Verification of Java's AbstractCollection class in Isabelle

Marieke Huisman

INRIA Sophia-Antipolis
France
Marieke.Huisman@sophia.inria.fr

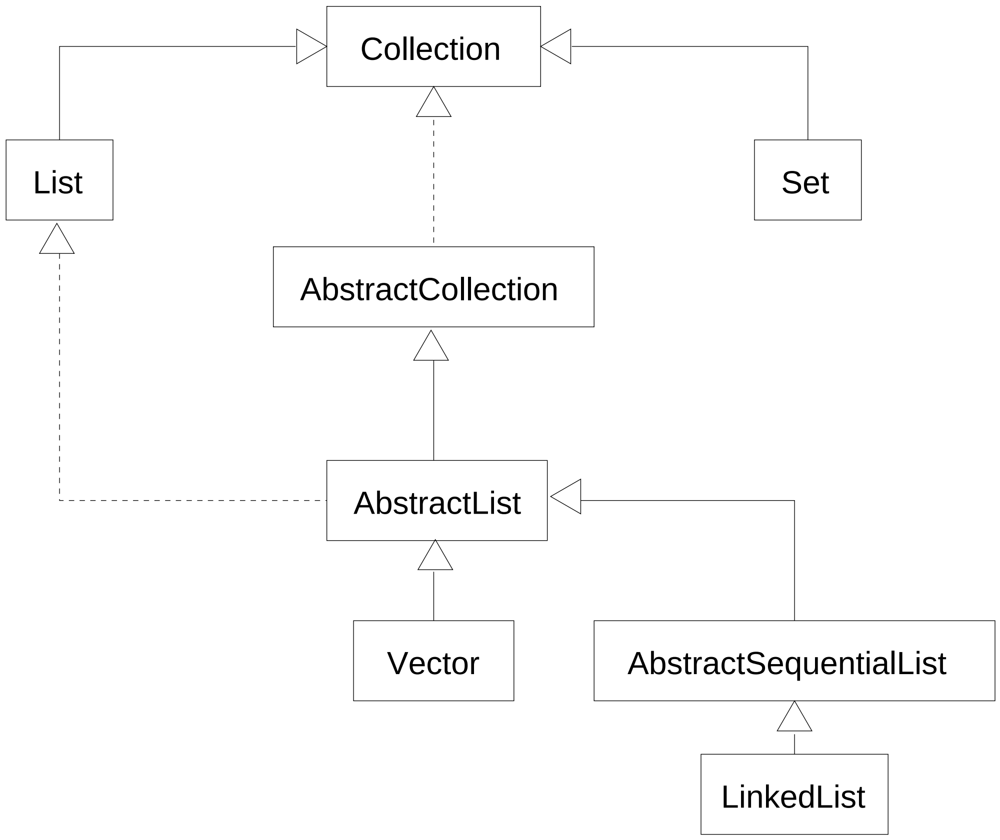
A Case Study within the LOOP project

- LOOP tool translates Java programs into language of theorem prover
- Output theorem provers: PVS and Isabelle
- Case Studies
  - Show feasibility of program verification
  - Improve automatisation
  - Earlier: class invariant of Vector
  - Today: functional behaviour of Collection

Contents

The Collection hierarchy

The Collection hierarchy



AbstractCollection

- `AbstractCollection` implements `Collection`
- But... leaves several methods without implementation
- Other methods are implemented in terms of these methods

Methods without implementation

- Abstract methods

```
public abstract int size();
```

- Optional methods

```
public boolean add(Object o) {
  throw new UnsupportedOperationException();
}
```

Methods without implementation

- Abstract methods have to be defined in subclasses

```
public abstract int size();
```

- Optional methods might be overridden

```
public boolean add(Object o) {
  throw new UnsupportedOperationException();
}
```

