
CSE 331

Software Design & Implementation

Hal Perkins

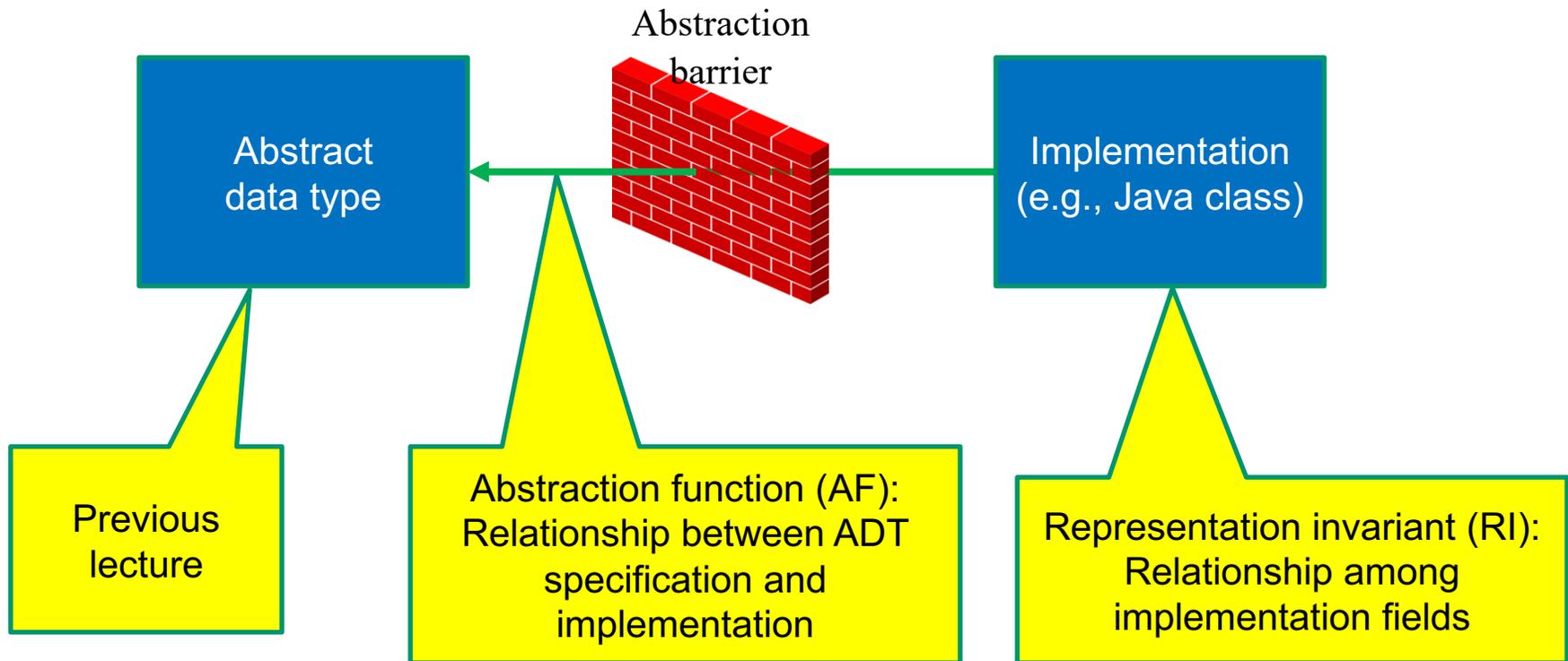
Winter 2021

Representation Invariants

Data abstraction outline

ADT specification

ADT implementation



Review: a data abstraction is defined by a specification

A collection of procedural *abstractions*

- 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 (affects `equals (...)` but not `==`)
 - Observers: allow the client to distinguish different values

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
- 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 \rightarrow boolean

- An assertion about the object state
- 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 \rightarrow 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

Implementing a Data Abstraction (ADT)

