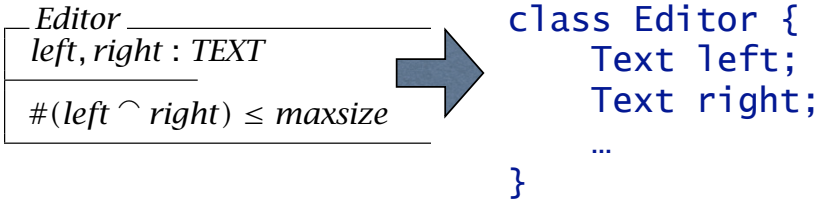
From Specification to Code

(and back again)

Software Engineering
Andreas Zeller • Saarland University

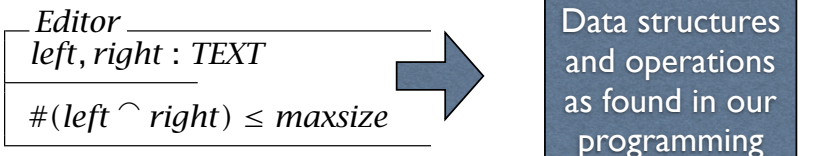
The Challenge




```
class Editor {
    Text left;
    Text right;
    …
}
```

From Jacky, “The way to Z”

Refinement



Data structures and operations as found in our programming language

We need to implement the abstract operation (from spec) in a concrete data structure (in code)

Data Refinement

- Most Z data types can be directly refined into traditional data structures
- *Sets* become arrays or trees
- *Sequences* become lists or arrays
- *Maps* become hash tables or trees

Schema Refinement

- Z variables are translated into variables or (better) object attributes

Editor

$left, right : TEXT$

$\#(left \frown right) \leq maxsize$

```
class Editor {
    Text left;
    Text right;
    …
}
```

Operation Refinement

- Operations must be *implemented* in the actual programming language
- Implementation is *constructive* while specification is (typically) *declarative*
- We want to ensure correctness!

Refinement

Abstract

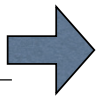
$s : \mathbb{P}\, X$

AStore

$\Delta Abstract$

$x? : X$

$s' = s \cup \{x?\}$



Data structures and operations as found in our programming language

Refinement

Abstract

$s : \mathbb{P}\, X$

Concrete

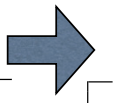
$ss : \text{seq}\, X$

AStore

$\Delta Abstract$

$x? : X$

$s' = s \cup \{x?\}$



CStore

$\Delta Concrete$

$x? : X$

$ss' = ss \frown \langle x? \rangle$

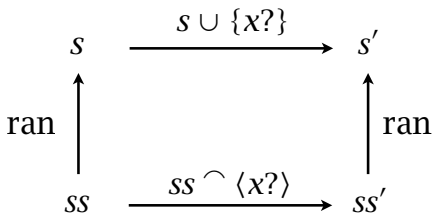
The concrete operation must satisfy all properties of the abstract operation!

Refinement

- We need an operation to express the relationship between the representations

Abstract Operation

Concrete Operation



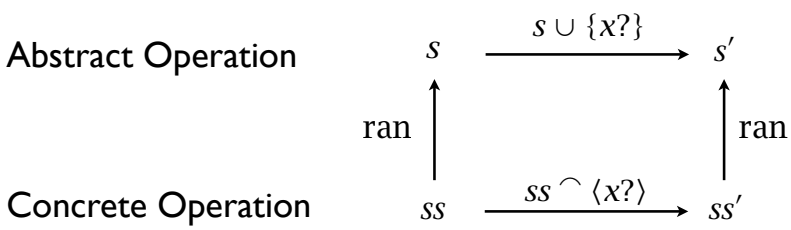
Refinement using *range*

$\text{s} = \text{ran}\, ss \land s' = \text{ran}\, ss'$

We must show that the two data structures are equal!

Refinement Proof

$ss' = ss \frown \langle x? \rangle \land s = \operatorname{ran} ss \land s' = \operatorname{ran} ss' \Rightarrow s' = s \cup \{x?\}$



Why is this a valid formula?

If all preconditions are met, then the result must be equivalent, too.

