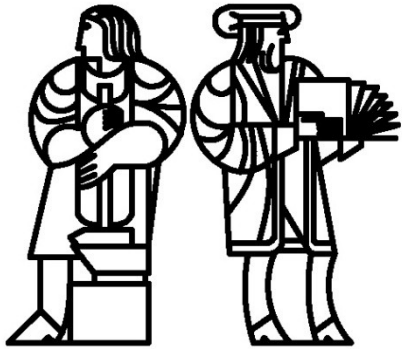
6.170 Lecture 8
Representation invariants and abstraction functions



MIT EECS

CharSet Abstraction

// Overview: CharSets are finite mutable sets of Characters

// effects: creates a fresh, empty CharSet
public CharSet ( )

// modifies: this
// effects: $this_{post} = this_{pre} \cup \{c\}$
public void insert (Character c);

// modifies: this
// effects: $this_{post} = this_{pre} - \{c\}$
public void delete (Character c);

// returns: ($c \in this$)
public boolean member (Character c);

// returns: cardinality of this
public int size ( );

A CharSet Implementation ?

```
class CharSet {
      private List<Character> elts
          = new ArrayList<Character> ();
      public void insert (Character c)   {
          elts.add (c);
      }
      public void delete (Character c)   {
          elts.remove (c);
      }
      public boolean member (Character c) {
          return elts.contains (c);
      }
      public int size ( ) {
          return elts.size ();
      }
}
```

```
CharSet s = new CharSet( );
Character a
    = new Character('a');
s.insert(a);
s.insert(a);
s.delete(a);
if (s.member(a))
    // print wrong;
else
    // print right;
```

Where Is the Error?

The answer to this question tells you what needs to be fixed

Put the blame on delete

It should remove all occurrences

Put the blame on insert

It should not insert a character that is already there

1st viewpoint on the design flaw ***Missing abstraction function***

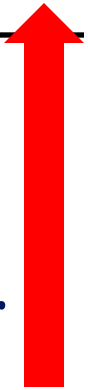
We did not state how the abstract set is related to elts

2nd viewpoint on the design flaw ***Missing rep invariant***

An implicit part of the precondition of delete did not hold

Two aspects of the same thing

A Representation Invariant: A Solution to a Paradox



The state of the program needs to be constrained for the methods to work correctly

Precondition: constraints that the client is responsible for

But the client has no clue what is going on inside the object

Rep invariant: constraints guaranteed by all the public methods to one another; a pact among them

"I promise to leave the common area tidy when I exit"; In return, I (the method) expect to get it tidy (satisfying specific constraints) every time I enter

Writing Representation Invariants

Write it this way:

```
class CharSet {
  // Rep invariant: elts has no nulls and no duplicates
  private List<Character> elts;
  …
```

Or, if you are the pedantic sort:

∀ indices i of elts . elts.elementAt(i) ≠ null

∀ indices i, j of elts .

i ≠ j ⇒ ¬ elts.elementAt(i).equals(elts.elementAt(j))

Now, where is the error?

```
// Rep invariant: elts has no nulls and no duplicates

public void insert (Character c) {
   elts.add (c);
}

public void delete (Character c) {
   elts.remove (c);
}
```

Rep invariant for Account

```
class Account {
    private List<Transaction> transactions;
    private int balance;
    …
}
```

// real-world constraints

balance ≥ 0

balance = Σ_i transactions.get(i).amount

// implementation-related constraints

transactions ≠ null

no nulls in transactions

Listing the elements of a CharSet

Consider adding the following method to CharSet

```
// returns: a List containing the members of this
public List<Character> getElts ( );
```

Consider this implementation:

```
public List<Character> getElts ( )  { return elts; }
```

Recall rep invariant: elts has no nulls and no duplicates

The implementation of getElts preserves the rep invariant

… sort of

Representation exposure

Consider the client code (outside implementation of the type)

```
CharSet s = new CharSet();
Character a = new Character('a');
s.insert(a);
s.getElts( ).add(a);
s.delete(a);
if (s.member(a)) …
```

Representation exposure is almost always evil

If you do it, document why and how

And feel guilty about it!

Avoiding Rep Exposure

Exploit immutability

```
Character choose ( ) {
    return elts.elementAt (0);
}
```

Character is immutable

Make a copy

```
List<Character> getElts ( ) {
    return new ArrayList<Character>(elts);
    // or: return (ArrayList<Character>) elts.clone ( );
}
```

Mutating a copy doesn't affect the original

And keep the rep fields private

Checking rep Invariants

Should code check that the rep invariant holds?

Yes, if it's inexpensive

Yes, for debugging (even when it's expensive)

It's quite hard to justify turning the checking off

Some private methods need not check (Why?)

