CSE 331
Software Design & Implementation

James Wilcox & Kevin Zatloukal
Fall 2022
ADT Implementation: Representation Invariants
Specifying an ADT

Different types of methods:

1. creators
2. observers
3. producers
4. mutators (if mutable)

Described in terms of how they change the **abstract state**
- abstract description of what the object means
  - difficult (unless concept is already familiar) but vital
  - specs have no information about concrete representation
    - leaves us free to change those in the future
Implementing a Data Abstraction (ADT)

To implement an ADT:
- select the representation of instances
- implement operations in terms of that representation

Choose a representation so that:
- it is possible to implement required operations
- the most frequently used operations are efficient / simple / …
  - abstraction allows the rep to change later
  - almost always better to start simple

Then use reasoning to verify the operations are correct
- two intellectual tools are helpful for this...
Abstract
States

Fields in our
Java class

Abstraction Barrier

Abstraction Function (AF):
 mapping between ADT
implementation and specification
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] }}
} ⇒ {{ len > 0 }}
    {{ len = len_pre - 1 }}
    {{ this = vals[0..len-1] = vals[0..len_pre-2] }}
```
Abstract

States

Fields in our Java class

Abstraction Barrier

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

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 before and after any public method is called
- **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
/** 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: Polynomial

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

public class IntPoly {

    // Rep invariant: coeffs != null
    private final int[] coeffs;

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

    // ... coeff, degree, etc.
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
    // no two terms have the same degree and
    // terms is sorted in descending order by degree
    private final LinkedList<IntTerm> terms;

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

    // ... coeff, degree, etc.

}
Example: IntStack

/** List that only allows insert/remove at right end. */
public class IntStack {

    // RI: vals != null and 0 <= len <= vals.length
    // AF(this) = vals[0.. len-1]
    private int[] vals;
    private int len;

Another example

class Account {
    private int balance;

    // history of all transactions
    private List<Transaction> transactions;

    ...
}

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

Real-world constraints:
• Balance = Σ_i transactions.get(i).amount
• Balance ≥ 0
Defensive Programming with ADTs
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 must not check

A great debugging technique:

*Design your code to catch bugs by implementing and using a function to check the rep-invariant*
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() {...}
Example: CharSet ADT

// Rep invariant: elts != null and
// elts has no nulls and no dups
// AF(this) = list of chars in elts
private List<Character> elts;
Checking the rep invariant

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

```java
public void delete(Character c) {
    checkRep();
    elts.remove(c); // removes 0 or 1 copies of c
    checkRep();
}

// Verify that elts contains no nulls or dups
private void checkRep() {
    for (int i = 0; i < elts.size(); i++) {
        assert elts.get(i) != null;
        assert elts.indexOf(elts.get(i)) == i;
    }
}
```
Practice *defensive programming*

- Question is not: will you make mistakes? You will.
- Question is: will you **catch** those mistakes before users do?

- Write and incorporate code designed to catch the errors you make
  - check rep invariant on entry and exit (of mutators)
  - check preconditions (don’t trust other programmers)
  - check postconditions (don’t trust yourself either)

- Checking the rep invariant helps **discover** errors while testing
- Reasoning about the rep invariant helps **discover** errors while coding
Practice *defensive programming*

- Checking pre- and post-conditions and rep invariants is one tip
- More of these in Effective Java
  - first required reading (see calendar for items)

- Focus on defensive programming against *subtle bugs*
  - obvious bugs (e.g., crashing every time) will be caught in testing
  - subtle bugs that only occasionally cause problems can sneak out
  - be especially defensive against (and scared of) these
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
public List<Character> getElts() { return elts; }
```

Does this implementation 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**

• Rule #1: Don’t do it!
• Rule #2: If you do it, document it clearly and then 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
  - absolutely must avoid in libraries with many clients
  - can allow (but feel guilty) for code with few clients
- *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
  - issue is *aliasing of mutable data outside the abstraction*

- So `private` is a hint to you: no aliases outside abstraction to references to mutable data reachable from `private` fields
- Three 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);
    }
    ...
  }
```
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;
      }
  }
  ...
  ```
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.
The good news

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

- Clients cannot modify (mutate) the rep
  - cannot break the rep invariant
- (For long lists,) more efficient than copy out
- Uses standard libraries
```
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

Ambiguity of “returns a list containing the current set elements”

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

versus

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

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
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