Implementing an ADT: Representation invariants

CSE 331
University of Washington

Michael Ernst
A data abstraction is defined by a specification

An ADT is a collection of **procedural abstractions**

*Not* a collection of procedures

Together, these procedural abstractions provide:

- A set of values
  - *All* the ways of directly using that set of values
    - Creating
    - Manipulating
    - Observing

Creators and producers: make new values
Mutators: change the value (but don’t affect `==`)
Observers: allow one to tell values apart
ADTs and specifications

Specification: only in terms of the abstraction
Never mentions the representation
An ADT is more than just a data structure
data structure + a set of conventions

Why do we need to relate the specification to the representation?
Connecting specifications and implementations

Representation invariant: Object → boolean
Indicates whether a data structure is well-formed
Only well-formed representations are meaningful
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?
Implementation of an ADT is provided by a class

To implement a data abstraction:
  – Select the representation of instances, the rep
  – Implement operations in terms of that rep

Choose a representation so that
  – It is possible to implement operations
  – The most frequently used operations are efficient
    • But which will these be?
    • Abstraction allows the rep to change later
 CharSet Abstraction

// Overview: A CharSet is a finite mutable set 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. What client code will expose the error?

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();
s.insert('a');
s.insert('a');
s.delete('a');
if (s.member('a'))
    // print “wrong”;
else
    // print “right”;

Where is the error?
Where Is the Error?

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

*Perhaps* delete *is* wrong

It should remove all occurrences

*Perhaps* insert *is* wrong

It should not insert a character that is already there

How can we know?

The representation invariant tells us
The representation invariant

- States data structure well-formedness
- Holds before and after every CharSet operation
- Operation implementations (methods) may depend on it

Write it this way:

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

    ...
}
```

Or, if you are the pedantic sort:

\[
\forall \text{ indices } i \text{ of } \text{elts} . \text{elts.elementAt}(i) \neq \text{null} \\
\forall \text{ indices } i, j \text{ of } \text{elts} . \\
i \neq j \Rightarrow \neg \text{elts.elementAt}(i).\text{equals}(\text{elts.elementAt}(j))
\]
Now, we can locate the error

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

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

dpublic void delete(Character c) {
    elts.remove(c);
}
Another rep invariant example

class Account {
    private int balance;
    // history of all transactions
    private List<Transaction> transactions;
    ...
}

// 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:

    // Rep invariant: elts has no nulls and no duplicates
    public List<Character> getElts() { return elts; }

Does the implementation of getElts preserve the rep invariant?

... sort of
Consider this client code (outside the CharSet implementation):

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

**Representation exposure** is external access to the rep

Representation exposure is almost always **EVIL**

- Enables violation of abstraction boundaries and the rep invariant

If you do it, document why and how
- And feel guilty about it!

---

How can we avoid/prevent rep exposure?
Ways to avoid rep exposure

1. Exploit immutability
   Character choose() {
     return elts.elementAt(0);
   }
   Character is immutable.

2. 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.
   Don’t forget to make a copy on the way in!

3. Make an immutable copy
   List<Character> getElts() {
     return Collections.unmodifiableList<Character>(elts);
   }
   Client cannot mutate
   Still need to make a copy on the way in

Defining fields as private is not sufficient to hide the representation.
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?)
Checking the rep invariant

Rule of thumb: check on entry and on exit (why?)

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;
    }
}

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 term
The rep invariant constrains structure, not meaning

New implementation of insert that preserves the rep invariant:

```java
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);
}
```

The program is still wrong
Clients observe incorrect behavior
What client code exposes the error?
Where is the error?
We must consider the meaning
The abstraction function helps us

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