Implementing an ADT: Representation invariants and abstraction functions

CSE 331
University of Washington

Michael Ernst

A data abstraction is defined by a specification

- A collection of procedural abstractions
  - Not a collection of procedures
- Together, these procedural abstractions provide
  - A set of values
  - ***<u>All</u>*** the ways of directly using that set of values
    - Creating
    - Manipulating
    - Observing
- Creators and producers: make new values
- Mutators: change the value (but don't affect **==**)
- Observers: allow one to tell values apart

ADTs and specifications

Specification: only in terms of the abstraction

- Never mentions the representation

An ADT is more than just a data structure

- data structure + a set of conventions

Why do we need to relate the specification to the representation?

Connecting specifications and implementations

Representation invariant: Object → boolean

- Indicates whether a data structure is well-formed
  - Only well-formed representations are meaningful
- Defines the set of valid values of the data structure

Abstraction function: Object → abstract value

- What the data structure means (as an abstract value)
- How the data structure is to be interpreted
- How do you compute the inverse, abstract value → Object ?

Implementation of an ADT is provided by a class

To implement a data abstraction:

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

Choose a representation so that

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

CharSet Abstraction

```
// Overview: A CharSet is a finite mutable set of Characters

// effects: creates a fresh, empty CharSet
public CharSet ( )

// modifies: this
// effects: this_post = this_pre U {c}
public void insert (Character c);

// modifies: this
// effects: this_post = this_pre - {c}
public void delete (Character c);

// returns: (c ∈ this)
public boolean member (Character c);

// returns: cardinality of this
public int size ( );
```

A CharSet implementation.

What client code will expose the error?

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

  public void insert(Character c) {
    elts.add(c);
  }
  public void delete(Charact
    elts.remove(c);
  }
  public boolean member(Char
    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))
    // print "wrong";
else
    // print "right";
```

Where Is the Error?

The answer to this question tells you what needs to be fixed

Perhaps **`delete`** is wrong

- It should remove all occurrences

Perhaps **`insert`** is wrong

- It should not insert a character that is already there

How can we know?

- The representation invariant tells us

The representation invariant

- States data structure well-formedness
- Holds before and after every CharSet operation
- Operation implementations (methods) may depend on it

Write it this way:

```
class CharSet {
  // Rep invariant: elts has no nulls and no duplicates
  private List<Character> elts;
  …
```

Or, if you are the pedantic sort:

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

$\forall$ indices i, j of elts .

i $\neq$ j $\Rightarrow \neg$ elts.elementAt(i).equals(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 rep invariant example

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

// real-world constraints:

balance $\geq$ 0

balance = Σ_i transactions.get(i).amount

// implementation-related constraints:

transactions $\neq$ null

no nulls in transactions

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 duplicates
public List<Character> getElts() { return elts; }
```

Does the implementation of getElts preserve the rep invariant?

... sort of

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

- Enables violation of abstraction boundaries and the rep invariant

If you do it, document why and how

- And feel guilty about it!

Ways to avoid rep exposure

Defining fields as **`private`** is not sufficient to hide the representation

1. Exploit immutability

```
Character choose() {
  return elts.elementAt(0);
}
```

Character is immutable.

2. Make a copy

```
List<Character> getElts() {
  return new ArrayList<Character>(elts);
  // or: return (ArrayList<Character>) elts.clone();
}
```

Mutating a copy doesn’t affect the original.

Don’t forget to make a copy on the way in!

3. Make an immutable copy

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

Client cannot mutate

Still need to make a copy on the way in

Checking rep invariants

Should code check that the rep invariant holds?

- Yes, if it's inexpensive
- Yes, for debugging (even when it's expensive)
- It's quite hard to justify turning the checking off
- Some private methods need not check (Why?)

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?
  // See handouts for a less error-prone way to check at exit.
  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

- On entry:
  - Check rep invariant
  - Check preconditions (<u>requires</u> clause)
- On exit:
  - Check rep invariant
  - Check postconditions

Checking the rep invariant helps you discover errors

Reasoning about the rep invariant helps you avoid errors

- Or prove that they do not exist!
- We will discuss such reasoning, later in the term

The rep invariant constrains structure, not meaning

New implementation of insert that <u>preserves the rep invariant</u>:

```
public void insert(Character c) {
  Character cc = new Character(encrypt(c));
  if (!elts.contains(cc))
    elts.addElement(cc);
}
public boolean member(Character c) {
  return elts.contains(c);
}
```

The program is still wrong

- Clients observe incorrect behavior
- What client code exposes the error?
- Where is the error?
- We must consider the meaning
- The *abstraction function* helps us

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

Abstraction function: rep → abstract value

The abstraction function maps the concrete representation to the abstract value it represents

AF: Object → abstract value

AF(CharSet this) = { c | c is contained in this.elts }

“set of Characters contained in this.elts”

Typically *not* executable

The abstraction function lets us reason about behavior from the client perspective

Abstraction function and insert impl.

