CSE 331
Software Design & Implementation

Kevin Zatloukal
Spring 2020
ADT Implementation: Representation Invariants
Data abstraction outline

ADT specification

Abstract States

Abstraction barrier

Abstraction function (AF): Relationship between ADT specification and implementation

ADT implementation

Fields in our Java class
Last time: abstraction function

• Allows us to check correctness
  – use reasoning to show that the method leaves the abstract state
    such that it satisfies the postcondition

```java
// AF(this) = vals[0..len-1]
private int[] vals;
private int len;

//@requires length > 0
//@modifies this
//@effects this = this[0..length-2]
public void pop() { ... }
```
Last time: abstraction function

- Allows us to check correctness
  - use reasoning to show that the method leaves the abstract state such that it satisfies the postcondition

```java
// AF(this) = vals[0..len-1]

// @requires length > 0
// @modifies this
// @effects this = this[0..length-2]
public void pop() {
  {{ length > 0 }}
  len = len - 1;
  {{ this = this\_pre[0 .. length\_pre – 2] }}
} => {{ this = vals[0..len-1] = vals[0..len\_pre-2] }}
```
Abstract

States

Fields in our Java class

Abstraction function (AF):
Relationship between ADT specification and implementation

Abstraction Barrier

Representation invariant (RI):
Relationship among implementation fields

ADT specification

ADT implementation
Connecting implementations to specs

For implementers / debuggers / maintainers of the implementation:

**Representation Invariant**: maps Object → boolean
- defines the set of valid concrete values
- must hold at all times (outside of mutators)
- no object should ever violate the rep invariant
  - such an object has no useful meaning

**Abstraction Function**: maps Object → abstract state
- says what the data structure *means* in vocabulary of the ADT
- only defined on objects meeting the rep invariant
Example: Circle

/** Represents a mutable circle in the plane. For example, * it can be a circle with center (0,0) and radius 1. */
public class Circle {

    // Rep invariant: center != null and rad > 0
    private Point center;
    private double rad;

    // Abstraction function:
    // AF(this) = a circle with center at this.center
    // and radius this.rad

    // ...
}
/** Represents a mutable circle in the plane. For example, * it can be a circle with center (0,0) and radius 1. */

public class Circle {

    // Rep invariant: center != null and edge != null
    // and !center.equals(edge)
    private Point center, edge;

    // Abstraction function:
    // AF(this) = a circle with center at this.center
    // and radius this.center.distanceTo(this.edge)

    // ...

}
Example: CharSet ADT

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

// @modifies: this
// @effects: this changed to this + {c}
public void insert(Character c) {...}

// @modifies: this
// @effects: this changed to this - {c}
public void delete(Character c) {...}

// @return: true iff c is in this set
public boolean member(Character c) {...}
// @return: cardinality of this set
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:

class CharSet {
    private List<Character> elts = new ArrayList<>();

    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.println("wrong");
else
    System.out.println("right");

Where is the error?
Where Is the Error?

• *Perhaps* insert *is wrong*
  – should not insert a character that is already there?

• *Perhaps* delete (and size) *is wrong*
  – should remove all occurrences?

• In this case, the representation invariant tells us which is correct
  – RI says if character can appear multiple times or not
  – this is how we document our choice for “the right answer”
    • again, good invariant makes the code write itself...
The representation invariant

Write it like this:

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

    Or, more formally (if you prefer):
    for all indices i of els, we have els.elementAt(i) ≠ null
    for all indices i, j of els with i != j,
    we have ! els.elementAt(i).equals(els.elementAt(j))
```
The representation invariant

Write it like this:

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

    // Must hold before and after every CharSet operation
    // Methods may assume it implicitly (along with @requires)
    //   - no need to state your assumption that RI holds
```
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);
}
Example: Polynomial

/** An immutable polynomial with integer coefficients. * Examples include 0, 2x, and x + 3x^2 + 5x. */
public class IntPoly {

    // Rep invariant: coeffs != null and
    //(coeffs[coeffs.length-1] != 0
    private final int[] coeffs;

    // Abstraction function:
    // AF(this) = sum of this.coeffs[i] * x^i
    // for i = 0 .. this.coeffs.length

    /** Returns the highest exponent with nonzero coefficient
     * or zero if none exists. */
    public int degree() {
        ...
    }

    // ...
}
Example: Polynomial 2

/** An immutable polynomial with integer coefficients.
   * Examples include 0, 2x, and x + 3x^2 + 5x. */
public class IntPoly {

   // Rep invariant: terms != null and
   //   terms is sorted by degree and
   //   no two terms have the same degree
   private final List<IntTerm> terms;

   // Abstraction function:
   // AF(this) = sum of monomials in this.terms

   /** Returns the highest exponent with nonzero coefficient
        * or zero if none exists. */
   public int degree() { ... }

   // ...
Checking rep invariants

Should you write code to check that the rep invariant holds?

- Yes, if it’s inexpensive [depends on the invariant]
- Yes, for debugging [even when it’s expensive]
- Often hard to justify turning the checking off
  • better argument is removing clutter (improve understandability)
- Some private methods need not check (Why?)

A great debugging technique:

*Design your code to catch bugs by implementing and using a function to check the rep-invariant*
Checking the rep invariant

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

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

    // Is this guaranteed to get called?
    // (could guarantee it with a finally block)
    checkRep();
}
```

```java
/** Verify that elts contains no duplicates. */
private void checkRep() {
    for (int i = 0; i < elts.size(); i++) {
        assert elts.indexOf(elts.elementAt(i)) == i;
    }
}
```
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
private List<Character> elts;
public List<Character> getElts() { return elts; }
```

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

*Can’t say!*
Representation exposure

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 **bad**
  - can cause bugs that will be **very hard to detect**
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);
    }
  }
  ```
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)?
  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 via exposure
  – see CSE341 for more!

• Does require different code (e.g., if Point immutable)
  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, redux

- 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:

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); //copy out!
    }
    ...
}

Back to getElts
Alternative #3

// 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(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.
Suggestions

Best options for implementing `getElts()`

- if $O(n)$ time is acceptable for relevant use cases, copy the list
  - safest option
  - best option for changeability

- if $O(1)$ time is required, then return an unmodifiable list
  - prevents breaking rep invariant
  - clearly document that behavior is unspecified after mutation
  - ideally, write a your own unmodifiable view of the list
    that throws an exception on all operations after mutation

- if $O(1)$ time is required and there is no unmodifiable version and
  you don’t have time to write one, expose rep and feel guilty