
CSE 331

Software Design & Implementation

Hal Perkins

Winter 2013

Data Abstraction: Abstract Data Types (ADTs)

(Based on slides by Mike Ernst and David Notkin)

Outline

First:

Data Abstraction – ADTs

ADT specification and Implementation

Then: Reasoning about data abstractions

Representation Invariants (RIs)

Abstraction Functions (AFs)

Review: Satisfaction of a specification

Let P be an implementation and S a specification

Think “procedures/methods/functions” for the moment

P satisfies S iff

Every behavior of P is permitted by S

“The behavior of P is a subset of S ”

The statement “ P is correct” is meaningless

Though often made!

If P does not satisfy S , either (or both!) could be “wrong”

“One person’s feature is another person’s bug.”

It’s usually better to change the program than the spec

Scaling Up Specifications

Procedural abstraction:

- Abstracts from details of procedures

- A specification mechanism

- Satisfy the specification with an implementation

Data abstraction:

- Abstracts from details of data representation

- A specification mechanism

 - A way of thinking about programs and design

- Standard terminology: **Abstract Data Type**, or **ADT**

Why we need Data Abstractions (ADTs)

Organizing and manipulating data is pervasive

Inventing and describing algorithms is rare

Start your design by **designing data structures**

Potential problems with choosing a data abstraction:

Decisions about data structures often made too early

Duplication of effort in creating derived data

Very hard to change key data structures

A Data Abstraction is a set of operations

ADT abstracts from the **organization** to **meaning** of data

ADT abstracts from **structure** to **use**

Representation does not matter; this choice is (or should be) irrelevant to the client:

```
class RightTriangle {  
    float base, altitude;  
}
```

```
class RightTriangle {  
    float base, hypot, angle;  
}
```

Instead, think of a type as a set of operations

create, getBase, getAltitude, getBottomAngle, ...

Force clients (users) to use operations to access data

Are these classes the same?

```
class Point {  
    public float x;  
    public float y;  
}
```

```
class Point {  
    public float r;  
    public float theta;  
}
```

Different: can't replace one with the other

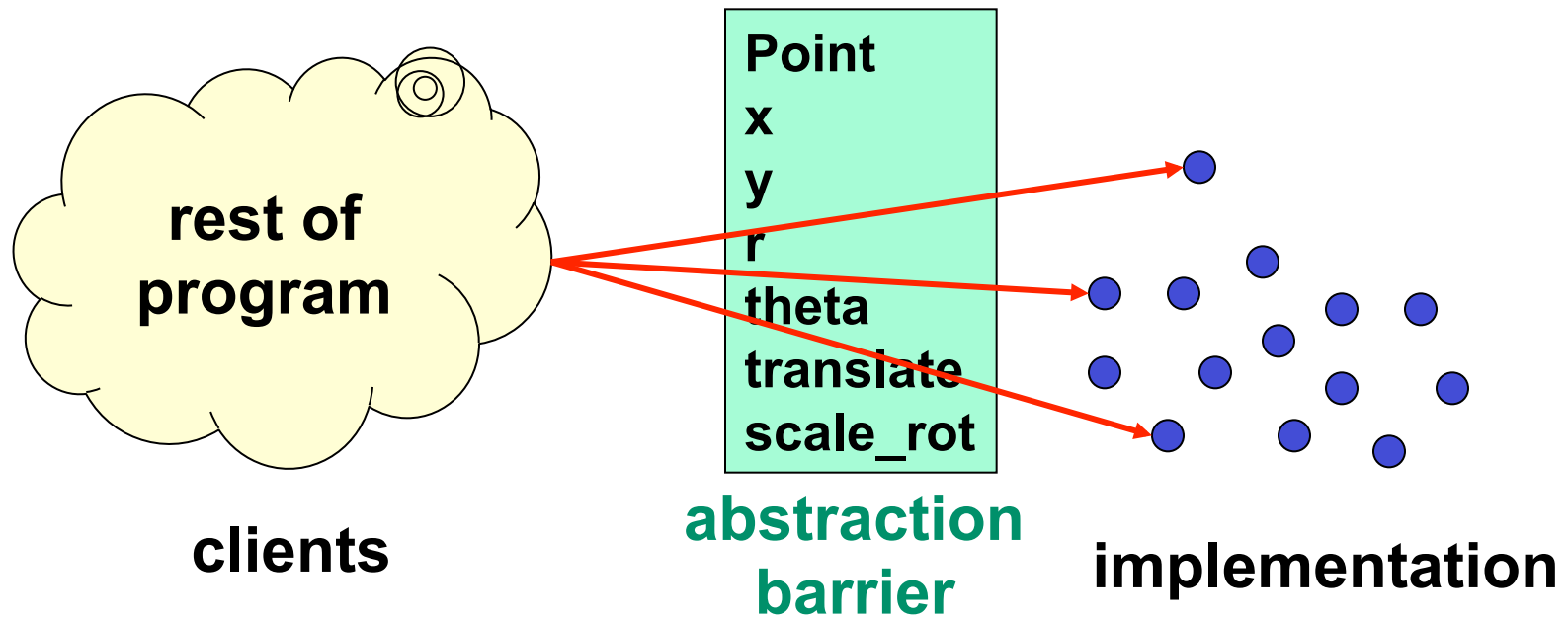
Same: both classes implement the concept "2-d point"