Refinement Schema

- Defines relationship between concrete and abstract structures

Abs

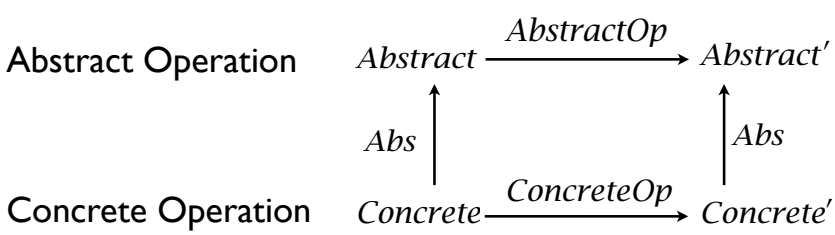
Abstract
Concrete

$s = \operatorname{ran} ss$

The range operator "ran" is the schema appropriate for this example. Every other refinement must come with its own operation.

Proof Obligation

$$\forall \Delta Abstract;\ \Delta Concrete;\ x? : X \bullet$$
$$\operatorname{pre} AbstractOp \land ConcreteOp \land \Delta Abs \Rightarrow AbstractOp$$



This has to be shown for every operation!

This is the more abstract pattern

All Proof Obliga

Liskov Substitution Principle

- An object of a superclass should always be substitutable by an object of a subclass:
  - Same or weaker preconditions
  - Same or stronger postconditions
- Derived methods should *not assume more or deliver less*

- Valid initial state
  $\forall Abstract; Concrete \bullet ConInit \land Abs \Rightarrow AbsInit$
- Same or weaker precondition
  $\forall Abstract; Concrete; x? : X \bullet$
  $\quad \text{pre } AbstractOp \land Abs \Rightarrow \text{pre } ConcreteOp$
- Same or stronger postcondition
  $\forall \Delta Abstract; \Delta Concrete; x? : X \bullet$
  $\text{pre } AbstractOp \land ConcreteOp \land \Delta Abs \Rightarrow AbstractOp$

Now that is a lot to prove!

A Pragmatic Approach

- Instead of *proving* all conditions, we may just as well *check* them.
- Essence of *design by contract*

Design by Contract

- A *contract* of a method describes
  - what the method *requires* (precondition)
  - what the method *provides* (postcondition)
- The contract binds clients (method callers) and suppliers (method implementors)

From Meyer: Object-Oriented Software Construction, §11 “Design by Contract”

A Stack

```
class STACK[G]      -- A stack of G's
    count: INTEGER
    empty: BOOLEAN -- true if empty
    full: BOOLEAN  -- true if full




end
```

An example in Eiffel – an OO language that supports contracts

A Stack

```
class STACK[G]      -- A stack of G's
    count: INTEGER
    empty: BOOLEAN -- true if empty
    full: BOOLEAN  -- true if full


    top: G          -- returns topmost item
        require     -- precondition
            not empty
        do
            …       -- implementation
        end

end
```

http://en.wikibooks.org/wiki/Computer_programming/Design_by_Contract

A Stack

```
class STACK[G]      -- A stack of G's
    count: INTEGER
    empty: BOOLEAN -- true if empty
    full: BOOLEAN  -- true if full
    put(x: G)       -- add x to stack
        require     -- precondition
            not full
        do …        -- implementation
        ensure      -- postcondition
            not empty
            item = x
            count = old count + 1
        end
end
```

http://en.wikibooks.org/wiki/Computer_programming/Design_by_Contract

Stacks in Z

Stack

$count : \mathbb{N}$
$state : \{empty, filled, full\}$
$item : G$

put

$\Delta Stack$
$x? : G$

$state \neq full$
$state' \neq empty$
$item' = x?$
$count' = count + 1$

Preconditions

- Expresses the constraints under which a *method functions properly*
- Applies to all calls of the method

```
top: G          -- returns topmost item
    require     -- precondition
        not empty
```

Postconditions

- Expresses the *properties of the state* resulting from a method execution
- Applies to all calls of the method

```
put(x: G)       -- add x to stack
    ensure      -- postcondition
        not empty
        item = x
        count = old count + 1
    end
```

"old" stands for the value at method entry

Checking Contracts

- Contracts are *checked at runtime*
- Failing contracts indicate internal errors (and therefore raise exceptions)
- Contracts guarantee *correctness* – *if* the program terminates
- Can also be used for program proofs (as Z)

Rights and Obligations

If you promise to call *m* with *pre* satisfied, then I, in return, promise to deliver a final state in which *post* is satisfied.

