Konzepte objektorientierter Programmierung
– Lecture 11 –

Prof. Dr. Peter Müller

Chair of Programming Methodology

Herbstsemester 2008

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Meeting the Requirements

Cooperating Program Parts with Well-Defined Interfaces

→ Objects (data + code)
→ Interfaces
→ Encapsulation

Classification and Specialization

→ Classification, subtyping
→ Polymorphism
→ Substitution principle

Inherently Concurrent Execution Model

→ Active objects
→ Message passing

Correctness

→ Interfaces
→ Encapsulation
→ Simple, powerful concepts

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Agenda for Today

11. Interface Specifications

11.1 Frame Properties

11.2 Invariants and Callbacks

11.3 Invariants of Object Structures

Objectives

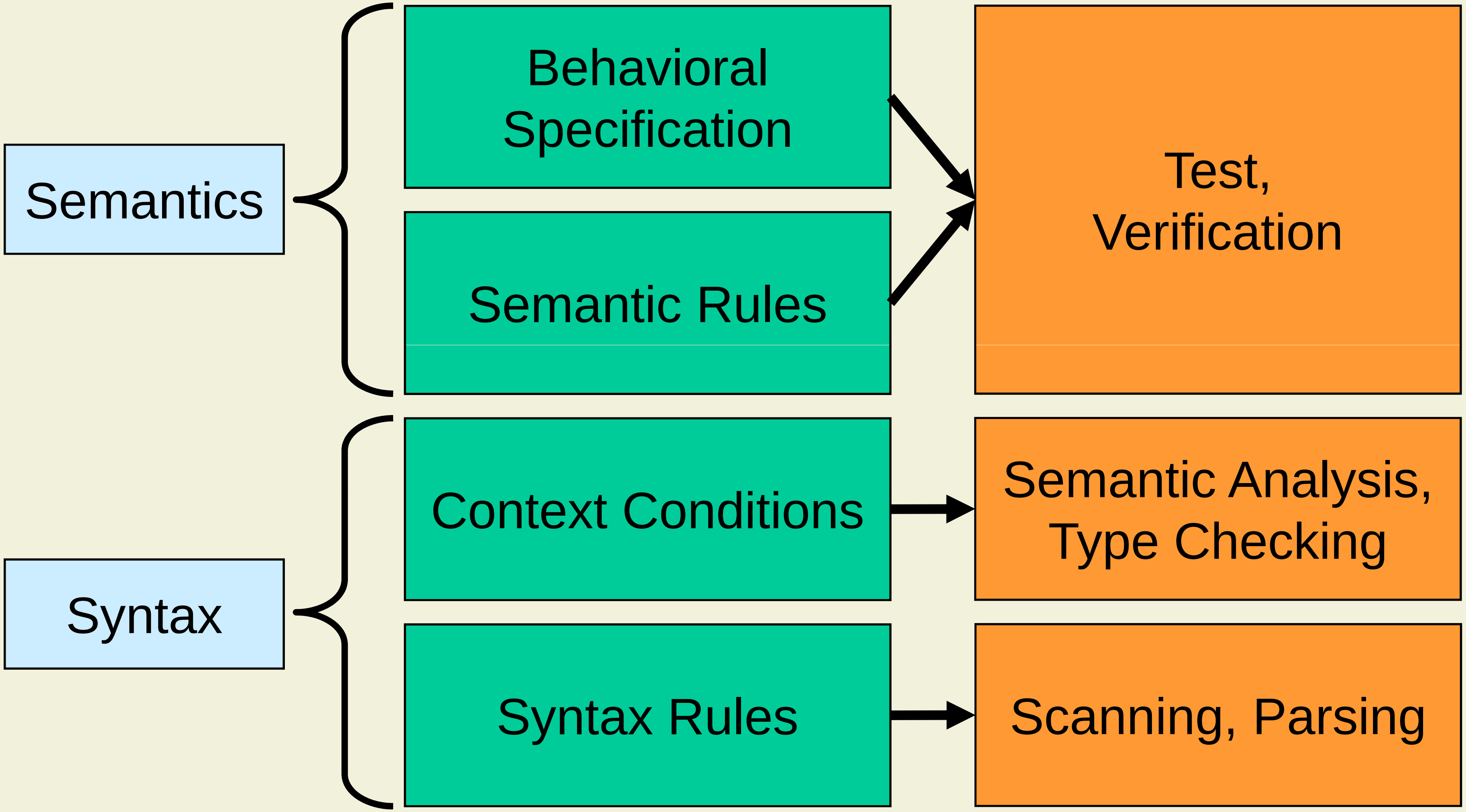
- Understanding subtle correctness conditions

