CSE 331
Software Design & Implementation

Hal Perkins

Winter 2012

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 ADTs

- 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 and operations on that data
  - A way of thinking about programs and design
  - Satisfy the specification with an implementation
  - Standard terminology: Abstract Data Type, or ADT

Why we need Abstract Data Types

- Organizing and manipulating data is pervasive
  - Inventing and describing algorithms is rare
- Often most important place to start a design is by designing data structures/abstractions
- Potential problems with choosing a data abstraction:
  - Decisions about data structures often made too early
  - Very hard to change key data structures

An ADT is a set of operations

- ADT abstracts from a specific representation to focus on the semantic meaning of the data
- 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 {                class Point {
  public float x;              public float r;
  public float y;              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

Concept of 2-d point, as an ADT

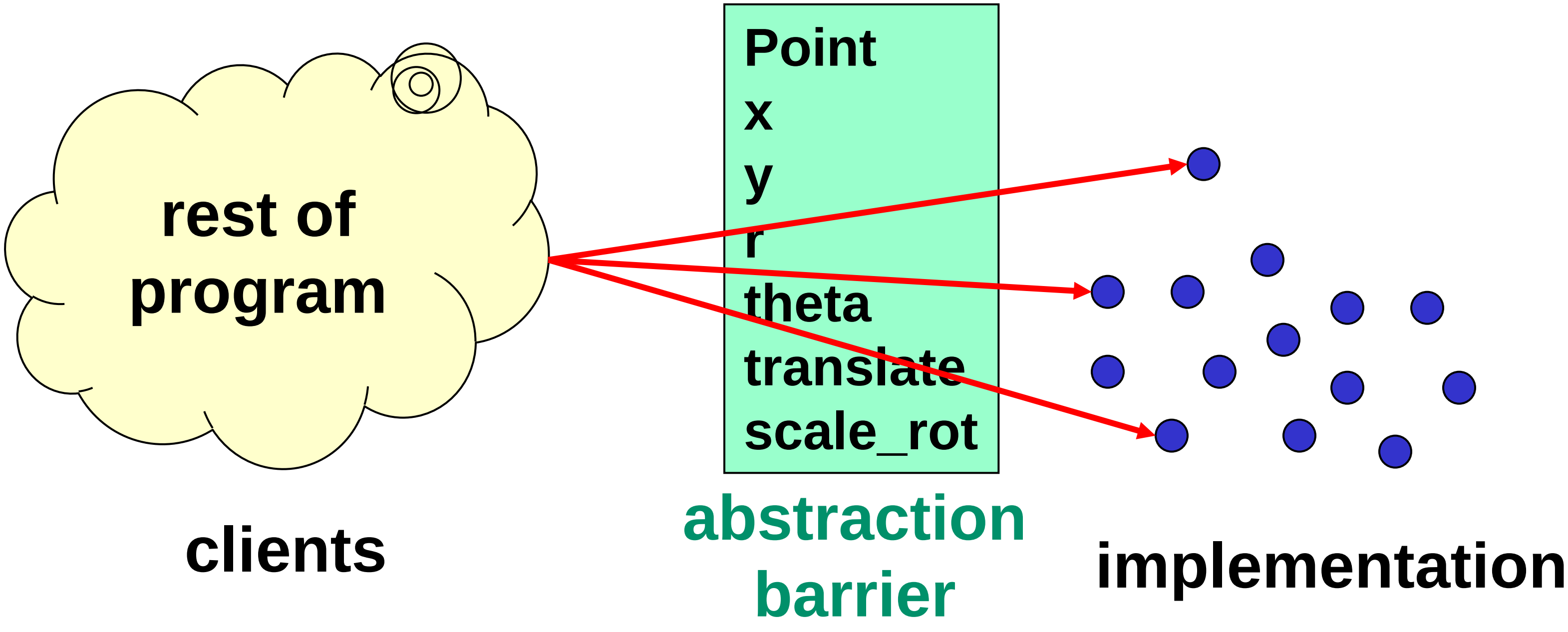
```
class Point {
  // A 2-d point exists somewhere in the plane, ...

  public float x();
  public float y();
  public float r();
  public float theta();

  // ... can be created, ...
  public Point();          // new point at (0,0)

  // ... can be moved, ...
  public void translate(float delta_x,
                        float delta_y);
  public void scaleAndRotate(float delta_r,
                             float delta_theta);
}
```

Abstract data type = objects + operations



The implementation is hidden

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

Reasoning about ADTs

- Specification: describes ADT only in terms of the abstraction
  - Never mentions the representation
- *Abstraction Function*: maps object → abstract value
  - What the data structure *means* as an abstract value
    - Ex: point in the plane represented by Point object
- *Representation Invariant*: maps object → boolean
  - True iff an object (the representation) is *well-formed*
  - Only well-formed representations (values) make sense as implementations of an abstract value

A data abstraction 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

Implementing an ADT

- To implement a data abstraction
  - Select the representation of instances, the “rep”
  - Implement operations in terms of that rep
  - In Java this is done in a class (something you’ve already done before many times)
- Choose a representation so that:
  - It is possible to implement the required operations
  - The most frequently used operations are efficient
    - But which will these be?
    - Abstraction allows the rep. to change later

Example: CharSet Abstraction
Finite, mutable set of characters

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

// effects: creates an 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: 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?

- 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 is executed – and after a **`CharSet`** is initialized
- Two ways to write it (in the **`CharSet`** class comments)

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

Or, more formally:

∀ indices i of elts . elts.elementAt(i) ≠ null

∀ indices i, j of elts .

  i ≠ j ⇒ ¬ 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);
}
```

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:
  - Here the client code can see the representation and (in this case) even manipulate it directly
  - We want only ADT operations to see/change the rep (otherwise we can't guarantee rep invariants maintained)
- Representation exposure is almost always EVIL
- If you do it, document why and how
  - And feel guilty about 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?
  // 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 (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: 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

Our real goal is to satisfy the specification of insert:

// modifies: this

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

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(elts) U {encrypt(‘a’)}

What if we used this abstraction function?

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

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

Placing the blame

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?

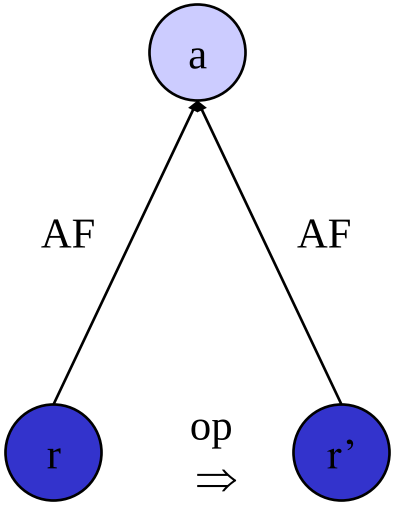
AF(this) = { c | encrypt(c) is contained in this.elts }

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

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

The abstraction function is a function

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”
  - “derived specification fields” more complex

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

- A printed representation is valuable for debugging

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