Our real goal is to satisfy the specification of insert:

// modifies: this
// effects: $this_{post} = this_{pre} \cup \{c\}$
public void insert (Character c);

The AF tells us what the rep means (and lets us place the blame)

AF(CharSet this) = { c | c is contained in this.elts }

Consider a call to insert:

On entry, the meaning is $AF(this_{pre}) \approx elts_{pre}$

On exit, the meaning is $AF(this_{post}) = AF(this_{pre}) \cup \{encrypt(‘a’)\}$

What if we used this abstraction function?

AF(this) = { c | encrypt(c) is contained in this.elts }

= { decrypt(c) | c is contained in this.elts }

Stack example

Stack rep:

```
int[] elements;
int top;  // first unused index
```

`new Stack()`

0	0	0

Top=0

`stack = <>`

`push(17)`

17	0	0

Top=1

`stack = <17>`

`push(-9)`

17	-9	0

Top=2

`stack = <17,-9>`

`pop()`

17	-9	0

Top=1

`stack = <17>`

Abstract states are the same

`stack = <17> = <17>`

Concrete states are different

`<[17,0,0], top=1>`

≠

`<[17,-9,0], top=1>`

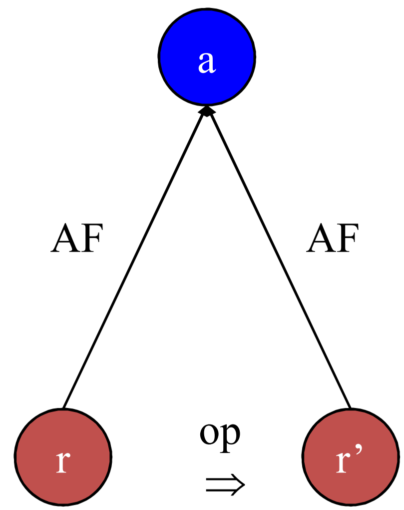
AF is a function

AF^{-1} is not a function

Benevolent side effects

Different implementation of member:

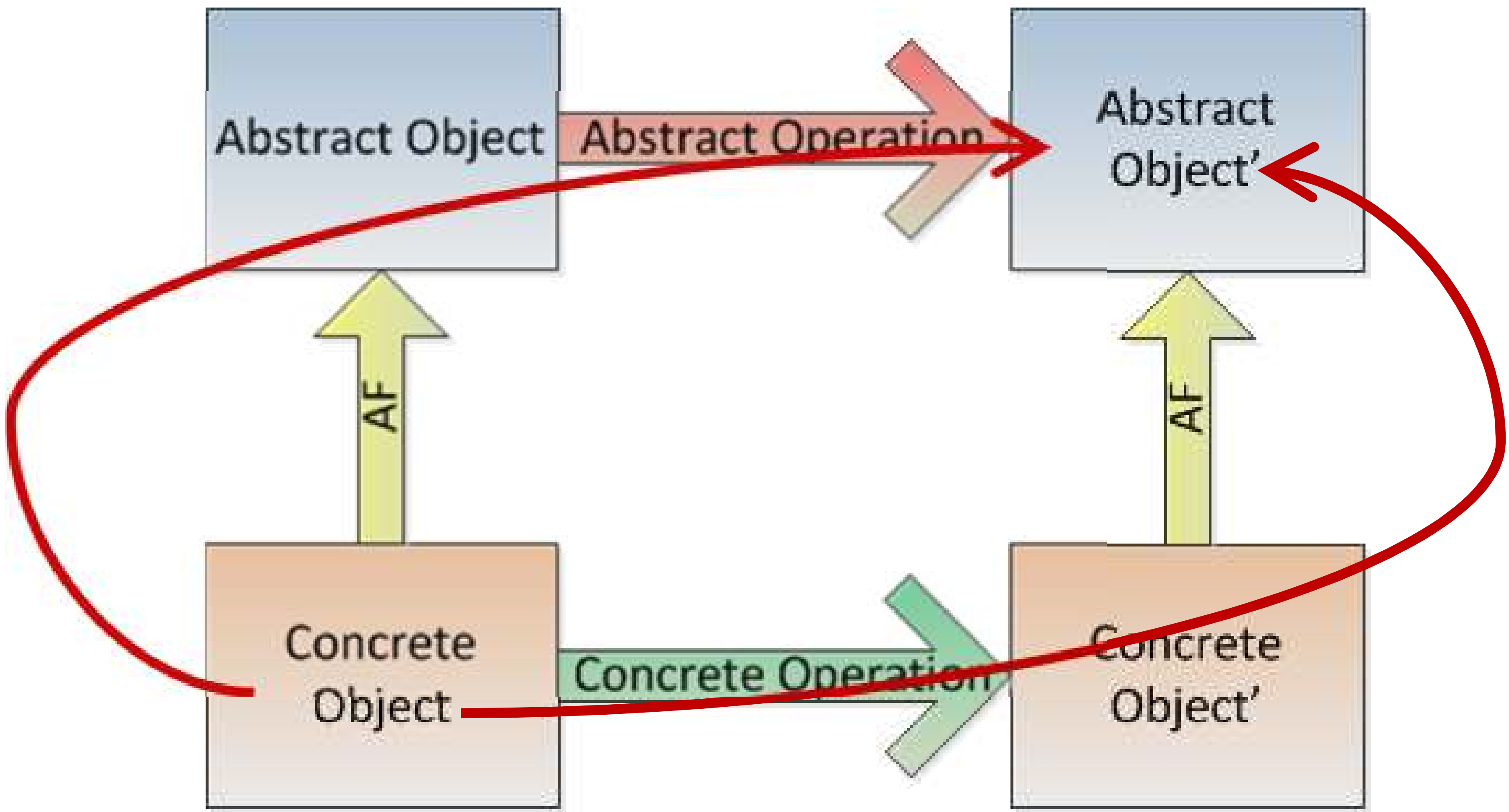
```
boolean member(Character c1) {
  int i = elts.indexOf(c1);
  if (i == -1)
    return false;
  // move-to-front optimization
  Character c2 = elts.elementAt(0);
  elts.set(0, c1);
  elts.set(i, c2);
  return true;
}
```