Correctness

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;
  …

  public void insert( int i ) {
    for ( int j = 0; j < next; j-- )
      if array[ j ] == i then return true;
    return false;
  }
}
```

- Behavioral Specification
- Semantic Rules
- Context Conditions
- Syntax Rules

Aspects of Correctness



Test and Verification

Test

- Objective
  - Detect bugs
- Examples
  - White box test
  - Black box test
- Problems
  - Successful test does not guarantee correctness

Verification

- Objective
  - Prove correctness
- Examples
  - Formal verification based on a logic
  - Symbolic execution
- Problems
  - Expensive
  - Formal specification of behavior is required

11. Interface Specifications

11.1 Frame Properties

11.2 Invariants and Callbacks

11.3 Invariants of Object Structures

Pre-Post Specifications

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;

  public boolean has( int i ) {
    for ( int j = 0; j < next; j++ )
      if ( array[ j ] == i ) return true;
    return false;
  }
  private void resize( ) {
    int[ ] tmp = new int[ array.length + 10 ];
    System.arraycopy( array, 0, tmp, 0,
      array.length );
    array = tmp;
  }
```

```
  // requires !has( i )
  // ensures has( old( i ) )
  public void insert( int i ) {
    if ( next == array.length )
      resize( );
    array[ next ] = i;
    next++;
  }

  …
}
```

Frame Properties

```
// requires !has( i )
// ensures has( old( i ) )
public void insert( int i ) {
  array[ 0 ] = i;
  next = 1;
}
```

```
Set s = new ArraySet( );
s.insert( 42 );
s.insert( 1492 );
boolean b = s.has( 42 );
```

```
// requires !has( i )
// ensures has( old( i ) )  &&
//     ∀j: old( has( j ) ) ⇒ has( j )
public void insert( int i ) { … }
```

- Methods can have side-effects
- Frame properties describe **what is left unchanged** by a method execution

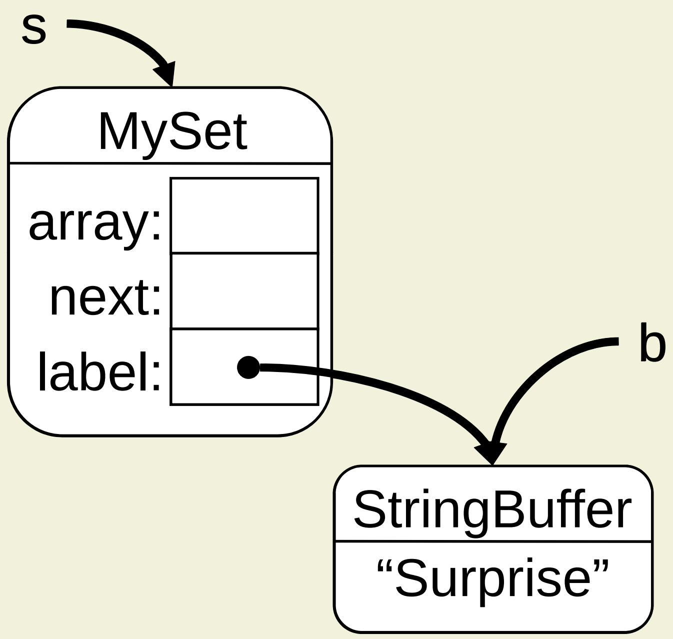
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Frame Properties (cont'd)

```
// requires !has( i )
// ensures has( old( i ) )  &&
//     ∀j: old( has( j ) ) ⇒ has( j )
public void insert( int i ) { … }
```

```
void foo( Set s, StringBuffer b ) {
  b.insert( 0, "Hello" );
  s.insert( 1492 );
  System.out.println( b );
}
```

```
class MySet extends ArraySet {
  StringBuffer label;
  public void insert( int i ) {
    label.insert( 0, "Surprise" );
    super.insert( i );
  }
}
```



Modifies Clauses

- In **modular programs**, not all locations that remain unchanged are known
- **Modifies clauses** specify which locations **may be modified** by a method
- Locations **not mentioned** in the modifies clause **must remain unchanged**

```
// requires !has( i )
// ensures has( old( i ) )  &&
//    ∀j: old( has( j ) ) ⇒ has( j )
// modifies array[ next ], next,
//                    array
public void insert( int i ) {
  if ( next == array.length )
    resize( );
  array[ next ] = i;
  next++;
}
```

