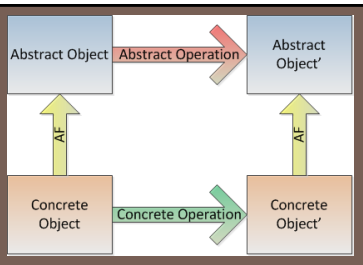


CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
ABSTRACT DATA TYPES

Autumn 2011

What is an ADT?

- Recall procedural abstraction
  - Abstracts from the details of procedures
  - A specification mechanism
  - Satisfying the specification with an implementation
- Data abstraction (Abstract Data Types or ADTs):
  - Abstracts from the details of data representation
  - A specification mechanism
    + a way of thinking about programs and designs
  - Satisfying the specification with an implementation

Why we need Abstract Data Types

- Organizing and manipulating data is pervasive
- Often crucial to start by designing data structures
- Potential problems with choosing a data structure
  - Decisions about data structures are made too early
  - Very hard to change key data structures later on

An ADT is a set of operations

- ADT abstracts away from a specific representation to focus on the semantic meaning of the data
- In what sense are the following two definitions different?

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

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

- Although the representations differ, the client should instead consider a `Point` as a set of operations to create and manipulate 2D points on the plane
- By restricting the client to *only* call operations to access data, the potential for modifying the representation (and supporting algorithms) remains

2D point

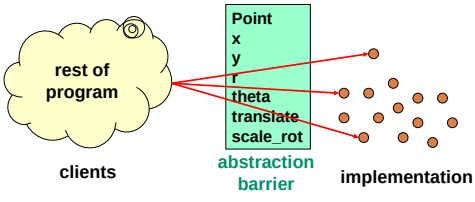

```
class Point {
  // A 2-d point exists somewhere in the plane, ...
  public float x();
  public float y();
  public float r();
  public float theta();

  // ... 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);
}
```


ADT = objects + operations

- The only operations on objects of the type are those provided by the abstraction
- The implementation is hidden



ADTs and specifications

- Specification: only in terms of the abstraction
  - Never mentions the representation
- *Abstraction function*: `Object` ⇒ `abstract value`
  - What the data structure means as an abstract value
    - Ex: where in the plane is that 2D point?
- *Representation invariant*: `Object` ⇒ `boolean`
  - Indicates whether the `Object` – the representation in the implementation – is well-formed
  - Only well-formed representations in the implementation can be mapped to abstract values

Implementing an ADT

- To implement a data abstraction
  - Select the representation of instances, the "*rep*"
  - Implement operations in terms of that rep
  - In Java, you do this in a class – in fact, you've done it many times before
- Choose a representation so that
  - It is possible to implement operations
  - The most frequently used operations are efficient

CharSet specification

Finite mutable set of Characters

```
// effects: creates an empty CharSet
public CharSet ( )

// modifies: this
// effects: this_post = this_pre ∪ {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

```
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 operation is applied – and after initialization
- Two ways of writing the `CharSet` rep invariant (as part of the comments for the `CharSet` class)
  - `// Rep invariant: elts has no nulls and no duplicates`
    `private List<Character> elts;`
  - `∀ 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

```
class CharSet {
  // Rep invariant: elts has no nulls and no duplicates

  private List<Character> elts = new ArrayList<Character>();
  public void insert(Character c)   {
     elts.add(c);
  }
  public void delete(Character c)   {
     elts.remove(c);
  }
  …
}
```

```
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";
```

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?
- Kind of, sort of, not really…

Representation exposure

- Consider the **client** code

```
CharSet s = new CharSet();
Character a = new Character('a');
s.insert(a);
s.getElts().add(a);
s.delete(a);
if (s.member(a)) …
```

- The client sees the representation and (in this case) can even manipulate it directly – makes it hard to maintain the rep invariant!
  - The client is no longer constrained to only manipulate the representation through the specification
- Representation exposure is external access to the rep; it is almost always evil (so if you do it, document why and how, and feel guilty about it!) – more on avoiding rep exposure next lecture

New implementation of `insert`

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

- This maintains the representation invariant for the class
- The rep invariant only considers structure – well-formedness – not meaning
- In this case, there is still an error – consider this client code

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

OOPS

Abstraction function to the rescue

17

- The abstraction function maps the representation to the abstract value it represents
- **`AF(CharSet this) =`**
  `{ c | c is contained in this.elts }`
  - Or the "set of Characters contained in **`this.elts`**"
- The abstraction function lets us reason about behavior from the client perspective
  - The AF is typically not executable
- Do we satisfy the specification of **`insert`**?

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


Helps identify problem

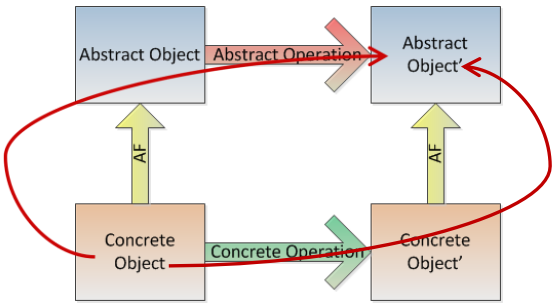
- Applying the abstraction function to the result of the call to **`insert`** yields **`AF(elts) ∪ {encrypt('a')}`**
  - So when member is checked, the implementation looks for 'a' rather than the encrypted value of 'a' – from the client's view, an inserted element is no longer found, even though it has not been deleted
- 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 }
```

Recap: the CharSet representation

19

- Ex: ['a','b','c'], ['b','a','c','d'], ['9','1','6']
- How do we know that these represent sets of characters to the client?
- How do we know that they don't represent hexadecimals numbers
  [$ABC_{16} = 2748_{10}$, $BACD_{16} = 47821_{10}$, $916_{16} = 2326_{10}$]?
- Or even unary numbers
  [ABC = 3, BACD = 4, 916 = 3]?
- **It is the AF and the specification that make this explicit**



- Creating the concrete object must establish the representation invariant
- Every concrete operation must maintain the rep invariant
- Creating the abstraction object must establish the abstraction function
- Every abstract operation must maintain the AF to provide consistent semantic meaning to the client
- If things are right, either red arrow above will give the same result

Next steps

21

- Assignment 1
  - Due tonight 11:59PM
- Assignment 2
  - out later today
  - due in two parts (M 11:59PM and F 11:59PM)
- Lectures
  - Abstract data types (M)
  - Modular design (W)