Goal of ADT methodology is to express the sameness:

- Clients depend only on the concept "2-d point"

- Can delay implementation decisions, fix bugs, change algorithms without affecting clients

Abstract data type = objects + operations



The implementation is hidden

The only operations on objects of the type are those provided by the abstraction

Concept of 2-d point, as an ADT

```
class Point {  
    // A 2-d point exists somewhere in the plane, ...  
  
    public float x();  
    public float y();  
    public float r();  
    public float theta();  
  
    // ... can be created, ...  
    public Point(); // new point at (0,0)  
    public Point centroid(Set<Point> points);  
  
    // ... can be moved, ...  
    public void translate(float delta_x,  
                          float delta_y);  
    public void scaleAndRotate(float delta_r,  
                               float delta_theta);  
}
```

Observers

Creators/
Producers

Mutators

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

All 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

Connecting specifications and implementations

Specification: describes ADT only in terms of the abstraction

Never mentions the representation

Abstraction Function: maps object \rightarrow abstract value

What the data structure *means* as an abstract value

How the data structure is to be interpreted

Ex: point in the plane represented by Point object

Representation Invariant: maps object \rightarrow boolean

Indicates whether a data structure is *well-formed*

Defines set of valid values of the data structure

Only well-formed representations (values) make sense as implementations of an abstract value

Implementing a Data Abstraction (ADT)

To implement a data abstraction

- Select the representation of instances, the “rep”

- Implement operations in terms of that rep

 - In Java this is typically done with a class

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 an empty CharSet

public CharSet ()

// modifies: this

// effects: $this_{post} = this_{pre} \cup \{c\}$

public void insert (Character c);

// modifies: this

// effects: $this_{post} = this_{pre} - \{c\}$

public void delete (Character c);

// returns: ($c \in this$)

public boolean member (Character c);

// returns: cardinality of this

public int size ();

A CharSet implementation: Is it OK?

```
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))
    // print "wrong";
else
    // print "right";
```

Where Is the Error?

Answer this and you know what to fix

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

Must hold before and after every `CharSet` operation

Operations (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, more formally:

\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);  
}
```

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 the 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**

If you do it, document why and how

And feel guilty about it! (even if you have to do it)

Ways to avoid rep exposure

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?
    // (there are ways to guarantee it)
    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 (requires 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!

Rep inv. constrains structure, not meaning

New implementation of insert that preserves the rep invariant:

```
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: $\text{rep} \rightarrow \text{abstract value}$

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

AF: $\text{Object} \rightarrow \text{abstract value}$

$\text{AF}(\text{CharSet this}) = \{ c \mid c \text{ is contained in } \text{this.elts} \}$

“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

Our real goal is to satisfy the **specification of insert**:

```
// modifies: this  
// effects: thispost = thispre U {c}  
public void insert (Character c);
```

Once again we can place the blame

Applying the abstraction function to the result of the call to insert yields $AF(\text{elts}) \cup \{\text{encrypt}('a')\}$

What if we used this abstraction function?

$$\begin{aligned} AF(\text{this}) &= \{ c \mid \text{encrypt}(c) \text{ is contained in } \text{this.elts} \} \\ &= \{ \text{decrypt}(c) \mid c \text{ is contained in } \text{this.elts} \} \end{aligned}$$

Data Abstraction: 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 usually less useful

Next time: more examples and perspective