index == \old(index) + 1;
       @ also
       @ exceptional_behavior
       @ signals (NoSuchElementException e) index >= els.length;@*/
     public E next ( ) throws NoSuchElementException;
}
```



示例：Poly and IntSet

```
public class Poly {
   /*@ ensures (\result != null) && (\result.els.length == cof.length) &&
     @ (\forall int i; 0<=i&&i<cof.length;\result.els[i].intValue()==cof[i])
     @ \result.index == 0;
     @*/
    public Iterator<Integer> coefs(){…}
}

public class IntSet {
 /*@ ensures (\result != null) && (\result.els.length == ia.length) &&
   @ (\forall int i; 0<=i&&i<ia.length;\result.els[i].intValue() == ia[i]) &&
   @ \result.index == 0;
   @*/
   public Iterator<integer> elements() {…}
}
```

集合类型提供一个**迭代**方法，返回**生成器对象**

- 可以定义多个迭代器方法来提供多个生成器
- 生成器实现Iterator接口

生成器的实现模板

```
public class Poly{
   //private int [] trms; //cof
   //private int [] degs; //degree
   public Iterator coefs() {return new PolyGen(this);}
   private static class PolyGen implements Iterator {
        private Poly p;          // the Poly being iterated
        private int n;           // the next term to iterate
        PolyGen (Poly it){  p = it;  n= 0;   }
        public boolean hasNext() {return n< p.trms.length;}
        public Object next () throws NoSuchElementException {
            if(n<p.trms.length){ return new Integer(p.trms[n]); n++; }
            else throw new NoSuchElementException("Poly.terms");
    } // end PolyGen
}
```

迭代访问的状态变化

Poly p … // p = 2 + 3 x^2+ 4 x^5

Iterator itr = p.coefs(); // itr = ([2,3,4],0)

itr.hasNext() // return true, itr = ([2,3,4],0)

itr.next() // return 2, itr = ([2,3,4],1)

itr.next() // return 3, itr = ([2,3,4],2)

itr.hasNext() // return true, itr = ([2,3,4],2)

itr.next() // return 4, itr = ([2,3,4],3)

itr.hasNext() // return false, itr = ([2,3,4],3)

itr.next() // throw NoSuchElementException, itr = ([2,3,4],3)

生成器的抽象函数和不变式

- 生成器是一个数据抽象，提供了hasNext和next操作来访问其管理的数据
  - Poly类生成器的表示对象rep
    - private Poly p; //the Poly being iterated ➔但仅能访问其每一项的系数
    - private int n; //the next element (index) to iterate
- Poly类生成器的抽象函数
  - AF(itr_c) = {<els[i].intValue(), itr_c.p.cof[i]>|0<=i<itr_c.p.cof.length} U {<index,itr_c.n>}

课堂练习：TwoWayInterator