This is the actual **contract** – between client and supplier

Design by Contract

Method	Obligations	Benefits
Client	Satisfy precondition	Obtain postcondition
Supplier	Satisfy postcondition	Rely on precondition

Design by Contract

put(x)	Obligations	Benefits
Client	Only call put(x) on a non-full stack	• Stack is updated • x is on top • Count is increased
Supplier	• Update stack • Put x on top • Increase count	Need not check whether stack is full

Contract Violations

- A violation in the *precondition* indicates a defect in the *client*
- A violation in the *postcondition* indicates a defect in the *supplier*
- Useful for locating defects (and for putting the blame on someone)

A Bounded Stack

```
class BOUNDED_STACK[G]  -- A stack of G's
    count: INTEGER
    capacity: INTEGER
    … more attributes and methods




end
```

An Invariant

```
class BOUNDED_STACK[G]  -- A stack of G's
    count: INTEGER
    capacity: INTEGER
    … more attributes and methods
    put(x: G)       -- add x to stack
        require     -- precondition
            0 ≤ count and count ≤ capacity
            …
        do …        -- implementation
        ensure      -- postcondition
            0 ≤ count and count ≤ capacity
            …
        end
end
```

We call this an invariant - a condition that holds at the beginning and at the end of every public method

Class Invariants

```
class BOUNDED_STACK[G]  -- A stack of G's
    count: INTEGER
    capacity: INTEGER
    … more attributes and methods

    invariant
        0 ≤ count
        count ≤ capacity
        capacity = rep.capacity
        empty = (count = 0)
        full = (count = capacity)
        count > 0 ⇒ rep[count] = item

end
```

rep stands for the internal representation (say, an array)
Note: In Eiffel, array[1] is the first element

Class Invariants…

- must hold after the constructor
- must hold *before* and *after* every public method call
- must hold *before* the destructor (if any)

Contracts and Inheritance

- The *Liskov substitution principle* applies:
  - same or weaker precondition
  - same or stronger postcondition
- In Eiffel, parent pre-/postconditions are always tested first

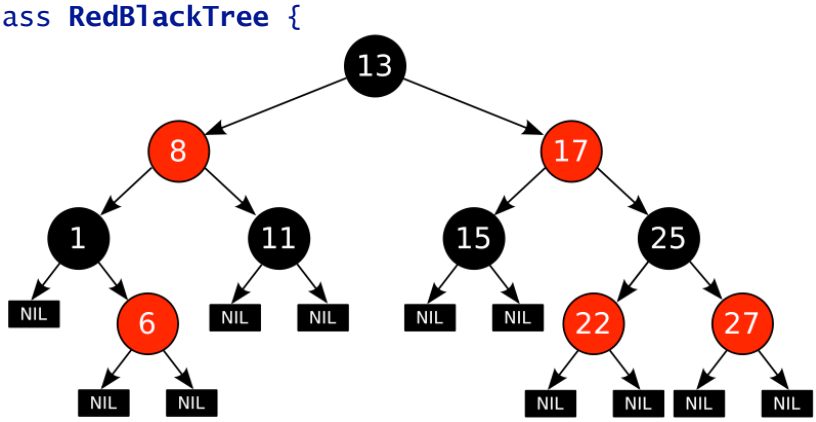
Invariants in Java

- Few languages explicitly support contracts
- We therefore need to implement them using the *available language constructs*
- Most frequently used: *assertions*

A Red/Black Tree

http://en.wikipedia.org/wiki/Red_black_tree

```
class RedBlackTree {
}
```



Class Invariant

```
class RedBlackTree {
  private boolean inv() {
    assert rootHasNoParent();
    assert rootIsBlack();
    assert redNodesHaveOnlyBlackChildren();
    assert equalNumberOfBlackNodesOnSubtrees();
    assert treeIsAcyclic();
    assert parentsAreConsistent();

    return true;
  }
}
```

We use an inv() method to check the invariant – using assert()

```
class RedBlackTree {
  private boolean redNodesHaveOnlyBlackChildren()
  {
    workList = new LinkedList();
    workList.add(rootNode());
    while (!workList.isEmpty()) {
      Node current = (Node)workList.removeFirst();
      Node cl = current.left;
      Node cr = current.right;
      if (current.color == RED) {
        assert cl == null || cl.color == BLACK;
        assert cr == null || cr.color == BLACK;
      }
      if (cl != null) workList.add(cl);
      if (cr != null) workList.add(cr);
    }

    return true;
}
```