Move-to-front speeds up repeated membership tests

Mutates rep, but does not change *abstract* value

- AF maps both reps to the same abstract value
- Example: AF(a u c t i o n) = { a, c, i, n, o, t, u } = AF(c a u t i o n)
- Example: AF(s h r u b) = { b, h, r, s, u } = AF(b r u s h)



Creating the concrete object:

- Establishes the rep invariant
- Establishes the abstraction function

Every operation:

- Maintains the rep invariant
- Maintains the abstraction function

Why is each of these properties important?

The abstraction function: concrete → abstract

Q: Why do we map concrete to abstract rather than vice versa?

1. It’s not a function in the other direction.

   E.g., lists [a,b] and [b,a] each represent the set {a, b}

2. It’s not as useful in the other direction.

   Can construct objects via the provided operators

Writing an abstraction function

The domain: all representations that satisfy the rep invariant

The range: can be tricky to denote

- For mathematical entities like sets: easy
- For more complex abstractions: give them fields
  - AF defines the value of each “specification field”
  - For “derived specification fields”, see the handouts

The overview section of the specification should provide a way of writing abstract values

- A printed representation is valuable for debugging

ADTs and Java language features

- Java classes
  - Make operations in the ADT public
  - Make other ops and fields of the class private
  - Clients can only access ADT operations
- Java interfaces
  - Clients only see the ADT, not the implementation
  - Multiple implementations have no code in common
  - Cannot include creators (constructors) or fields
- Both classes and interfaces are sometimes appropriate
  - Write and rely upon careful specifications
  - Prefer interface types instead of specific classes in declarations (e.g., **`List`** instead of **`ArrayList`** for variables and parameters)

Summary

- Rep invariant
  - Which concrete values represent abstract values
- Abstraction function
  - For each concrete value, which abstract value it represents
- Together, they modularize the implementation
  - Can examine operators one at a time
  - Neither one is part of the abstraction (the ADT)
- In practice
  - Always write a representation invariant
  - Write an abstraction function when you need it
    - Write an informal one for most non-trivial classes
    - A formal one is harder to write and often less useful

A half-step backwards

- Why focus so much on invariants (properties of code that do not – or are not supposed to – change)?
- Why focus so much on immutability (a specific kind of invariant)?

- Software is complex – invariants/immutability reduce the intellectual complexity
- If we can assume some property remains unchanged, we can consider other properties instead
- Reducing what we need to think about can be a huge benefit