

6.170 Lecture 12 Understanding ADTs



John Guttag
MIT EECS

6.170 First of a Three Lecture Series

Theme is rigorous reasoning

Will be rather formal

- Don't expect you to use in practice
- Understanding what is involved aids informal reasoning
- Something all good programmers do in practice

Topics

- Reasoning about implementations of types
- Reasoning about uses of types
- Reasoning about subtypes
- Reasoning about loops
- Induction a recurring theme

6.170 A Data Abstraction Is Defined by a Specification

A collection of procedural abstractions

Not a collection of procedures

Together, procedural abstractions provide

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

Creators and producers create new values

Mutators change value (but don't effect =)

Observers allow one to tell values apart

The key to understanding

6.170 Implementation of an ADT Provided by a Class

To implement a data abstraction

- Select representation of instances, the *rep*
- Implement operations in terms of that rep

Choose rep so that

- It is possible (preferably easy) to implement operations
- Most frequently used operations are efficient
- But which will these be?
- Abstraction allows changes to rep late in game

6.170 What Is the Rep?

A data structure?

Not exactly

It's a data structure + a set of conventions

Conventions defined by

- Rep invariant*
 - Defines set of reachable values of the data structure
- Abstraction function*
 - How the data structure is to be interpreted
 - I.e., how it relates to the abstract values

6.170 CharSet Abstraction

Overview: CharSets are finite sets of chars

```
public CharSet ()
  effects: creates a fresh, empty CharSet
public void insert (char c);
  modifies: this
  effects: this' = this U {c}
public void delete (char c);
  modifies: this
  effects: this' = this - {c}
public boolean member (char c);
  effects: returns (c in this)
public int size ();
  effects: returns cardinality of this
```

6.170 A CharSet Implementation ?

```

class CharSet {
private Vector elts;
CharSet () {elts = new Vector ();}
public void delete (char c) { //JG's code
  elts.removeElement (new Character (c));
}
public void insert (char c) { //JC's code
  Character cc = new Character (c);
  elts.addElement (cc);
}
public boolean member (char c) {
  return elts.contains (new Character (c));
}
public int size () {return elts.size ();}
}

```

```

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

```

© John Chapin/John Guttag Spring 2000 Slide 7

6.170 So, Where Is the Error?

An important question, since
*"It's not whether you win or lose,
 it's how you place the blame."*
 Tells you what needs to be fixed

Delete is wrong
 It should remove all occurrences

Insert is wrong
 It should not insert a char that is already there

We have no way of knowing
 Or do we?

What representation invariants are all about

© John Chapin/John Guttag Spring 2000 Slide 8

6.170 A Rep Invariant

Captures information that must be shared across implementations of multiple operations

The way I would write it

```

class CharSet {
private Vector elts;
//Rep inv: noDups(elts) & allChars(elts)
...
}

```

Or, if you are the pedantic sort

- ⊕ indices i, j of elts .
 $elts.elementAt(i).equals(elts.elementAt(j)) \Rightarrow i = j$
- & ⊕ elements e of elts . $e instanceof Character$

© John Chapin/John Guttag Spring 2000 Slide 9

6.170 Now, Whose Fault Is It ?

```

public void delete (char c) { //JG's code
  elts.removeElement (new Character (c));
}
public void insert (char c) { //JC's code
  Character cc = new Character (c);
  elts.addElement (cc);
}

```

JC's
 Obviously

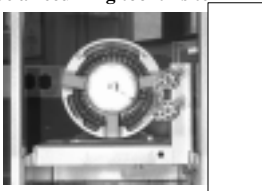
Because insert breaks the rep invariant

Reason about invariants using induction

© John Chapin/John Guttag Spring 2000 Slide 10

6.170 Induction

Induction will be a recurring tool this term

Not related to 

Way we think about things that are potentially infinite
 Or so large that they might as well be infinite

Most interesting computations fall in that class

© John Chapin/John Guttag Spring 2000 Slide 11

6.170 A Review of Ordinary Induction

Done with respect to a set of values, S

Start with a set of base values, B

Choose a set of operators, O, s.t. every value in S can be generated by applying the operators in O a finite number of times to the values in B

To prove that a property, P: $S \rightarrow \text{Boolean}$, holds
 Base case: $\forall s \in B. P(s)$
 Induction step: $P(s) \Rightarrow P(o(s))$, for all $o \in O$

Many other kinds of induction
 Complete induction (later in term)
 Engineers induction
 Deprecated

© John Chapin/John Guttag Spring 2000 Slide 12

6.170 **The Proof of I: noDups(elts) & allChars(elts)**

Base case
creator: elts = empty vector satisfies I

Induction step, assume I holds before other ops are called
member: doesn't change elts, so I preserved
delete
 delete either leaves elts untouched or removes element
 I can only be made false by adding elements
 So rep invariant must be preserved
insert
 If c in elts, elts unchanged and I preserved
 if c not in elts, adding c does not introduce a duplicate,
 and c is a character

© John Chapin/John Guttag Spring 2000 Slide 19

6.170 **A Question**

Q: Why did I have to look at member in proof ?
Specification says that it does not mutate set

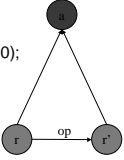
A: Specification does not preclude modifying rep

© John Chapin/John Guttag Spring 2000 Slide 20

6.170 **Benevolent Side Effects**

Variant of member op

```
boolean member (char c) {
  Character c1 = new Character (c);
  int i = elts.indexOf (c1);
  if (i == -1) return false;
  Character c2 = (Character) elts.elementAt (0);
  elts.setElementAt (c1, 0);
  elts.setElementAt (c2, i);
  return true;
}
```



Speeds up repeated membership tests

What's going on?
Mutates r to r', but does not change abstract value

© John Chapin/John Guttag Spring 2000 Slide 21