Example Revisited

```
// requires !has( i )
// ensures has( old( i ) )  &&
//  ∀j: old( has( j ) ) ⇒ has( j )
// modifies array[ next ], next, array
public void insert( int i ) { … }
```

```
void foo( Set s, StringBuffer b ) {
  b.insert( 0, "Hello" );
  s.insert( 1492 );
  System.out.println( b );
}
```

```
class MySet extends ArraySet {
  StringBuffer label;
  public void insert( int i ) {
    label.insert( 0, "Surprise" );
    super.insert( i );
  }
}
```

- **Behavioral subtyping:** Subtype methods have to satisfy modifies clauses of overridden supertype methods

Semantics of Modifies-Clauses

- Each method m may **modify**
  - **Locations mentioned in its modifies clause**
  - **Locations of newly allocated objects**

- All other locations have to be left unchanged
  - Temporary modifications are possible

Modifies Clauses: Example

```
// modifies array[ next ], next,
//                 array
public void insert( int i ) {
  if ( next == array.length )
    resize( );
  array[ next ] = i;
  next++;
}
```

```
// modifies array
private void resize( ) {
    int[ ] tmp = new int[ array.length + 10 ];
    System.arraycopy( array, 0, tmp, 0,
                                        array.length );
    array = tmp;
}
```

- insert modifies directly
  - this.array[this.next], this.next
- insert modifies indirectly via invocation of resize
  - this.array
  - Locations of newly allocated array

Problems of Modifies Clauses

- Enumerating modifiable locations **violates information hiding**
- Enumerating modifiable locations **does not work for interfaces**

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;
  // modifies array[ next ], next, array
  public void insert( int i ) { … }
  …
}
```

```
interface Set {
  // modifies ??
  public void insert( int i );
}
```

Extended State Problem

- **Behavioral subtyping** requires subtype methods to satisfy modifies clauses of supertype methods

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;

  // modifies array[ next ], next, array
  public void insert( int i ) { … }
  …
}
```

- **But:** subtype methods must have the right to modify the **extended state**

```
class MaxSet extends ArraySet {
  private int max;

  public void insert( int i ) {
    if ( i > max ) max = i;
    super.insert( i );
}
```


ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstraction: Wildcards

- Frame properties have to be specified in an **abstract** way
- Wildcard * represents **all fields of an object** (independent of the declaration class)
- Works with information hiding, interfaces, and extended state

```
class Tuple {
  private int a, b;
  // modifies this.*
  public void setFirst( int i ) { a = i; }
}
```

```
class BackupTuple extends Tuple {
  private int ba, bb;
  // modifies this.*
  public void setFirst( int i ) {
    ba = a;
    a = i;
  }
}
```

Problem: Imprecision

- **Missing precision** can be compensated (partly) by **additional postconditions**
  - Private postconditions

```
class Tuple {
  private int a, b;

  // modifies this.*
  // private ensures b == old( b )
  public void setFirst( int i ) { a = i; }
}
```

Problem: Imprecision

- **Missing precision** can be compensated (partly) by **additional postconditions**
  - Private postconditions
  - Getter methods
- Subclasses have to **strengthen inherited specification**
  - Even if method is not overridden

```
class Tuple {
  private int a, b;
  // modifies this.*
  // ensures getB( ) == old( getB( ) )
  public void setFirst( int i ) { a = i; }
}
```

```
class BackupTuple extends Tuple {
  private int ba, bb;
  // ensures bb == old( bb )
  public void setFirst( int i )
    { ba = a; a = i; }
}
```

Strengthening Postconditions

```
class Tuple {
  private int a, b;

  // modifies this.*
  // ensures b == old( b )
  public void setFirst( int i )
    { a = i; m( ); }

  // modifies this.*
  // ensures b == old( b )
  public void m( ) { }
}
```

```
class BackupTuple extends Tuple {
  private int ba, bb;


  // ensures bb == old( bb )
  public void setFirst( int i )
    { ba = a; super.setFirst( i ); }



  public void m( ) { bb = bb + 1; }
}
```

- Calling supertype methods on subtype objects may still lead to imprecise frame properties
- Further reading: K.R.M. Leino: *Data groups: Specifying the modification of extended state*