This is one of the checked properties

Using Contracts

```
class RedBlackTree {
  void add(Object element) {
    assert inv();              // Invariant

    // actual operation goes here

    assert inv();              // Invariant
    assert has(element);       // Postcondition
  }
}
```

We check the invariant at the beginning and the end of each public method

Java Modeling Language

- JML is a formal specification language
- Provides annotations (assertions) for
  - preconditions
  - postconditions
  - invariants
- All integrated into Java code

Using JML

- JML annotations are added as *comments* in Java source (`/*@ … @*/` or `//@ …`)
- Contracts expressed as *boolean expressions*
  - using some special operators `\result, \forall, \old`…
  - and some special keywords `requires, ensures, invariant`…

Red/Black Tree in JML

```
class RedBlackTree {
    //@ invariant rootHasNoParent()
    //@ invariant rootIsBlack()
    //@ more invariants…

    //@ ensures has(\old element)
    void add(Object element) {

        // actual operation goes here
    }
}
```

Note the usage of \old to refer to the initial value

Data Invariants in JML

```
private final Object[] objs;
/*@ invariant objs != null &&
    objs.length == CURRENT_OBJS_SIZE &&
    (\forall int i;
             0 <= i && i < CURRENT_OBJS_SIZE;
             objs[i] != null);
@*/
```

We can document design decisions this way.

Example: Date

Date

$$hours : \mathbb{N}$$
$$minutes : \mathbb{N}$$
$$seconds : \mathbb{N}$$

$$0 \leq hours \leq 23$$
$$0 \leq minutes \leq 59$$
$$0 \leq seconds \leq 60$$

set_hour

$$\Delta Date$$
$$h? : \mathbb{N}$$

$$0 \leq h? \leq 23$$
$$hours' = h?$$
$$minutes' = minutes$$
$$seconds' = seconds$$

Date in Eiffel

```
invariant
    0 ≤ hours()   && hours() ≤ 23
    0 ≤ minutes() && minutes() ≤ 59
    0 ≤ seconds() && seconds() ≤ 60

set_hour (h: INTEGER) is
        -- Set the hour from `h'
    require
        0 ≤ h and h ≤ 23
    ensure
        hour = h
        minutes = old minutes
        seconds = old seconds
```

Date in Java

```
boolean inv()   // invariant
{
    return (0 ≤ hour()    && hour() ≤ 23) &&
           (0 ≤ minutes() && minutes() ≤ 59) &&
           (0 ≤ seconds() && seconds() ≤ 60);
}
```

Date in Java

```
    void set_hour(int h)
    {
       int old_minutes = minutes();
       int old_seconds = seconds();
       assert (inv());

       // Actual code goes here

       assert (inv());
       assert (hour() == h);
       assert (minutes() == old_minutes &&
               seconds() == old_seconds);
    }
```

Date in JML

```
class Date {
  //@ invariant 0 ≤ hours()   && hours() ≤ 23
  //@ invariant 0 ≤ minutes() && minutes() ≤ 59
  //@ invariant 0 ≤ seconds() && seconds() ≤ 60

  /*@ requires 0 ≤ h && h ≤ 23
    @ ensures  hours() == h &&
    @          minutes() == \old(minutes()) &&
    @          seconds() == \old(seconds())
    @*/
  void set_hour(int h) { … }
}
```

A Purse in JML

```
public class Purse {
  final int MAX_BALANCE;
  int balance;
  //@ invariant  0 ≤ balance && balance ≤ MAX_BALANCE;

  byte[] pin;
  /*@ invariant pin != null && pin.length == 4 &&
    @           (\forall int i; 0 ≤ i && i < 4;
    @                         0 ≤ byte[i] && byte[i] ≤ 9)
    @*/

  /*@ requires   amount ≥ 0;
    @ assignable balance;
    @ ensures    balance == \old(balance) - amount &&
    @            \result == balance;
    @ signals    (PurseException) balance == \old(balance);
    @*/
  int debit(int amount) throws PurseException { … }
```

To check or not to check

- Checking assertions is expensive
  (especially invariants)
- Assertions can be activated and deactivated
  (typically at compile time)
- Assertions should not have any side effects
  (i.e., change the state of the program)
- Assertions should be the *only* mechanism to check for contracts and consistency
  (exception: external input must *always* be checked by extra code)

Hoare on Assertions

It is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he gets to sea?