The rôle of Iterator

- Every collection contains a method declaration

```
Iterator iterator();
```

- `Iterator` interface

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();}
```

- In practice, every implementation of `Collection` contains inner class, implementing `Iterator`

Verification techniques

The verification process

- Formal specification of `Iterator` and `Collection` in JML
- Translation into Isabelle
  - Java classes translated by LOOP tool
  - JML annotations translated by hand
- Verification of the concrete methods in class `AbstractCollection`
- Assuming other methods satisfy their specification

Behavioural subtypes and verification

- Proof obligations for each method implementation:
  - satisfies own specification
  - satisfies specifications of method in superclass
- Assumption for each method call: it satisfies the specification of the method in the declared class

The Java Modeling Language (JML)

- Annotation language for Java, developed by Leavens et.al., Iowa State University
- Pre- and post-conditions, frame conditions and invariants written as special comments
- Uses Java-like notation
- Annotations are side-effect-free Java expressions + some extra keywords (`\exists`, `\forall`, `\old(−)`, `\result`...)
- LOOP compiler currently extended to generate proof obligations, based on annotations

Specification-only variables

- Fields often private
- Thus: it does not make sense to use them in public specifications
- Solution: model variables
- Model variables (or abstract variables) represent a set of concrete variables
- Model variables provide a way to hide concrete implementations from the specifier

The specification of Collection

Writing the specifications

- Identify model variables to describe internal state
- Restrict internal state with class invariants
- Specify pre- and post-conditions

From the documentation of Collection

```
Bags or multisets (unordered collections that may contain
duplicate elements) should implement this interface directly.
[...]
Some collections allow duplicate elements and others do not.
```

From the documentation of Collection

`Bags or multisets (unordered collections that may contain duplicate elements) should implement this interface directly.`
`[...]`
`Some collections allow duplicate elements and others do not.`

- `contents` of type multiset
- `allowDoubles` of type boolean

From the documentation of Collection (2)

The documentation of `add`

```
Ensures that this collection contains the specified element
(optional operation).
[ ...]
Collections that support this operation may place limitations
on what elements may be added to this collection.
```

From the documentation of Collection (2)

The documentation of `add`

```
Ensures that this collection contains the specified element
(optional operation).
[ ...]
Collections that support this operation may place limitations
on what elements may be added to this collection.
```

- `addDefined` of type boolean
- `storable` of type `Object` $\longrightarrow$ boolean

From the documentation of Collection (3)

The documentation of `remove`

`Removes a single instance of the specified element from this collection, if it is present (optional operation).`

From the documentation of Collection (3)

The documentation of `remove`

```
Removes a single instance of the specified element from this
collection, if it is present (optional operation).
```

- `removeDefined` of type boolean

Summary: the model variables for Collection

- `contents` of type multiset
- `allowDoubles` of type boolean
- `addDefined` of type boolean
- `storable` of type `Object` $\longrightarrow$ boolean
- `removeDefined` of type boolean

Invariant of `Collection`

- `contents` is always a finite bag
- if `allowDoubles` is false, every element occurs at most once in the collection

From the documentation of Iterator

```
An iterator over a collection.
```

From the specification of remove

```
Removes from the underlying collection the last element
returned by the iterator (optional operation).  This method
can be called only once per call to next.
[...]
@exception 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.
```

Model variables for Iterator

The following model variables are needed to specify `Iterator`:

- `contents` of type multiset
- `removeDefined` of type boolean
- `underlyingCollection` of type `Collection`
- `lastElement` of type `Object`
- `removeAllowed` of type boolean

Invariant of `Iterator`

- `contents` is always a finite bag

The methods in Iterator

```
/*@ normal_behavior
  @   ensures \result == (contents != {#});
 @*/
public boolean hasNext();
```

The methods in Iterator (2)

```
/*@ normal_behavior
  @     requires contents != {#};
  @   modifiable contents, lastElement, removeAllowed;
  @      ensures contents ==  \old(contents) -  {#\result#} &&
  @              \old(contents.elem(\result) &&
  @              removeAllowed &&
  @              lastElement == \result;
  @*/
public Object next();
```

The methods in Iterator (3)