Problem: Object Structures

- **Object structures** can be handled through ownership

- Right to modify o.* includes to modify all objects owned by o

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;
  // modifies array[ next ], next, array
  public void insert( int i ) { … }
  …
}
```

Problem: Object Structures

- **Object structures** can be handled through ownership
- Right to modify o.* includes to modify all objects owned by o

```
class ArraySet implements Set {
  rep private int[ ] array;
  private int next;

  // modifies this.*
  public void insert( int i ) { … }
  …
}
```

11. Interface Specifications

Object Invariants

- Object invariants describe **consistency criteria** for objects
- **Invariants** have to hold in all states, in which an object can be accessed by other objects

```
class ArraySet implements Set {
  private int[ ] array;
  private int next;

  // invariant array != null      &&
  //    0 <= next <= array.length

  public void insert( int i ) {
    if ( next == array.length )
      resize( );
    array[ next ] = i;
    next++;
  }
 …
}
```

Assuming Invariants

- Methods **assume the invariant of this** to hold in the prestate
- Methods **assume the invariants of all allocated objects** to hold in the prestate

```
  // requires !has( i )
  public void insert( int i ) {
    if ( next == array.length )
      resize( );
    array[ next ] = i;
    next++;
  }
```

```
class Client {
  ArraySet as;
  // requires !as.has( 5 )
  public void foo( ) {
    as.insert( 5 );
  }
}
```

Semantics of Invariants

- Invariants of all allocated objects have to **hold in pre- and poststates** of all method (and constructor) executions
  - but can be temporarily violated in between

```
class Redundant {
  private int a, b;
  // invariant a == b
  public void set( int v ) {
    // all invariants hold
    a = v;
    // invariant of this violated
    b = v;
    // all invariants hold
  }
}
```

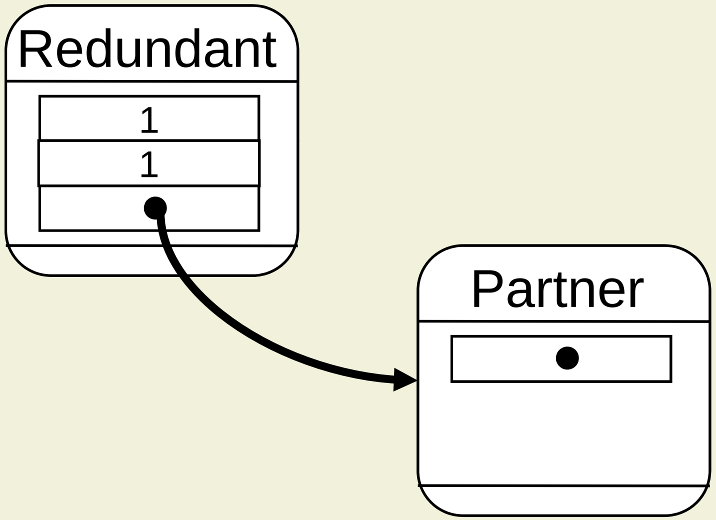
Checking Invariants

- For every method, **assume the invariants of all allocated objects in the prestate**
  - For constructors, exclude **this**
- For every method (and constructor), **prove** that all **invariants that are possibly violated** by the body hold (again) in the **poststate**
  - Vulnerable invariants depend on encapsulation
- Example: very strong encapsulation
  - Suppose invariants depend only on state of **this**
  - Suppose methods update only fields of **this**
  - Obligation: Prove invariant of **this** in poststate

Problem: Callbacks

```
class Redundant {
  private int a, b;
  private Partner p;
  // invariant a == b
  public void set( int v ) {
    a = v;
    p.foo( );
    b = v;
  }
  public int div( int v )
    { return v / ( a – b – 1 ); }
}
```

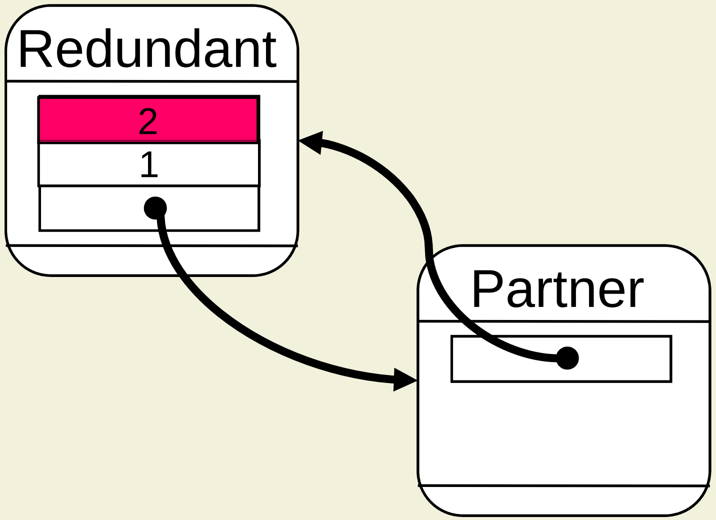
```
class Partner {

