Lecture 5

Representation Invariants
Announcements
Announcements

• Happy Friday!
  – My t-shirt
• Next week
  – HW2 due Monday, July 4 at 10pm
    • It’s harder than HW1
    • Please take advantage of today’s office hours!
  – July 4 holiday on Wednesday -- no class!
    • TAs will announce OH changes
  – HW3 due Thursday
Motivation
Review

lec04

Method Specification
(abstraction)

lec05

Abstract Data Type
(abstraction)

Method Body
(concrete code)

Data Structure
(concrete code)

IMPLEMENTS

IMPLEMENTS
An ADT is a specification

**Abstract state** + collection of *procedural abstractions* (aka method specs)

- Not a collection of procedures

Together, these procedural abstractions provide some 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
- Observers: allow one to distinguish different values
An ADT has an abstract value

Abstract Value: An Int List is a finite sequence of integer values

```
size: 4
head
  Integer(1)
    Integer(2)
      Integer(42)
        Integer(17)
          null

size: 3
head
  Integer(1)
    Integer(2)
      Integer(42)
        Integer(17)
          null

size: 0
head
  null
    Integer(2)
      Integer(42)
        Integer(17)
          null
```

1, 2, 42, 17

???

??????
ADTs and specs

Values allowed by the Data Structure (concrete values)

Valid ADT Values (abstract values)

- from producers
- from value in ADT + operation
ADTs and specifications

So far, we have only specified ADTs
  - Specification makes no reference to the implementation

Of course, we need *guidelines for how* to implement ADTs
  • Data Structure implements an ADT

Of course, we need *guidelines for how* to ensure our implementations satisfy our specifications

Two intellectual tools are really helpful…
Connecting implementations to specs

**Representation Invariant**: maps Object → boolean
- Indicates if an instance is *well-formed*
- Defines the set of valid concrete values
- Only values in the valid set make sense as implementations of an abstract value
- For implementors/debuggers/maintainers of the abstraction: no object should ever violate the rep invariant
  - Such an object has no useful meaning

**Abstraction Function**: maps Object → abstract value
- What the data structure *means* as an abstract value
- How the data structure is to be interpreted
- Only defined on objects meeting the rep invariant
- For implementors/debuggers/maintainers of the abstraction: Each procedure should meet its spec (abstract values) by “doing the right thing” with the concrete representation
Representation

Invariants
Implementing a Data Abstraction (ADT)

To implement a data abstraction:

- Select the representation of instances, “the rep”
  - In Java, typically instances of some class you define
  - Implement operations in terms of that rep

Choose a representation so that:

- It is possible to implement required operations
- The most frequently used operations are efficient
  - But which will these be?
  - Abstraction allows the rep to change later
Example: CharSet Abstraction

// Overview: A CharSet is a finite mutable set of Characters
// @effects: creates a fresh, empty CharSet
public CharSet() {...}

// @modifies: this
// @effects: this\textsubscript{post} = this\textsubscript{pre} + \{c\}
public void insert(Character c) {...}

// @modifies: this
// @effects: this\textsubscript{post} = this\textsubscript{pre} - \{c\}
public void delete(Character c) {...}

// @return: (c \in this)
public boolean member(Character c) {...}

// @return: cardinality of this
public int size() {...}
An implementation: Is it right?

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();
    }
}
An implementation: Is it right?

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))
    System.out.print("wrong");
else
    System.out.print("right");
An implementation: Is it right?

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();
    }
}

Where is the error?
Where Is the Error?

If you can answer this, then you know what to fix

*Perhaps* `delete` is wrong
  – Should remove all occurrences?

*Perhaps* `insert` is wrong
  – Should not insert a character that is already there?

