Data Abstraction

David Evans
www.cs.virginia.edu/cs205

Managing Complexity

- Modularity
  - Divided problem into procedures
  - Used specifications to separate what from how
- A big program can have thousands of procedures
  - How can we group them into modules?

Abstract Data Types

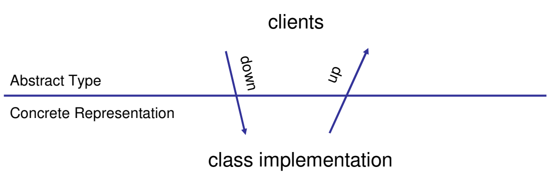
- Separate *what* you can do with data from *how* it is represented
- Client interacts with data through provided operations according to their specifications
- Implementation chooses how to represent data and implement its operations

Data Abstraction in Java

- A class defines a new data type
- Use **private** instance variables to hide the choice of representation
  - **private** declarations are only visible inside the class

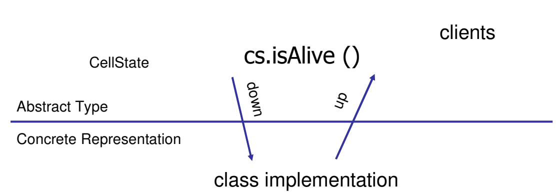
Up and Down

Clients manipulate an abstract data type by calling its operations (methods and constructors)



The representation of an abstract data type is visible only in the class implementation.

Cell State Representation




```
private boolean alive;
public boolean isAlive () { return alive; }
```

Advantages/Disadvantages

- More code to write and maintain
- Run-time overhead (time to call method)

+ Client doesn't need to know about representation
+ Suppose we want to add more states (e.g., question 2)

Set Example (ps2)

- Set abstract data type: represent a set of objects
- Operations:
  - Create an empty set
  - Mathematical set operations: add, contains, size, remove, union

Type Parameters

- We want to have sets of different types of objects
- How should we declare the Set methods?

```
public boolean add(?? elem)
public boolean contains(?? elem)
public ?? choose()
```

We don't want just one Set datatype.
We want different Sets for different element types.

Generic Datatype

```
public class Set<T> {
   ...
   public boolean add(T el)
   public T choose()
   public boolean contains(T el)
   ...
}
```

Note: Java did not support generic datatypes until version 1.5 (this is why the book doesn't use them)

Creating Specific Types

```
public class Set<T> {
   ...
   public boolean add(T el)
   public T choose()
   public boolean contains(T el)
   ...
}
```

Set<String> ⇒

```
public class Set<String> {
   ...
   public boolean add(String el)
   public String choose()
   public boolean contains(String el)
   ...
}
```


Abstract Data Type Specifications

- **Overview**: what the type represents
  - Mutability/Immutability
    - A Set is a *mutable*, unbounded set of objects of type T.
  - Abstract Notation
    - A *typical* Set is { x_1, …, x_n }.
- **Operations:** procedural specifications for each operation (public methods and constructors); use the abstract notation introduced in overview.

Set Specification

```
public class Set<T> {
  OVERVIEW: A Set is a mutable, unbounded set of objects of
      type T.  A typical Set is {x_1, ..., x_n }.

  public Set()
    EFFECTS: Initializes this to an empty set: { }.

  public boolean add(T el)
    MODIFIES: this
    EFFECTS: Adds el to the elements of this:
         this_post = this_pre U { el }
      Returns true iff el was not an element of this_pre.
```


contains

~~EFFECTS: Checks if *el* is an element of *this* and returns true if it is.~~

EFFECTS: Returns true iff *el* is an element of *this*.

EFFECTS: Returns $el \in this$.

EFFECTS: Returns el isIn this.

union

public void union(Set<T> t)

~~MODIFIES: this~~
~~EFFECTS: Adds the elements of t to this.~~

MODIFIES: this
EFFECTS: $this\_post = this\_pre \cup t$

Specifications should be **declarative** (what the outcome is), not **operational** (how it does it).

Implementing Abstract Data Types

Choosing a Representation

- Need a concrete data representation to store the state
  - Think about how it maps to abstract state
  - Think about how methods will be implemented
- A good representation choice should:
  - Enable straightforward implementations of as many methods as possible
  - Allow performance-critical methods to be implemented efficiently

Set Representation

- Option 1: private T [] rep;
  - Recall Java arrays are bounded
  - Easy to implement most methods, hard to implement insert
- Option 2: private Vector<T> rep;
  - Easy to implement all methods
  - Performance may be worse than for array

Implementing Set

```
public class Set<T> {
   // OVERVIEW: Sets are unbounded, mutable sets of elements of type T.
   //    A typical Set is {x1, ..., xn}

   // Representation:
   private Vector rep;
   public StringSet () {
      // EFFECTS: Initializes this to be empty: { }
      rep = new Vector ();
   }
   public void insert (String s) {
      // MODIFIES: this
      // EFFECTS: Adds s to the elements of this:
      //     this_post = this_pre U { s }
      rep.add (s);
   }
```

Could this implementation of insert be correct?