  public void foo( )
    { return 3; }
}
```



Problem: Callbacks

```
class Redundant {
  private int a, b;
  private Partner p;
  // invariant a == b
  public void set( int v ) {
    a = v;
    p.foo( );
    b = v;
  }
  public int div( int v )
    { return v / ( a – b – 1 ); }
}
```

```
class Partner {
  private Redundant r;
  public void foo( )
    { return r.div( 5 ); }
}
```



Common Variations

- Self-calls

```
class Redundant {
  private int a, b;
  // invariant a == b

  public void set( int v )
    { a = v; this.div( 5 ); b = v; }

  public int div( int v )
    { return v / ( a – b – 1 ); }
}
```

- Re-entrant monitors

```
class Redundant {
  private int a, b;
  // invariant a == b
  public synchronized
                    void set( int v )
    { a = v; this.div( 5 ); b = v; }
  public synchronized
                    int div( int v )
    { return v / ( a – b – 1 ); }
}
```

Dealing with Callbacks

- **Check invariant before** every method **call**
  - Overly restrictive: most methods do not call back (sqrt)
  - Too expensive for runtime checking
- **Detect** possible **callback** statically
  - Check invariant before call only if callback is possible
  - Difficult: requires knowledge of executed code including subclass methods
- **Specify** in each precondition which **invariants** the method actually **requires**
  - Check required invariants before method call
  - High documentation overhead

Example: Version 1

```
class Redundant {
  private int a, b;
  private Partner p;
  // invariant a == b
  public void set( int v ) {
    a = v;
    p.foo( );
    b = v;
  }
  public int div( int v )
    { return v / ( a – b – 1 ); }
}
```

Check fails

Unclear whether callback is possible

```
class Partner {

  public int foo( )
    { return 3; }
}
```

Requires no invariants

- Solution 1 reports error
  - Invariant does not hold
- Solution 2 reports error
  - Subclass of Partner might introduce callback
- Solution 3 succeeds

Example: Version 2

```
class Redundant {
  private int a, b;
  private Partner p;
  // invariant a == b
  public void set( int v ) {
    a = v;
    p.foo( );
    b = v;
  }
  public int div( int v )
    { return v / ( a – b – 1 ); }
}
```

Check fails

Callback possible

```
class Partner {
  private Redundant ;
  public void foo( )
    { return r.div( 5 ); }
}
```

Requires invariant of **this** and r

- Solution 1 reports error
  - Invariant does not hold
- Solution 2 reports error
  - Callback depends on aliasing
- Solution 3 reports error

11. Interface Specifications

Repetition: Invariants and Aliasing

```
class ArrayList {
  private int[ ] array;
  private int next;

  // invariant array != null        &&
  //    0<=next<=array.length   &&
  //    ∀i.0<=i<next: array[ i ] >= 0

  public void add( int i )    { … }
  public void addElems( int[ ] ia )
   { array = ia; next = ia.length; }