```
public interface TwoWayIterator<E>{
     //@ public model int findex, bindex;
     //@ public model E[] els;
     //@ ensures \result==(findex<els.length)
     public /*@pure@*/ boolean hasNext ( );
     /*@ normal_behavior
       @ requires findex < els.length;
       @ assignable this;
       @ ensures \result == els[\old(findex)] && findex==\old(findex) + 1;
       @ also
       @ exceptional_behavior
       @ signals (NoSuchElementException e) findex >= els.length;
       @ */
     public E next ( ) throws NoSuchElementException;
     public /*@pure@*/ boolean hasPrevious ( );
     public E prev ( ) throws NoSuchElementException;
}
```

- 当前的部分规格是否有错?
- 请修改和补充其规格

如果你想支持remove操作

void remove() //optional operation
Removes from the underlying collection the last element returned by this iterator. This method can be called only once per call to next(). **The behavior of an iterator is unspecified if the underlying collection is modified *while the iteration is in progress* in any way other than by calling this method.**
//如果通过调用remove方法来移出相关元素时，有其它任何调用也在改变集合的状态，则迭代器的行为未定义。
Throws:
UnsupportedOperationException - if the *remove* operation is not supported by this iterator
IllegalStateException - if the *next* method has not yet been called, or the *remove* method has already been called after the last call to the *next* method

itr = ({1,2,3,5,10},0)
set = {1,2,3,5,10}
itr.next() ----1
itr = ({1,2,3,5,10},1)
set = {1,2,3,5,10}
itr.remove ()
set = {1,2,5,10}
itr = ({2,3,5,10},0)

如果你想支持remove操作

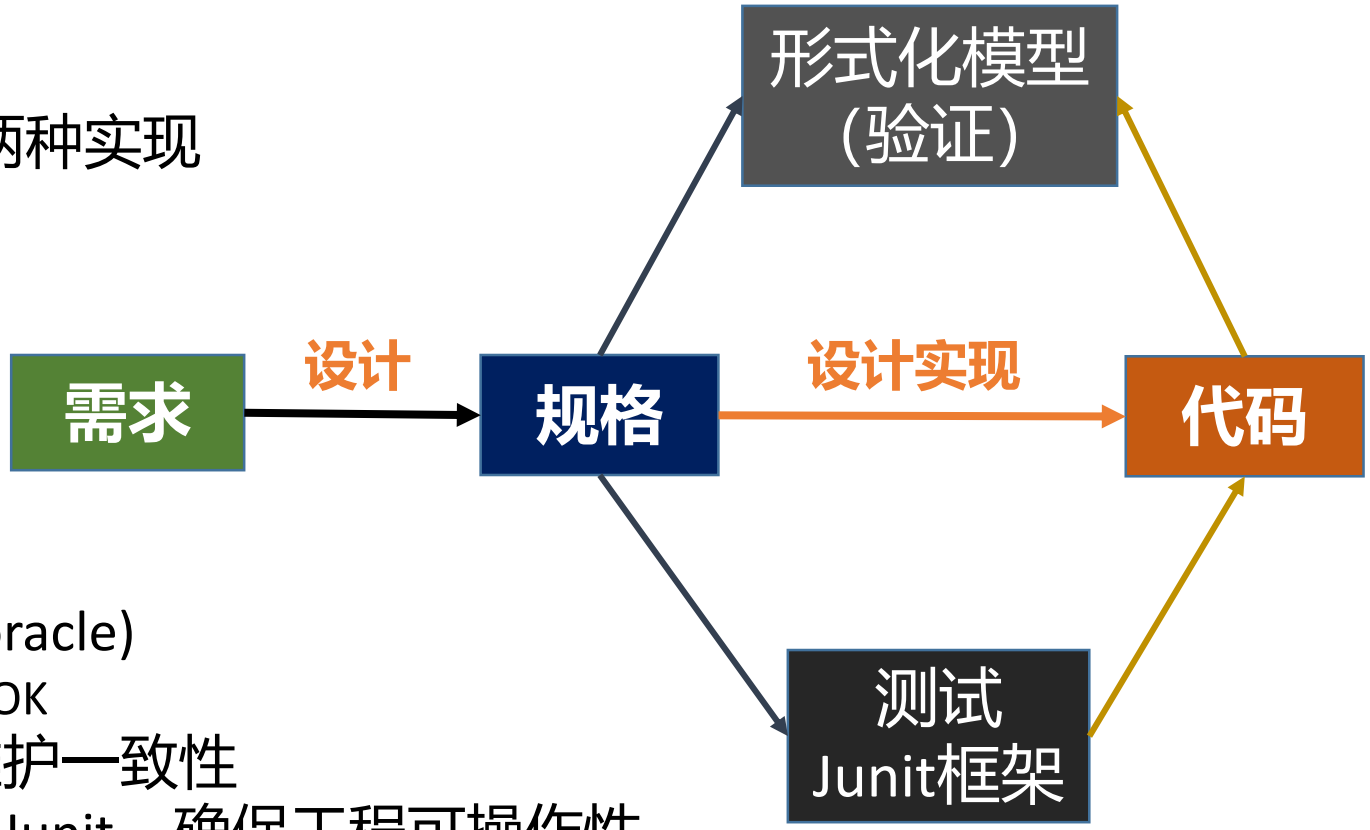
- Step 1: 确保remove操作时集合类对象处于保护状态
  - lock
- Step 2: 确保集合类提供remove指定对象的方法
  - 如PolyGen中的Poly对象必须提供remove(int d)方法
- Step 3: 确保生成器记录next,remove的调用状态
  - next() ----- nextStatus = true; removeStatus = false;
  - remove() -----nextStatus = false; removeStatus = true;
- Step 4: 记录next生成的对象
  - next() ---- justGenerated = …; return justGenerated;
- Step 5: 确保生成器对remove操作进行控制
  - if(nextStatus == false || removeStatus == true) return;
  - Lock(collection);
  - Collection.remove(justGenerated)

如何保证软件代码实现满足需求

- 需求描述的严谨性是基础
  - 将自然语言的需求描述通过形式化的规格描述表达
  - 解决规范性和无二义性问题，使用者、设计者和实现者的共同认知
- 从设计规格到代码：如何检查代码实现是否满足规格
  - 完全的定理证明
  - 在JML基础上的逻辑验证
  - 基于执行的测试验证
- 测试验证：工程之道
  - 功能测试：要求覆盖所有功能场景，依据需求设计测试用例
  - 设计测试：要求覆盖规格中所定义的边界条件组合
  - JML规格为测试提供了独立于代码的设计依据和检查依据

测试如何应对规格和代码的变化

- 规格保持，代码变化
  - 更有效率的实现代码
  - 适应新的场景，如多项式的两种实现
- 规格调整，代码势必调整
- 形式化验证的局限
  - 难以处理复杂的逻辑组合
  - 应用成本高
- 双剑合璧： JML+Junit
  - 规格直接提供测试判定(test oracle)
    - repOK、RequriesOK、EnsuresOK
  - JML→JUnit，工具链支持可维护一致性
  - 实现代码的变化不影响JML和Junit，确保工程可操作性



正确性的推理验证

- 一般的，任何一个程序模块或者一段代码s，设其前置条件和后置条件分别为*P* 和*Q*
- 按照Hoare正确性证明，可以表示为Hoare三元组断言：
  - {*P*} *s* {*Q*}
  - 在s执行前如果程序的状态满足P，则执行后满足Q
- 如果s在任意情况下的执行都能确保上述断言为真，那么s就称为是正确的

针对语句类型的推理验证规则

Statement Type	**Inference Rule**
1. *Assignment* t = s ;	$true \vdash \{Q[\text{t} \leftarrow \text{s}]\}\ s\ \{Q\}$
2. *Block* (sequence) s_1 ; s_2	$\{P\}\ s_1\ \{R\}, \{R\}\ s_2\ \{Q\} \vdash \{P\}\ s_1\ ;\ s_2\ \{Q\}$
3. *Conditional* if (test) then s_1 else s_2	$\{\text{test} \wedge P\}\ s_1\ \{Q\}$, $\{\neg\text{test} \wedge P\}\ s_2\ \{Q\} \vdash \{P\}\ s\ \{Q\}$
4. *Loop* while (test) s	$\{\text{test} \wedge R\}\ s\ \{R\} \vdash \{R\}$ while (test) $s\ \{\neg\text{test} \wedge R\}$
5. *Rule of consequence*	$P \Rightarrow P', \{P'\}\ s\ \{Q'\}, Q' \Rightarrow Q \vdash \{P\}\ s\ \{Q\}$

Note: the form $p_1, p_2 \vdash q$ reads, "if p_1 and p_2 are valid then q is valid."

针对OO程序的形式化验证

- 正确性证明的基本单位是类和对象，方法是其组成部分
- 程序运行阶段，所有堆中对象可视为活对象(active object)，其状态由其所有属性的取值联合决定
- 对象方法的*P*和*Q*是关于其状态和输入参数的谓词逻辑表达
- OO程序的规格化设计与开发所用语言
  - *Specifications* : JML
  - *Design*: UML + JML
  - *Coding*: Java + JML
  - *Verification*: JML

OO程序的正确性内涵

- 三个层次
  - 方法层次:前置条件P、后置条件Q、循环不变式R (loop invariants)
  - 类层次: 类不变式
  - 系统层次: 关于系统状态的不变式

- P: $\{n \geq 1\}$