To implement a data abstraction:

- Select the representation of instances, “*the rep*”
 - In Java, typically instance variables in objects of a class
- 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 new, empty CharSet
public CharSet() {...}

// @modifies: this
// @effects: thispost = thispre + {c}
public void insert(Character c) {...}

// @modifies: this
// @effects: thispost = thispre - {c}
public void delete(Character c) {...}

// @return: (c ∈ 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();
    }
}
```

```
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");
```

Where is the defect?

Where Is the defect?

- Answer this and 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

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 example:

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

Or, more formally (if you prefer):

\forall indices i of `elts` . `elts.elementAt(i) \neq null`

\forall indices i, j of `elts` .

$i \neq j \Rightarrow \neg \text{elts.elementAt}(i).\text{equals}(\text{elts.elementAt}(j))$

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 example

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

Rep invariants often contain both problem domain and internal implementation parts. For this example:

- Real-world constraints:
 - $\text{balance} \geq 0$
 - $\text{balance} = \sum_i \text{transactions.get}(i).\text{amount}$
- Implementation-related constraints:
 - $\text{transactions} \neq \text{null}$
 - No nulls in transactions

A rep invariant is a pre/postcondition

- For any public ADT operation (method) the triple {rep invariant} method body {rep invariant} should be a valid Hoare triple
 - (Not necessarily true for private “helper” methods that execute while the rep is being updated)
- For constructors, the {rep invariant} is the constructor’s postcondition, but not part of the precondition
 - The constructor establishes the rep invariant for a newly created object by initializing it; rep inv doesn’t hold until that is done
- Our proof techniques, especially forward reasoning, can be helpful to check that the rep invariant is preserved by an ADT operation (or established by a constructor)

Checking rep invariants

Should code 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
- Some private methods need not check (Why?)
- Some private methods should not check (Why?)

A great debugging technique:

Design your code to catch bugs by implementing and using rep-invariant checking

Checking the rep invariant

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

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

    // Is this guaranteed to get called?
    // (could guarantee it with a finally block)
    checkRep();
}
...
/** Verify that elts contains no duplicates. */
private void checkRep() {
    for (int i = 0; i < elts.size(); i++) {
        assert elts.indexOf(elts.elementAt(i)) == i;
    }
}
```

Practice *defensive programming*

- Assume that you will make mistakes
- Write and incorporate code designed to catch them when feasible
 - On entry:
 - Check rep invariant
 - Check other preconditions
 - On exit:
 - Check rep invariant
 - Check other postconditions
- Checking the rep invariant helps you *discover* errors
- Reasoning about the rep invariant helps you *avoid* errors

New Question:

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 and no dups
public List<Character> getElts() { return elts; }
```

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

Kind of, sort of, not really....

Representation exposure

Consider this client code (outside the `CharSet` implementation):

```
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**
 - Allows violation of abstraction boundaries and rep invariant
 - *A big deal, a common bug, you now have a name for it!*
- If you do it (should be rare), document how and why
 - And feel guilty about it!