Practice Defensive Programming

Assume that you will make mistakes

Write and incorporate code designed to catch them

On entry:

Check rep invariant

Check preconditions (requires clause)

On exit:

Check rep invariant

Check postconditions

Checking the rep invariant helps you discover errors

Reasoning about the rep invariant helps you avoid errors

Or prove that they do not exist!

We will discuss such reasoning, later in the semester

Checking the Rep Invariant

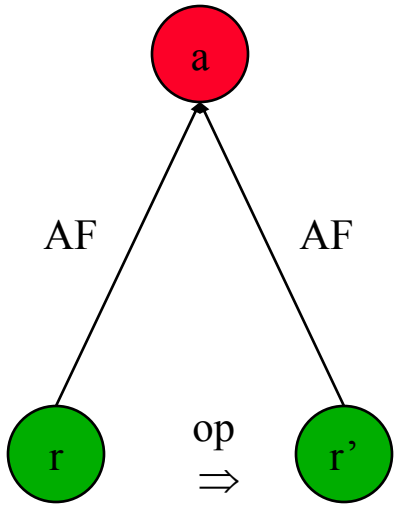
```
public void delete (Character c) {
      checkRep ( );
      elts.remove (c)
      // Is this guaranteed to get called?
      // See handouts for a less error-prone way to check at exit
      checkRep ();
}
...
/** Verify that elts contains no duplicates **/
private void checkRep( ) {
      for (int i = 0; i < elts.size( ); i++) {
          assert elts.indexOf(elts.elementAt(i)) == i;
      }
}
```

An alternative implementation: repOK() returns a boolean, and callers of repOK must check its return value

Benevolent Side Effects

Different implementation of member:

```
boolean member (Character c1) {
    int i = elts.indexOf (c1);
    if (i == -1) return false;
    // move-to-front optimization
    Character c2 = elts.elementAt (0);
    elts.set (0, c1);
    elts.set (i, c2);
    return true;
}
```



Speeds up repeated membership tests

Mutates rep, but does not change *abstract* value

AF maps both reps to the same abstract value

Rep Invariant constrains structure, not meaning

New implementation of insert that preserves invariant

```
public void insert (Character c) {
    Character cc = new Character(encrypt(c));
    if (!elts.contains(cc))
        elts.addElement(cc);
}
public boolean member (Character c) {
    return elts.contains (c);
}
```

Program still does wrong thing

Where is the error?

Abstraction function tells us

```
CharSet s = new CharSet( );
Character a = new Character('a'));
s.insert (a);
if (s.member(a))
    // print right;
else
    // print wrong;
```

Abstraction functions

The *abstraction function* relates the concrete representation to the abstract value it represents

AF: Object → abstract value

AF(CharSet this) = { c | c is contained in this.elts }

"set of Characters contained in this.elts"

Typically not executable

Combined with rep invariant

Allows us to examine operators independently

"Correctness" is now a local issue

Once again we can place the blame

Applying the abstraction function to the result of the call to insert yields AF(elts) ∪ {encrypt('a')}

What if we used this abstraction function?

AF(this) = { c | encrypt(c) is contained in this.elts }

AF(this) = { decrypt(c) | c is contained in this.elts }

The abstraction function is a function

Q: Why do we map concrete to abstract rather than vice versa?

It's not a function in the other direction.

E.g., lists [a,b] and [b,a] each represent the set {a, b}

It's not as useful in the other direction.

Can construct objects via the provided operators

Writing Abstraction Functions

The abstraction function must be defined properly on all representations that satisfy the rep invariant

Usually, rep invariant defines the domain of the AF

Writing the AF can be harder than writing the rep invariant

The problem lies in denoting range of abstraction function

- Not a problem for mathematical entities like sets
- For more complex abstractions, give them fields
- AF defines the value of each "specification field"

The overview section of the specification is the key

- Ideally, it provides a way of writing values of abstract type
- Having a printed representation is valuable for debugging

ADTs and specifications

An ADT is more than just a data structure

data structure + a set of conventions

Specification: only in terms of the abstraction

Never mentions the representation

Representation invariant: Object → boolean

Indicates whether a data structure is well-formed

Defines the set of valid values of the data structure

Abstraction function: Object → abstract value

What the data structure means (as an abstract value)

How the data structure is to be interpreted

How do you compute the inverse, abstract value→ Object ?

Key Points

Rep invariant

Which concrete values represent abstract values

Abstraction function

Which abstract value each concrete value represents

Together, they modularize the implementation

Can examine operators one at a time

Neither one is part of the abstraction (the ADT)

In practice

Always write a representation invariant

Write an abstraction function when you need it

- Write an informal one for most non-trivial classes
- A formal one is harder to write and usually less useful