  …
}
```

```
int foo( ArrayList list ) {
  // invariant of list holds
  int[ ] ia = new int[ 3 ];
  list.addElems( ia );
  // invariant of list holds
  ia[ 0 ] = -1;
  // invariant of list violated
}
```

- Aliases can be used to by-pass invariant check
- Strong encapsulation required

Semantics of Invariants Revisited

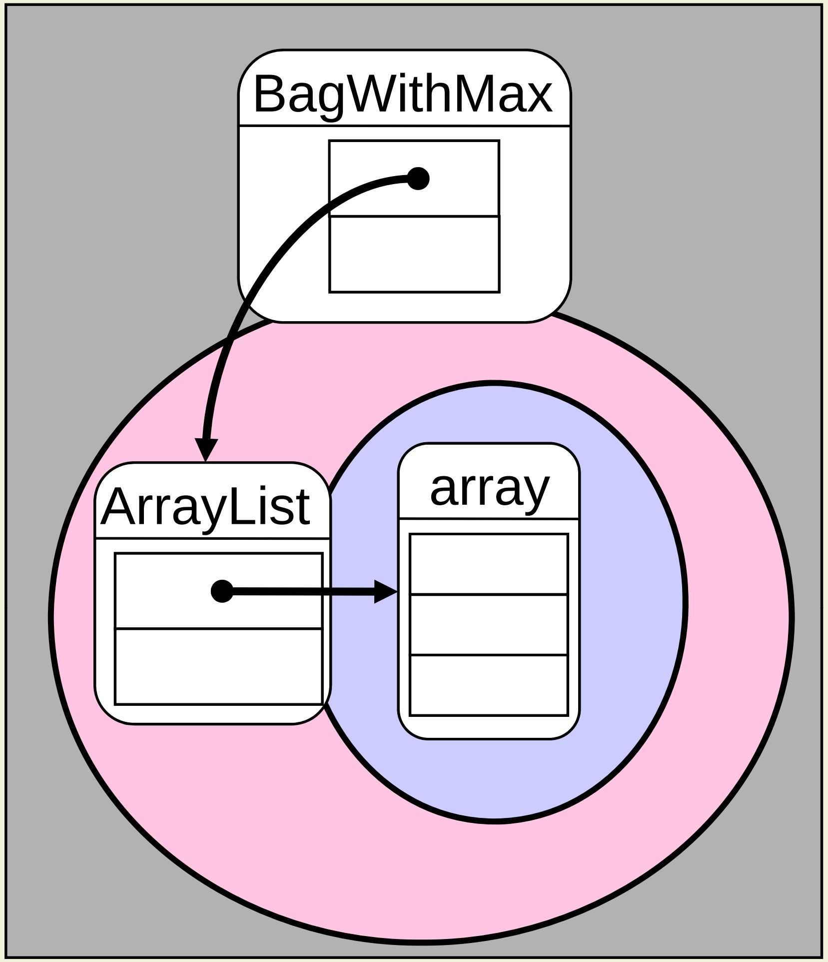
```
class ArrayList {
  // requires i>=0;
  public void add( int i ) { … }
  … }
```

- **Invariant** of BagWithMax **does not hold in** the **poststate** of call theList.add
- **Modularity problem:** BagWithMax not known when ArrayList is written

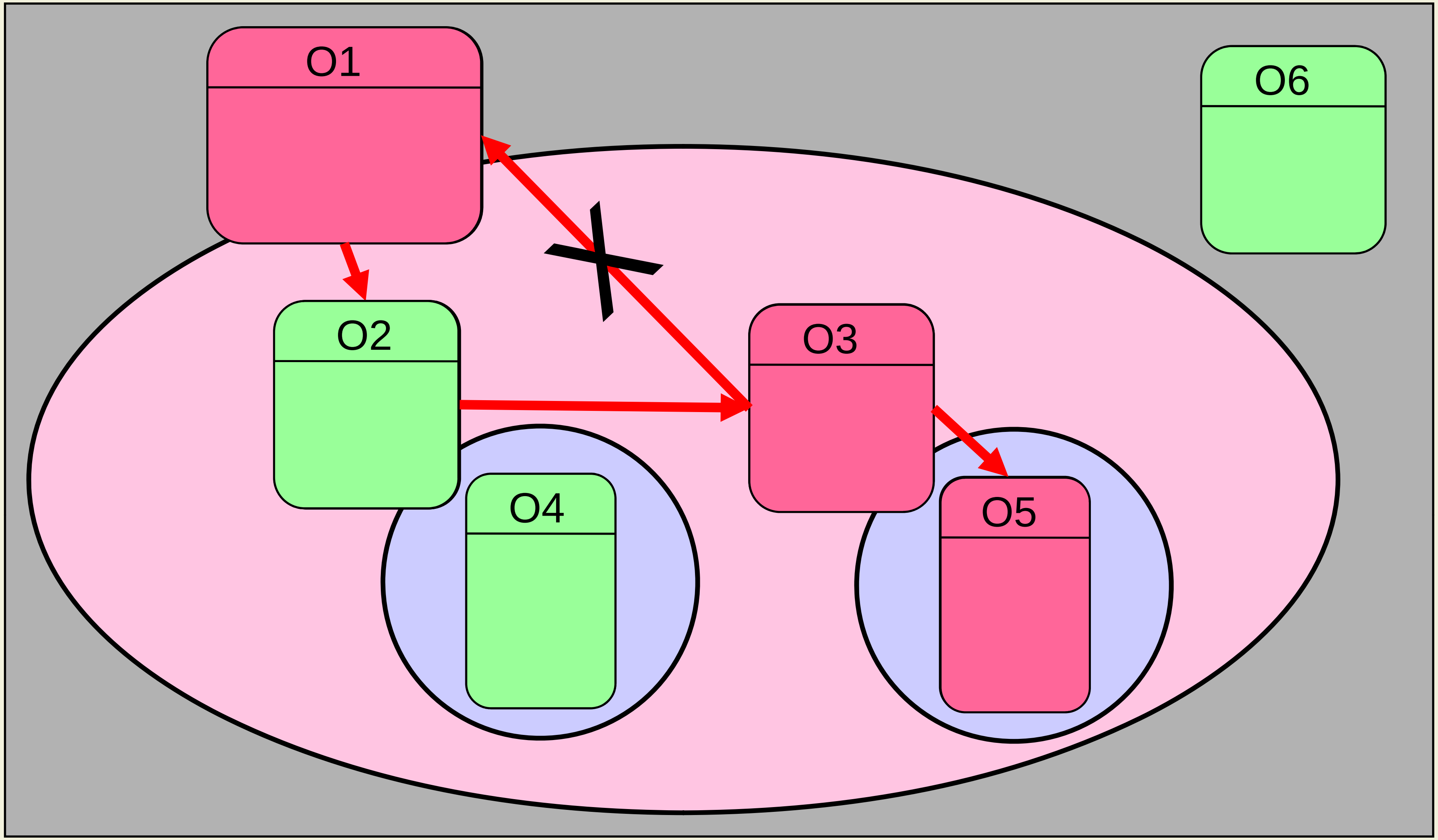
```
class BagWithMax {
  private rep ArrayList theList;
  private int maxElem;