How can we know?
  – The `representation invariant` tells us
  – If it’s “our code”, this is how we document our choice for “the right answer”
The representation invariant

• Defines data structure well-formedness
• Must hold before and after every CharSet operation
• Operations (methods) may depend on it
• Write it like this:

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

Or, more formally (if you prefer):

\[ \forall \text{indices } i \text{ of this.elts }, \text{this.elts.elementAt}(i) \neq \text{null} \]
\[ \forall \text{indices } i, j \text{ of this.elts }, \]
\[ \text{this.elts.elementAt}(i).equals(\text{this.elts.elementAt}(j)) \Rightarrow i = j \]
The representation invariant

class CharSet {
    // Rep invariant:
    // this.elts has no nulls and no duplicates
    private List<Character> elts = ...
    ...
}

• Written in terms of the representation (this.elts)
• Internal comment (not javadoc)
  – located just inside of the class definition at the very beginning
Class CharSet

// Rep Invariant:
// for all indices i of this.elts,
//   this.elts.elementAt(i) is not null
// for all indices i, j of this.elts,
//   this.elts.elementAt(i).equals(this.elts.elementAt(j)) implies that i = j

private List<Character> elts = ...
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 example

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

Real-world constraints:
    • Balance ≥ 0
    • Balance = \sum_i \text{transactions.get}(i).amount

Implementation-related constraints:
    • Transactions ≠ null
    • No nulls in transactions
Checking rep invariants

Should code check that the rep invariant holds?
  – **Short answer:** YES!!!!

More considerations
  – Development vs. Production
    • Always yes in development [even when it’s expensive]
    • Production... usually yes
      – (See Pragmatic Programmer: Assertive Programming)
  – computational cost [depends on the invariant]
    • sometimes no for expensive checks, in production
  – Some private methods need not check (Why?)

A great debugging technique:
  Design your code to catch bugs early by implementing and using rep-invariant checking
Checking the rep invariant

Rule of thumb: check on entry \textit{and} on exit (why?)

/** Verify \textit{elts} has no nulls or duplicates and... */
private void checkRep() {
    for (int i = 0; i < elts.size(); i++) {
        assert elts.elementAt(i) != null;
        assert elts.indexOf(elts.elementAt(i)) == i;
    }
    ... // more checks
}

public void delete(Character c) {
    checkRep();
    elts.remove(c);
    // Is this guaranteed to get called?
    // (could guarantee it with a finally block)
    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
  – On exit:
    • Check rep invariant
    • Check postconditions

Checking the rep invariant helps you *discover* errors

Reasoning about the rep invariant helps you *avoid* errors
Summary so far...

• We implement an **Abstract Data Type** with a **Data Structure**
• Every Data Structure has a **Representation** that is a concrete way of representing an object’s **abstract value**

• The representation allows concrete values that do not correspond to an abstract value
  – “gray area” ...........................................

• **Representation Invariant** describes what makes the concrete representation valid (green area)
• **checkRep()** method verifies that the rep is valid, throws exception if not, protects you from yourself
• Check your rep all the time!
  – generally at beginning and end of every public method
Representation
Exposure
Exposure
Listing the elements of a CharSet

Consider adding the following method to CharSet

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

Consider this implementation:

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

Does the implementation of `getElts` preserve the rep invariant?
Listing the elements of a CharSet

Consider adding the following method to CharSet

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

Consider this implementation:

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

Does the implementation of `getElts` preserve the rep invariant?
Kind of, sort of, not really....
Consider this client code (outside the CharSet implementation):

```java
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

**A big deal, a common bug, you now have a name for it!**

If you do it, document why and how

– And feel guilty about it!
Avoiding representation exposure

*Understand* what representation exposure is

*Design* ADT implementations to make sure it doesn’t happen

Treat rep exposure as a bug: *fix* your bugs

*Test* for it with *adversarial clients*:
  - Pass values to methods and then mutate them
  - Mutate values returned from methods
private is not enough

- Making fields **private** does *not* suffice to prevent rep exposure
  - See our example
- So **private** is a hint to you: no aliases outside abstraction to references to mutable data reachable from **private** fields
- Two general ways to avoid representation exposure...
Avoiding rep exposure (way #1)

One way to avoid rep exposure is to make copies of all data that cross the abstraction barrier

- Copy in [parameters that become part of the implementation]
- Copy out [results that are part of the implementation]

Examples of copying (assume Point is a mutable ADT):