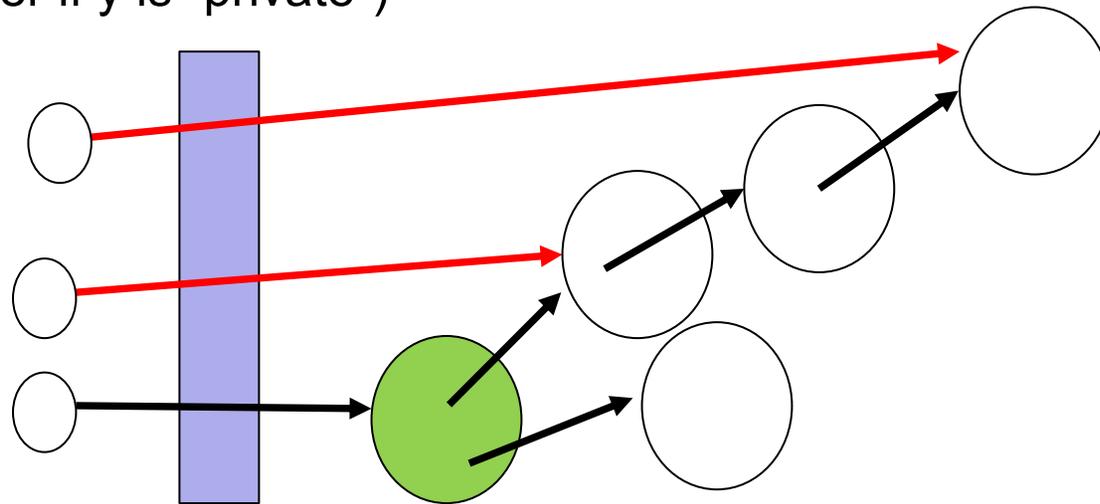
Avoiding representation exposure

The first step for getting help is to recognize you have a problem 😊

- *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
 - Check the rep invariant in addition to client behavior

private is not enough

- Making fields **private** does *not* suffice to prevent rep exposure
 - Issue is **aliasing of mutable data inside and outside the abstraction** (remember, after assignment $x=y$; x and y both refer to the **same** object! If clients can access x , it doesn't matter if y is "private")



- **private** is a hint: be sure you don't create aliases that let clients reference mutable data reachable from **private** fields
 - And be sure to use **private** to prevent direct access to rep

Avoiding rep exposure #1: immutability

- Exploit the **immutability** of (other) ADTs the implementation uses
 - Aliasing is no problem if client cannot change data
- Examples (assuming `Point` is an *immutable* ADT):

```
class Line {
    private Point start, end;
    public Line(Point start, Point end) {
        this.start = start;
        this.end = end;
    }
    public Point getStart() {
        return this.start;
    }
    ...
}
```

Why [not] immutability?

- Immutability greatly simplifies reasoning
 - Aliasing does not matter
 - No need to make copies with identical contents
 - Rep invariants cannot be broken

- Does require different designs

Suppose `Point` is immutable but `Line` is mutable:

```
void raiseLine(double deltaY) {
    this.start =
        new Point(start.x, start.y+deltaY);
    this.end =
        new Point(end.x, end.y+deltaY);
}
```

- Immutable classes in Java libraries include `String`, `Character`, `Integer`, ...

Avoiding rep exposure #2: copying

- 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):

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

Shallow copying is not enough

- Example: assume `Point` and `Line` are mutable ADTs

```
class Line {  
    private Point start;  
    private Point end;  
  
    public Line(Line other) {  
        this.start = other.start;  
        this.end = other.end;  
    }  
}
```

- Client code:

```
Line a = ...;  
Line b = new Line(a); // a and b share Points  
a.translate(3, 4); // also translates b!  
...
```

Full deep copy is not always needed

- 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

Back to getElts

Our initial rep-exposure problem, fixed now with copy-out :

```
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!
    }
    ...
}
```

Avoiding rep exposure #3: readonly wrapper (immutable “copy”)

```
public List<Character> getElts () {  
    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

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

The bad news

```
public List<Character> getElts() {  
    return new ArrayList<Character>(elts); //copy out!  
}
```

```
public List<Character> getElts() {  
    return Collections.unmodifiableList(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

“returns a list containing the elements”

Could mean any of these things:

1. Returns a fresh mutable list containing the elements in the set *at the time of the call*
 - likely hard to implement efficiently
2. Returns read-only view that is *always up to date* with the current elements of the set
 - Makes it hard to change the rep
3. 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
 - Weaker than either #1 or #2
 - More complex, harder to use, but sufficient for some purposes

Lesson: a seemingly simple spec may be ambiguous and subtle

Next...

- Abstraction functions: mapping concrete representation to abstract values.