It depends…

```
public int size () {
      // EFFECTS: Returns the number of elements in this.
      Set<T> unique = new Set<T> ();
      for (T el : rep) {
         if (!unique.isIn (el)) {
            unique.add (current);
         }
      }
      return unique.rep.size ();
   }
```


Is it correct?

```
public int size () {
     // EFFECTS: Returns the number of
     //     elements in this.
     return rep.size ();
}
```

```
public void insert (String s) {
   if (!contains (s)) rep.add (s);
}
```


Reasoning About Data Abstractions

- How can we possibly implement data abstractions correctly if correctness of one method depends on how other methods are implemented?
- How can we possibly test a data abstraction implementation if there are complex interdependencies between methods?

What must we know to know if size is correct?

```
public int size () {
    // EFFECTS: Returns the number of
    //     elements in this.
    return rep.size ();
}
```

This implementation is correct only if we know the rep does not contain duplicates

To Reason about Operations

- We need to know:
- How the concrete rep maps to abstract values: **Abstraction Function**
- What values of the concrete rep are valid: **Representation Invariant**

Rep Invariant

- Predicate that all legitimate objects of the ADT must satisfy

  I: $C \to$ boolean
- Helps us reason about correctness of methods independently
  - Prove all objects satisfy the invariant before leaving the implementation code
  - Assume all objects passed in satisfy the invariant

Reasoning with Rep Invariants

REQUIRES: Rep Invariant is true for this (and any other reachable ADT objects)

EFFECTS: Rep Invariant is true for all new and modified ADT objects on exit.

Every public datatype operation implicitly includes these preconditions and postconditions.

Rep Invariant for Set

```
public class Set {
    // Representation:
    private Vector<T> rep;

  // RepInvariant (c) = c contains no duplicates

  or
  // RepInvariant (c) =
  //      forall i, j: rep[i].equals(rep[j])
  //          only when i == j
```


Implementing Insert?

```
public void insert (String s) {
      // MODIFIES: this
      // EFFECTS: Adds s to the elements of this:
      //     this_post = this_pre U { s }
      rep.add (s);
   }
```

Not a correct implementation: after it returns this might not satisfy the rep invariant!

Implementing Insert

```
public void insert (String s) {
      // MODIFIES: this
      // EFFECTS: Adds s to the elements of this:
      //     this_post = this_pre U { s }
      if (!contains (s)) { rep.add (s); }
   }
```

Possibly correct implementation: we need to know how to map rep to abstraction notation to know if this_post = this_pre U { s }

Abstraction Function

- The Abstraction Function maps a concrete state to an abstract state:

  $\mathcal{AF}$: $C \to \mathcal{A}$

  Function from concrete representation to the abstract notation introduced in overview specification.

What is the range of the Abstraction Function?

Range is concrete states for which rep invariant is true

Abstraction Function for Set

```
public class Set<T> {
   // OVERVIEW: Sets are unbounded,
   //    mutable sets of objects of type T.
   //    A typical Set is {x1, ..., xn}

   // Representation:
   private Vector<T> rep;
```

// $\mathcal{AF}$ (c) =
//   { $\mathcal{AF}_T$ (c.rep.elementAt(i))
                | 0 <= i < c.rep.size () }

Correctness of Insert

```
public void insert (String s) {
      // MODIFIES: this
      // EFFECTS: Adds s to the elements of this:
      //     this_post = this_pre U { s }
      if (!contains (s)) { rep.add (s); }
   }
```

Use abstraction function to show if add implements its specification, then
$\mathcal{AF}$(rep_post) = $\mathcal{AF}$(rep_pre) U {$\mathcal{AF}_{String}$(s)}

Reality Check

- Writing abstraction functions, rep invariants, testing code thoroughly, reasoning about correctness, etc. for a big program is a ridiculous amount of work!
- Does anyone really do this?
  - Yes (and a lot more), but usually only when its really important to get things right:
- Cost per line of code:
  - Small, unimportant projects: $1-5/line
  - WindowsNT: about $100/line
  - FAA's Automation System (1982-1994): $900/line

Charge

- PS3: out today, due next Monday
  - Reason about data types using abstraction functions and rep invariants
  - Implement the DirectedGraph abstract data type you used in PS2
- Wednesday: Quiz 2
  - Mostly on data abstraction
  - Chapter 5 and lectures
  - Maybe a question or two on testing