```
int Factorial (int n) {
  int f = 1;
  int i = 1;
```

- **R: {1 ≤ i ∧ i ≤ n ∧ *f == i!*}**

```
  while (i < n) {
    i = i + 1;
    f = f * i;
  }
  return f;
}
```

- Q: $\{f == n!\}$

基于JML的表达

```
/*@ requires 1 <= n ;  //P
          ensures \result == (\product int i; 1<=i && i<=n; i) ;   //Q
   @*/
   static int Factorial (int n) {
         int f = 1;
         int i = 1;
   /*@ loop_invariant i <= n && f == (\product int j; 1 <= j && j <= i; j);  @*/          //R
         while (i < n) {
              i = i + 1;
              f = f * i;
         }
         return f;
   }
```

JML相关工具

- 编译工具 (jmlc)，基于 javac
  - 语法和类型检查，能够针对所有JML断言和Java代码生成字节码
- 静态检查工具， ESC/Java2，openjml
- 运行时断言检查 (jmlrac, openjml)，对java进程进行了扩展
  - 在方法调用时动态检查P的满足情况
  - 在方法退出时动态检查Q的满足情况
  - 在循环代码执行时动态R的满足情况
  - 如果P,Q,R有不满足情况则抛出Exception
- 测试工具jmlunit
- 文档化工具jmldoc
- 辅助证明推理工具 (Daikon)

作业

- 最终需要实现一个社交关系模拟系统。可以通过各类输入指令来进行数据的增删查改等交互
  - 通过继承官方提供的接口Person 、 Network 和Group ， 来实现自己的Person 、 Network 和Group 类
    - 注意抽象数据有扩展
  - 实现 Message,EmojiMessage,NoticeMessage 和 RedEnvelopeMessage接口
  - 阅读异常行为描述，实现自己的异常类
  - 新增关于message的操作指令
- 注意系统设计，规格是针对具体的类的实现，但整个系统的效率需要设计去保证