C.A.R. Hoare (1973): Hints on Programming Language Design. Stanford University Artificial Intelligence memo AIM-224/STAN-CS-73-403.

Meyer on Assertions

- Assess, from an engineering perspective:
  - How much you trust the correctness of the software
  - How crucial it is to get the utmost efficiency
  - How serious the damage of an undetected run-time error can be
- Consider at least *checking preconditions*

Static JML checking

- JML is usually checked at *runtime*…
- but can also be used to check programs at *compile time!*
- Approach: ESC/Java by Leino et al. ("Extended Static Checker for Java")

```
class Bag {
    int[] a;
    int n;


    int extractMin() {
        int m = Integer.MAX_VALUE;
        int mindex = 0;
        for (int i = 1; i ≤ n; i++) {
            if (a[i] < m) {
                mindex = i; m = a[i];
            }
        }
        n--;

        a[mindex] = a[n];
        return m;
    }
}
```

From ESC/Java2 documentation
http://kind.ucd.ie/products/opensource/ESCJava2/

ESC/Java Demo

```
class Bag {
    int[] a;
    int n;


    int extractMin() {
        int m = Integer.MAX_VALUE;
        int mindex = 0;
        for (int i = 1; i ≤ n; i++) {
            if (a[i] < m) {
                mindex = i; m = a[i];
            }
        }
        n--;

        a[mindex] = a[n];
        return m;
    }
}
```

From ESC/Java2 documentation
http://kind.ucd.ie/products/opensource/ESCJava2/

```
class Bag {
    int[] a;
    int n;


    int extractMin() {
        int m = Integer.MAX_VALUE;
        int mindex = 0;
        for (int i = 1; i ≤ n; i++) {
            if (a[i] < m) {
                mindex = i; m = a[i];
            }
        }
        n--;

        a[mindex] = a[n];
        return m;
    }
}
```

From ESC/Java2 documentation
http://kind.ucd.ie/products/opensource/ESCJava2/

```
class Bag {
    int[] a;//@ invariant a != null;
    int n;  //@ invariant 0 ≤ n && n ≤ a.length;

    //@ requires n > 0;
    int extractMin() {
        int m = Integer.MAX_VALUE;
        int mindex = 0;
        for (int i = 0; i < n; i++) {
            if (a[i] < m) {
                mindex = i; m = a[i];
            }
        }
        n--;

        a[mindex] = a[n];
        return m;
    }
}
```

From ESC/Java2 documentation
http://kind.ucd.ie/products/opensource/ESCJava2/

Mining Specifications

- Specifications can also be *learned* from existing code
- Popular approach: learn from *executions*

Daikon

```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

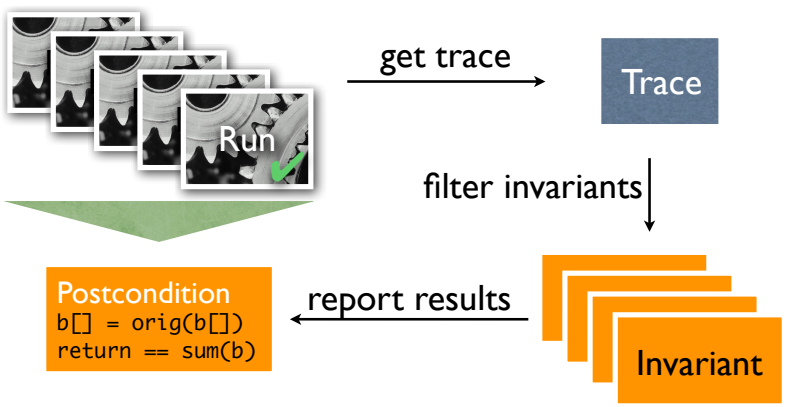
Precondition
```
n == size(b[])
b != null
n <= 13
n >= 7
```

Postcondition
```
b[] = orig(b[])
return == sum(b)
```

- Run with 100 randomly generated arrays of length 7–13

Daikon was developed by Michael Ernst (1999)

Daikon

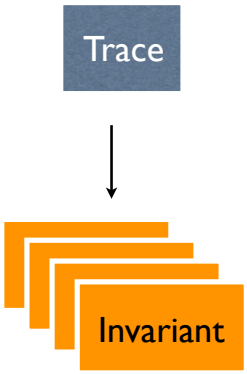


Getting the Trace

- Records all variable values at all function entries and exits
- Uses VALGRIND to create the trace

Filtering Invariants

- Daikon has a library of *invariant patterns* over variables and constants
- Only matching patterns are preserved



Method Specifications

using *primitive data*

x = 6	x ∈ {2, 5, –30}	x < y
y = 5x + 10	z = 4x +12y +3	z = fn(x, y)

using *composite data*

A subseq B	x ∈ A	sorted(A)

checked at method entry + exit

Object Invariants

string.content[string.length] = '\0'
node.left.value ≤ node.right.value
this.next.last = this

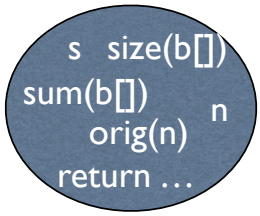
checked at entry + exit of public methods

Matching Invariants