  // invariant theList != null &&
  //       ( ∀i.0 <= i < theList.next ==>
  //       theList.array[ i ] <= maxElem )

  // requires i>=0
  public void insert( int i ) {
   theList.add( i );
   if ( i > maxElem )  maxElem = i;
  }
}
```

Refined Semantics

- Invariant may be broken while internal representation of an object is being modified
- Owned objects are part of the internal representation
- Exclude invariants of (transitive) owners from expected invariants

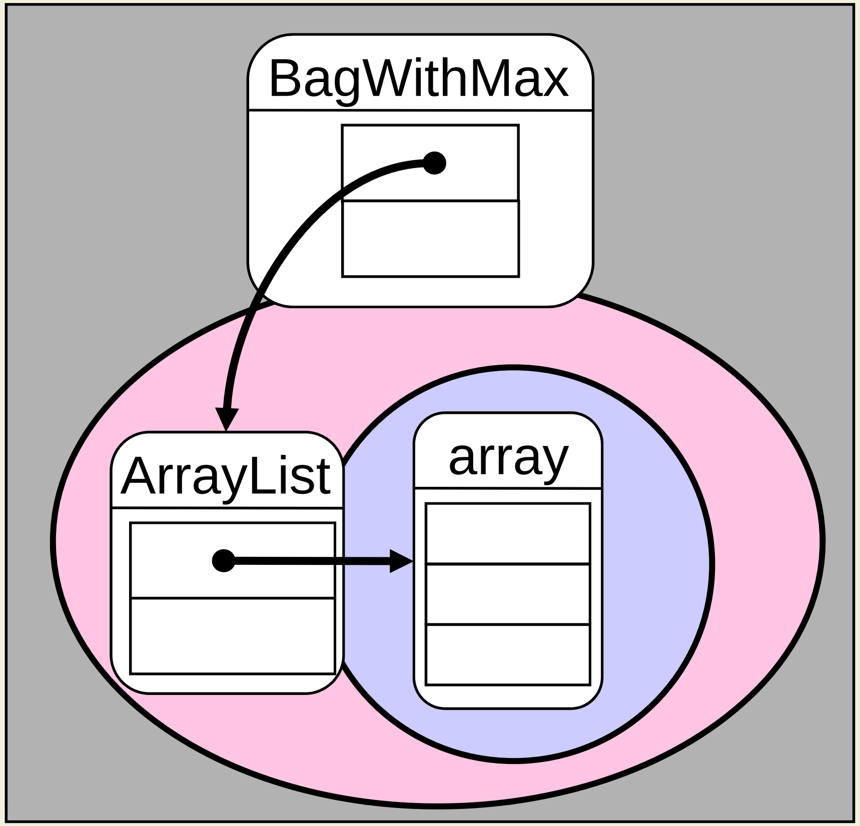


Refined Semantics: Example



Reasoning Idea

- Divide invariant check
  - A method's **implementor** is responsible for the objects in the context of the receiver
  - A method's **caller** is responsible for the objects in its context




