Abstract Data Types II

Spring 2013
A Word on PSet2

This problem set is longer & more difficult than previous ones.

Get started early.

Get help from TAs & LAs during office hours & LA hours.

Get started early!
Today’s Topics

Contextual Equivalence
Java Interfaces
More on theory of abstract data types
Abstraction Functions
Abstract Data Types: The Main Point

Now we can create new types that behave (almost) like built-in types, but with powerful domain-specific operations.

Necessary to design our types with safety, power, and coherence in mind

- Careful design will prevent many kinds of bugs
- Prevent clients from inadvertently misusing the type
Classifying Operations on Types

Four major kinds of operations.

For an abstract type $T$ and other types $t$

- Constructor: $t^* \rightarrow T$
- Producers: $T^+, t^* \rightarrow T$
- Mutators: $T^+, t^* \rightarrow \text{void}$
- Observers: $T^+, t^* \rightarrow t$

Useful classification, but not perfect

- In particular, some operations may be mutator+observer or mutator+producer
- In general, mutator is a stronger term than others
- Operation that mutates the object is mutator, regardless of what it returns
Preserving Invariants

A good ADT must preserve its own invariants, and prevent clients from breaking them.

Invariant – property of a program or object that is always true

- Immutability is one we’ve seen before
- Other invariants may express relationships between instance variables etc.

Preserving invariants makes reasoning about the use of the ADT easier

- Only need to reason about ADT itself, not its use (since use cannot violate invariant)
- E.g. String immutability means we don’t worry about disallowed modifications to Strings in our programs
Structural Induction

How to establish invariants?

- Must ensure the invariant is true when the object is constructed
- All other operations on the object must preserve invariant
- But also want to avoid representation exposure

Structural Induction

- If an invariant of an abstract data type is
  1. Established by its constructors
  2. Preserved by producers, mutators, and observers, and
  3. No representation exposure occurs

Then the invariant is true for all instances of the abstract data type.
Testing & Rep Independence

Contextual equivalence

- Two classes are contextually equivalent, if, given the same arbitrary sequence of method calls, we get the same results.

In other words, can test if multiple implementations actually representation independent

- Just have (many) test cases that compare outputs for arbitrary sequences of operations.
- Also compare initial states by testing output of constructors.
- Definitely not a formal way to verify representation independence (or correctness!)
- But useful way to think when considering whether your implementation is representation independent.
Interfaces

Codify the contract for an abstract data type & allow a program to be written using the ADT

Interfaces in Java
- A reference type that contains only constants, method signatures and nested types
- No method bodies, no signatures of constructors
- Cannot be instantiated

Classes implement the interface
- Ex: ArrayList implements List
- Implementing means the class provides method bodies for each declared method in the interface
- Not part of the class hierarchy!
Interfaces

Java’s compile-time static type checking ensures classes that declare they implement an interface actually do:

- In other words, if class says it is implementing the ADT contract, compiler checks to make sure.
- But many serious errors can be missed by type checker.
- Only checking the class implements the required methods with the required signatures.

Any class can implement multiple interfaces:

- Replacement for multiple inheritance.
- An object can be referred to through a reference of the type of any of its interfaces.
Why Interfaces?

Documentation for humans & compiler

- Easier to read than the implementation, & contains only essential information about ADT operations

Flexibility for performance tradeoffs

- Different implementations of the ADT may have different performance characteristics
- But correctness must be true for any implementation

Flexibility in providing invariants

- Optional methods allow some implementations to elide certain methods (e.g. immutable List implementations)
- Optional method is not a language construct though
- Methods with intentionally loose specifications (e.g. by not restricting ordering) give implementation freedom
Why Interfaces?

**Multiple views of a class**
- Any class can implement multiple interfaces
- Single object may be special case of multiple ADTs
Example Interface: List
Representation Values & Abstract Values

**Representation values**
- Values of the actual entities used to implement the ADT
- Usually more than one object
- To simplify, we will think about it as just a mathematical value

**Abstract values**
- Values that the ADT is designed to support
- Do not actually exist but are how we want clients to view the elements of the ADT
Example: Sets of characters

Suppose we want to design an ADT for a set of characters

- Choose a representation as a string
- Disallow duplicate characters
- E.g. the set \{a,c,e\} is represented as “ace” or “aec” or “cae” or…

Representation values

- Strings

Abstract values

- Sets of characters
Abstraction Function

In the graph

- Every abstract value is mapped to, since we need to manipulate/create all possible abstract values
- Some abstract values are mapped to from more than one representation value
- Not all representation values are mapped

Abstraction function $AF: R \rightarrow A$

- Maps representation values to the abstract values they represent
- Not necessarily one-to-one, but is an onto function
- For all values in $A$, there is a representation in $R$ such that $AF$ maps that representation to the value in $A$
Representation Invariant

**In the graph**
- Every abstract value is mapped to, since we need to manipulate/create all possible abstract values
- Some abstract values are mapped to from more than one representation value
- Not all representation values are mapped

**Representation invariant RI: R \rightarrow boolean**
- For a representation value \( r \), \( RI(r) \) is true iff \( r \) is mapped by \( AF \)
- Representation invariant tells us whether a given representation value corresponds to an abstract value
- \( RI \) forms a set: the subset of the representation values on which \( AF \) is defined
Abstraction Functions & Rep

Invariants

Important: abstract value space & choice of representation space do not automatically dictate RI or AF!