```
/*@ normal_behavior
  @   requires removeDefined && removeAllowed;
  @ modifiable underlyingCollection.contents, removeAllowed;
  @    ensures !removeAllowed &&
  @            underlyingCollection.contents ==
  @             \old(underlyingCollection.contents)
  @               -
  @               {#lastElement#};
  @*/
public void remove();
```

Specification of iterator in Collection

```
/*@ normal_behavior
  @   ensures \result instanceof Iterator &&
  @           \result.contents == contents &&
  @           \result.removeDefined == removeDefined &&
  @           \result.underlyingCollection == this &&
  @           !\result.removeAllowed;
  @*/
public Iterator iterator();
```

Implicitly, the result satisfies the specification of `Iterator`, because it is an instance of `Iterator`.

The other constraints show how `Iterator` should be initialised.

Specification of add in Collection

- Ensures the occurrence of an object
- What if equal object already occurs?
- Object equality differs from pointer equality
- Abbreviation `occurs` introduced (so-called model method) to deal with this.

Specification of add in Collection (2)

```
/*@ normal_behavior
  @     requires addDefined && storable(o);
  @   modifiable contents;
  @      ensures \result == (contents !=
  @                            \old(contents)) &&
  @                (!allowDoubles &&
  @                 \old(contents.occurs(o))) ?
  @                contents == \old(contents) :
  @                contents == \old(contents) + {#o#};
  @*/
public boolean add(Object o);
```

What is the result of `remove_one_element`?

```
class RemoveCollectionFromCollection {
  Vector w;
  boolean remove_one_element () {
    Vector v = new Vector ();
    Object o = new Object ();
    v.add(o);
    v.add(v);
    w = (Vector)v.clone();
    boolean first_time = v.contains(w);
    v.remove(o);
    boolean second_time = v.contains(w);
    return (first_time == second_time);
  }}
```

What is the result of `remove_one_element`? False!

```
class RemoveCollectionFromCollection {
  Vector w;
  boolean remove_one_element () {
    Vector v = new Vector ();
    Object o = new Object ();
    v.add(o);
    v.add(v);
    w = (Vector)v.clone();
    boolean first_time = v.contains(w);
    v.remove(o);
    boolean second_time = v.contains(w);
    return (first_time == second_time);
  }}
```

Specification of remove in Collection (2)

```
/*@ normal_behavior
  @   requires removeDefined &&
  @            !\reach(contents).has(this);
  @ modifiable contents;
  @    ensures
  @      \result == (contents != \old(contents)) &&
  @      (\forall (Object x) x != this ==>
  @       contents.count_occurs(x) ==
  @       (o == null && x == null) | (o != null && o.equals(x)) ?
  @       min (\old(contents.count_occurs(x) - 1), 0) :
  @       \old(contents.count_occurs(x)));
  @*/
public boolean remove(Object o);
```

The verification of AbstractCollection

Verification process

- Manual translation of specifications of `Collection` and `Iterator`
- Automatic translation (by LOOP compiler) of implementations
- Every method implementation is shown to satisfy the specification of this method in `Collection` (assuming that all other methods satisfy their implementation)

Translating the JML specifications into Isabelle specifications

- Translation by hand (error prone)
- Translation makes many details explicit
  - Remembering the pre-state
  - 'Instance of' implies that object satisfies the class specification of declared class
  - Parameters and local variable satisfy the specification of the declared class
  - Variables not in modifiable clause are unchanged

Verification techniques

- All specifications are phrased as Hoare triples $\Rightarrow$ use of Hoare logic for Java
- Experiment: letting Isabelle select the appropriate Hoare logic rule
  - Rule selection based on statement (sometimes backtracking needed)
  - Repetition could reduce the goal to the basic proof obligation (an abstract method, a while statement...)
  - But... reasoning with specifications required more user-interaction
- Development of rewrite rules directly based on semantics
  Rewriting in PVS different then in Isabelle $\Rightarrow$ requires different rewrite rules

Conclusions

- Impressive case study, showing feasibility to reason about actually used Java programs
- Modular verification techniques: class verified in isolation, using specifications only
- Verification revealed many subtle errors in specification $\Rightarrow$ specification can not do without verification
- These subtleties were not clear from informal documentation and code only