```
class ArrayList {
  public void add( int i ) { … }
  … }
```

```
class BagWithMax {
  public void insert( int i ) {
    theList.add( i );
    if ( i > maxElem )  maxElem = i;
  }  … }
```

Checking Invariants for Object Structures

- The invariant of object X may depend on the state of X and of all objects (transitively) owned by X
- For every method, assume the invariants of **all objects transitively owned by owner of receiver** in the prestate
- For every method (and constructor), prove that all invariants that are possibly violated by the body **except for the (transitive) owners of receiver** hold (again) in the poststate **and before calls that might call back**

Example

```
class ArrayList {
  // requires i>=0;
  public void add( int i ) { … }
  … }
```

Check invariant of **this**

```
class BagWithMax {
  private rep ArrayList theList;
  private int maxElem;

  // invariant theList != null &&
  //        ( ∀i.0 <= i < theList.next ==>
  //        theList.array[ i ] <= maxElem )

  // requires i>=0
  public void insert( int i ) {
    theList.add( i );
    if ( i > maxElem )  maxElem = i;
  }
}
```

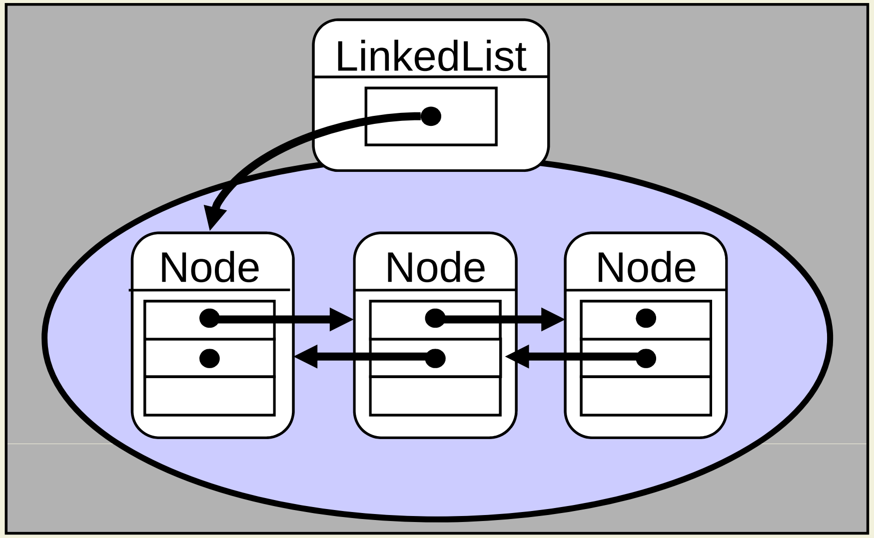
Admissible invariant

Invariant of **this** allowed to be broken

Check invariant of **this**

Limitations

- (Mutually) recursive data structures
  - No (mutual) ownership
  - Peer relationship
- Direct field access of peer objects
- Solution: check all vulnerable invariants of peer objects




```
class Node {
  peer Node next, prev;   int elem;
  // invariant next == null ||
  //               next.prev == this
  Node( int i, peer Node n ) {
    elem = i; next = n;
    if ( n != null ) n.prev = this;
  }
  ... }
```

Informal Guidelines

- Organize your system in layers
- A method execution should only modify objects in the same or underlying layers and only call methods in those layers
- The invariant of an object X should only depend on the state of objects in the layer that contains X or underlying layers

Informal Guidelines (cont‘d)

- If the invariant of an object X depends on a field f of an object in the same layer (including X itself):
  - Make sure that every method that can assign to f preserves the invariant of X before it terminates or calls a method on an object in L (including X itself)
- If the invariant of an object X in layer L depends on a field or an array element g in a deeper layer:
  - Make sure that every method execution on a receiver in layer L that modifies g is executed on receiver X and preserves the invariant of X before it terminates or calls a method on an object in L (including X itself)

Summary

- Frame properties
  - Crucial for program verification
  - Difficult to specify and prove (abstraction)
  - No good solution for runtime assertion checking
- Invariants
  - Semantics of invariants is non-trivial
  - Handling callbacks is difficult, especially for runtime assertion checking
  - Invariants of object structures require strong encapsulation