Representation invariants and abstraction functions

CSE 331

Autum 2010

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?

A data abstraction is defined by a specification

- 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, which is the key to understanding

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 (preferably easy) to implement operations
  - The most frequently used operations are efficient
    - But which will these be?
    - Abstraction allows changes to rep late in the game

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. Where is 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();
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
- *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
- Captures information that must be shared across implementations of multiple operations
- 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

$\forall$ indices i of elts . elts.elementAt(i) $\neq$ null
$\forall$ indices i, j of elts . i $\neq$ j $\Rightarrow$
    $\neg$ elts.elementAt(i).equals(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);
}

public 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 $\geq$ 0

balance = Σ_i transactions.get(i).amount

// implementation-related constraints:

transactions $\neq$ 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 & no duplicates
public List<Character> getElts()  { return elts; }
```

- Does the implementation of **`getElts`** preserve the rep invariant?

Well, sort of: Representation exposure

- Consider the client code

```
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 external access to the rep
- Representation exposure is almost always evil – why?
- If you do it, document why and how
  - And feel guilty about it!

Ways to avoid rep exposure

- 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!

- 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

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

```
/** 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
  - callers of **`repOK`** check its return value

Check on entry *and* on exit

- As a rule of thumb… but 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();
}
```

Practice defensive programming

- Assume that you will make mistakes
- Write and incorporate code designed to catch them
  - On entry
    - Check rep invariant
    - Check preconditions (<u>requires</u> 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 on

The rep invariant constrains structure, not meaning

- New implementation of insert that *preserves the rep 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);
}
```

- The program is still wrong
  - Clients observe incorrect behavior
  - Where is the error?
  - We must consider the meaning
  - The *abstraction function* helps us

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

Abstraction function: rep → abstract value

- The abstraction function maps 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
- The abstraction function lets us reason about behavior from the client perspective
- Our real goal is to satisfy the specification of insert
  - // modifies: this
  - // effects: $this_{post} = this_{pre} \cup \{c\}$
  - public void insert (Character c);
- Once again we can identify the problem
  - Applying the abstraction function to the result of the call to insert yields AF(elts) U {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 }

Placing the blame

- Our real goal is to satisfy the specification of insert:

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

- The AF tells us what the rep means (and lets us place the blame)

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

- Consider a call to insert:

  On entry, the meaning is $AF(this_{pre}) \approx elts_{pre}$
  On exit, the meaning is $AF(this_{post}) = AF(this_{pre}) \cup \{encrypt(\text{'a'})\}$

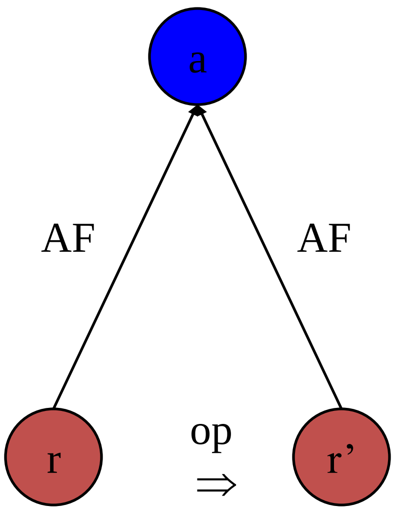
- What if we used this abstraction function?

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

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;
}
```



- Move-to-front speeds up repeated membership tests
- Mutates rep, but does not change *abstract* value
- AF maps both reps to the same abstract value

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.
  - Ex: 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 an abstraction function

- The domain: all representations that satisfy the rep invariant
- The range: can be tricky to denote
  - For mathematical entities like sets: easy
  - For more complex abstractions: give them fields
    - AF defines the value of each “specification field”
- The overview section of the specification should provide a way of writing abstract values
  - A printed representation is valuable for debugging

Summary

- 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