```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

A == B

Pattern



Variables

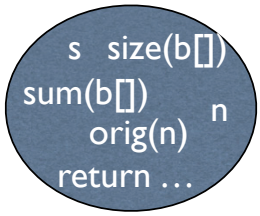
Matching Invariants

==	s	n	size(b[])	sum(b[])	orig(n)	ret
s		✘			✘	
n	✘			✘		✘
size(b[])						
sum(b[])		✘				
orig(n)	✘					✘
ret		✘			✘	

run 1

A == B

Pattern



Variables

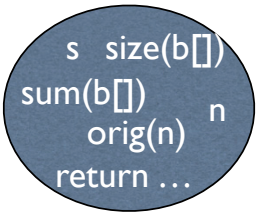
Matching Invariants

==	s	n	size(b[])	sum(b[])	orig(n)	ret
s		✘	✘		✘	
n	✘			✘	✘	✘
size(b[])	✘			✘		✘
sum(b[])		✘	✘		✘	
orig(n)	✘	✘		✘		✘
ret		✘	✘		✘	

run 2

A == B

Pattern



Variables

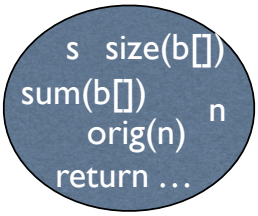
Matching Invariants

==	s	n	size(b[])	sum(b[])	orig(n)	ret
s		✘	✘		✘	
n	✘			✘	✘	✘
size(b[])	✘			✘		✘
sum(b[])		✘	✘		✘	
orig(n)	✘	✘		✘		✘
ret		✘	✘		✘	

run 3

A == B

Pattern



Variables

Matching Invariants

==	s	n	size(b[])	sum(b[])	orig(n)	ret
s		✘	✘		✘	
n	✘			✘	✘	✘
size(b[])	✘			✘		✘
sum(b[])		✘	✘		✘	
orig(n)	✘	✘		✘		✘
ret		✘	✘		✘	

s == sum(b[])

s == ret

n == size(b[])

ret == sum(b[])

Matching Invariants

```
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

s == sum(b[])

s == ret

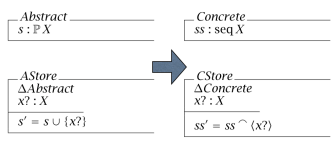
n == size(b[])

ret == sum(b[])

Daikon Discussed

- As long as some property can be observed, it can be added as a pattern
- Pattern vocabulary determines the invariants that can be found ("sum()", etc.)
- Checking all patterns (and combinations!) is expensive
- Trivial invariants must be eliminated

Refinement



The concrete operation must satisfy all properties of the abstract operation!

Class Invariants

```
class BOUNDED_STACK[G]  -- A stack of G's
    count: INTEGER
    capacity: INTEGER
    … more attributes and methods

    invariant
        0 ≤ count
        count ≤ capacity
        capacity = rep.capacity
        empty = (count = 0)
        full = (count = capacity - 1)
        count > 0 ⇒ rep[count] = item

end
```

Summary

```
class Bag {
    int[] a;//@ invariant a != null;
    int n;  //@ invariant 0 ≤ n && n ≤ a.length;

    //@ requires n > 0;
    int extractMin() {
        int m = Integer.MAX_VALUE;
        int mindex = 0;
        for (int i = 0; i < n; i++) {
            if (a[i] < m) {
                mindex = i; m = a[i];
            }
        }
        n--;

        a[mindex] = a[n];
        return m;
    }
}
```

Daikon