```java
class Line {
    private Point s, e;
    public Line(Point s, Point e) {
        this.s = new Point(s.x, s.y);
        this.e = new Point(e.x, e.y);
    }
    public Point getStart() {
        return new Point(this.s.x, this.s.y);
    }
    ...
}
```

Copy in

Copy out
Need deep copying

“Shallow” copying is not enough
  – Prevent any aliasing to mutable data inside/outside abstraction

What’s the bug (assuming `Point` is a mutable ADT)?
```java
class PointSet {
    private List<Point> points = ...
    public List<Point> getElts() {
        return new ArrayList<Point>(points);
    }
}
```

Not in example: Also need deep copying on “copy in”
Avoiding rep exposure (way #2)

One way to avoid rep exposure is to exploit the immutability of (other) ADTs the implementation uses

- Aliasing is no problem if nobody can change data
  - Have to mutate the rep to break the rep invariant

Examples (assuming Point is an immutable ADT):

```java
class Line {
    private Point s, e;
    public Line(Point s, Point e) {
        this.s = s;
        this.e = e;
    }
    public Point getStart() {
        return this.s;
    }
    ...
```
Why [not] immutability?

Several advantages of immutability

– Aliasing does not matter
– No need to make copies with identical contents
– Rep invariants cannot be broken
– Take CSE 341: Programming Languages for more!

Does require different designs (e.g., if Point immutable)

```java
void raiseLine(double deltaY) {
    this.s = new Point(s.x, s.y+deltaY);
    this.e = new Point(e.x, e.y+deltaY);
}
```

Immutable classes in Java libraries include String, Character, Integer, …
Deepness revisited

An immutable ADT must be immutable “all the way down”
  – No references *reachable* to data that may be mutated

So combining our two ways to avoid rep exposure:
  – Must copy-in, copy-out “all the way down” to immutable parts
Recall our initial rep-exposure example:

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

    // returns: elts currently in the set
    public List<Character> getElts() {
        return new ArrayList<Character>(elts);
    }
}
```
An alternative

```java
// returns: elts currently in the set
public List<Character> getElts() { // version 1
    return new ArrayList<Character>(elts); // copy out!
}

public List<Character> getElts() { // version 2
    return Collections.unmodifiableList<Character>(elts);
}
```

From the JavaDoc for `Collections.unmodifiableList`:

*Returns an unmodifiable view of the specified list. This method allows modules to provide users with "read-only" access to internal lists. Query operations on the returned list "read through" to the specified list, and attempts to modify the returned list… result in an UnsupportedOperationException.*
public List<Character> getElts() { // version 2
    return Collections.unmodifiableList<Character>(elts);
}

Clients cannot modify (mutate) the rep

• So they cannot break the rep invariant

(For long lists,) more efficient than copy out

Uses standard libraries
public List<Character> getElts() { // version 1
    return new ArrayList<Character>(elts); // copy out!
}

public List<Character> getElts() { // version 2
    return Collections.unmodifiableList<Character>(elts);
}

The two implementations do not do the same thing!
- Both avoid allowing clients to break the rep invariant
- Both return a list containing the elements

But consider:

```
xs = s.getElts();
s.insert('a');
xs.contains('a');
```

Version 2 is observing an exposed rep, leading to different behavior
Different specifications

Ambiguous spec: “returns a list containing the current set elements”

1. “returns a fresh mutable list containing the elements in the set at the time of the call”

versus

2. “returns read-only access to a list that the ADT continues to update to hold the current elements in the set”

3. A third spec weaker than both [but less simple and useful!]
   “returns a list containing the current set elements. *Behavior is unspecified (!) if* client attempts to mutate the list or to access the list after the set’s elements are changed”

Also note: Version 2’s spec also makes changing the rep later harder
   – Only “simple” to implement with rep as a List
Summary

- A data structure’s representation allows concrete values that do not correspond to an abstract value allowed by the ADT
  - “gray area” -------→

- **Representation Invariant** describes what makes the concrete representation valid (green area)

- **checkRep()** method verifies that the rep is valid

- **Rep Exposure** occurs when a client can modify the rep
  - Never let this happen!!!!
Closing
Closing Announcements

• HW2 due Monday 10 pm
  – Start early!
  – Go to office hours today!

• Thank you for coming to class!

• Enjoy your weekend!