Representation values determined by type(s) chosen for representation

- Does not necessarily mean all values of representation are legal
- E.g. Our rep for sets of chars could allow repetition as long as characters are sorted--- same rep value space, different RI
- E.g Our rep could be that a string “aceg” means the set {a,b,c,e,f,g} so different AF
Abstraction Functions & Rep Invariants

Designing an abstract type means

- Choosing abstract value space for the specification
- Choosing the representation value space for the implementation
- Deciding what representation values to use & how to interpret them
  - i.e. defining RI and AF
Example: Rationals
**Example: RatNum**

RatNum

We could have the same ADT with a looser Representation Invariant

- E.g. remove the requirement that numer/denom needs to be in reduced form
Checking Representation Invariants

/**
 * Check that the rep invariant is true.
 *
 * Assertion violations will be checked at runtime
 * if the JVM is run with -enableassertions
 */

private void checkRep() {
    assert denom > 0;
    // assume we have a gcd function
    assert gcd(numer, denom) == 1;
}

Checking RI can eliminate bugs in implementation

- Good practice to include checks that run during development
- Can include checks in production version for extra safety
Example: LinkedList

```
<table>
<thead>
<tr>
<th>List</th>
<th>Entry</th>
<th>Entry</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>dat</td>
<td>nxt</td>
<td>dat</td>
<td>nxt</td>
</tr>
<tr>
<td>null</td>
<td></td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>
```

Sentinel Object
Example: LinkedList

What’s the Representation Invariant & Abstraction Function if we implement the List interface using a linked list?

```java
public class LinkedList<E> implements List<E> {

    private Node first, last;

    private class Node {
        E elt;
        Node next;
    }

    ...

    Representation Invariant:
    ▶ Last is a sentinel object with elt null and next null, reachable from first by following next references, or is the same object referenced by first
```
Example: LinkedList

What’s the Representation Invariant & Abstraction Function if we implement the List interface using a linked list?

```java
public class LinkedList<E> implements List<E> {

    private Node first, last;

    private class Node {
        E elt;
        Node next;
    }

    // ...

    Abstraction Function:
    ➢ Represents the sequence first.elt, first.next.elt, ..., up to but not including the element last, unless first=last=sentinel, in which case it represents the empty list
```
Proving Correctness of ADT Implementations

Combination of Abstraction Function & Representation

Invariant define an invariant for the class

➢ Use Structural Induction!

Additional correctness constraint: representation-independent specification of each method

➢ Interpret the spec using the abstraction function & verify the method follows the specification
Example: LinkedList.set()

/**
 * Replace an element at an index.
 * @param i index into the list, starting from 0
 * @param v new value for that list position
 * @throws IndexOutOfBoundsException if i is not in [0,size)
 */

public void set(int i, E e) throws IndexOutOfBoundsException;

© 6.005 Course Staff 07-13
Example: LinkedList.set()
Example: LinkedList.set()

/** Replace an element at an index.
* @param i index into the list, starting from 0
* @param v new value for that list position
* @throws IndexOutOfBoundsException if i is not in [0,size)
*/

public void set(int i, E e) throws IndexOutOfBoundsException {
    Node n;

    // follow links from the beginning until we get to the right
    // element
    for (n=first; n!=last && i>0; n=n.next)
        --i;

    // if the element is found
    if (i==0 && n!=last) {
        n.elt = e;
        return;
    }

    throw new IndexOutOfBoundsException();
}
Example: LinkedList.set()

/** Replace an element at an index.  
 * @param i index into the list, starting from 0  
 * @param v new value for that list position  
 * @throws IndexOutOfBoundsException if i is not in [0,size)  
 */
    
    public void set(int i, E e) throws IndexOutOfBoundsException {
        Node n;

        // follow links from the beginning until we get to the right  
        // element
        for (n=first; n!=last && i>0; n=n.next)  
            --i;

        // if the element is found
        if (i==0 && n!=last) {
            n.elt = e;
            return;
        }

        throw new IndexOutOfBoundsException();
    }

    Is the RI preserved?
    ➢ Assume RI is true before the method is called.
    ➢ Since no next links are altered, the RI is trivially true after the method is called
Example: LinkedList.set()

```java
/** Replace an element at an index.
 * @param i index into the list, starting from 0
 * @param v new value for that list position
 * @throws IndexOutOfBoundsException if i is not in [0, size)
 */
public void set(int i, E e) throws IndexOutOfBoundsException {
    Node n;

    // follow links from the beginning until we get to the right
    // element
    for (n=first; n!=last && i>0; n=n.next)
        --i;

    // if the element is found
    if (i==0 && n!=last) {
        n.elt = e;
        return;
    }

    throw new IndexOutOfBoundsException();
}
```

Is the specification implemented correctly?

- First loop reduces the problem in each iteration, until replace 1st element of sublist
Reasoning About ADTs from Clients

A correct implementation of the ADT will be contextually equivalent to any other correct implementation.

Using AF/RI, possible to reason about ADT implementations from client code

- Example, starting with an empty list
  1. add(1)
  2. add(2)
  3. remove(1)
  4. int n = l.get(0)
Reasoning About ADTs from Clients

A correct implementation of the ADT will be contextually equivalent to any other correct implementation.

Using AF/RI, possible to reason about ADT implementations from client code.

- Example, starting with an empty list:

  ```
  l.add(1)
  l represents the list (1)
  l.add(2)
  l represents the list (1,2)
  l.remove(1)
  l represents the list (2)
  int n = l.get(0)
  n is 2
  ```
Reminder: Main Ideas

Abstraction Function & Representation Invariant relate ADT and its representation

You will be asked to write AF & RI for ADTs
Summary

Interfaces allow formalization of ADTs by specifying the set of operations needed

Methods of ADTs must have representation-independent specifications

Abstraction Function: maps representation values to abstract values

Representation Invariant: defines the subset of values from the representation value space that correspond to values in the abstract value space