Reminders

• HW3 is due tonight

• HW4 will be posted shortly
  – section tomorrow will be help on HW4
  – quite a bit more coding than HW3

• Quiz 2 is due on Friday
Outline

This lecture:
1. What is an Abstract Data Type (ADT)?
2. How to specify an ADT?
3. Design methodology for ADTs

Next lecture:
• Documenting an implementation of an ADT
  – representation invariants
  – abstraction functions
Procedural and data abstractions

Procedural abstraction:
- abstract from implementation details of procedures (methods)
- specification is the abstraction
- satisfy the specification with an implementation

Data abstraction:
- abstract from details of data representation
- also a specification mechanism
- way of thinking about programs and design
- standard terminology: Abstract Data Type, or ADT
  - invented by Barbara Liskov in the 70s
  - one of the fundamental ideas of computer science
Why we need Data Abstractions (ADTs)

Organizing and manipulating data is pervasive
  – inventing and describing algorithms is less common

Often best to start your design by designing data
  – what operations will be permitted on the data by clients
  – how will relevant data be organized (data structures)
  – see CSE 332 & CSE 344
Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

-- Linus Torvalds

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won’t usually need your flowcharts; they’ll be obvious.

-- Fred Brooks
Why we need Data Abstractions (ADTs)

Organizing and manipulating data is pervasive
– inventing and describing algorithms is less common

Often best to start your design by designing data structures
– how will relevant data be organized
– what operations will be permitted on the data by clients
– see CSE 332 & CSE 344

Potential problems with choosing a data abstraction:
– hard to know ahead of time what to optimize
  • programmers are “notoriously” bad at this (Liskov)
– if not done properly, hard to change key data structures
An ADT is a set of operations

- ADT abstracts from the organization to meaning of data
- ADT abstracts from data structures to use
- Representation should not matter to the client
  - so hide it from the client

Alternative representations of a right triangle:

```java
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 to use operations to access data
Are these classes the same?

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

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

*Different Details*: cannot replace one with the other in a program

*Same Concept*: both classes implement the concept “2D point”

Goal of Point ADT is to express the sameness:
- clients should depend only on the concept “2D point”
- achieve this by specifying operations not the representation
- write clients that can work with either representation
Benefits of ADTs

If clients “respect” or “are forced to respect” data abstractions…
- For example, “it’s a 2D point with these operations…”

- Can fix bugs by changing how ADT is implemented
- Can change algorithms
  - For performance
  - In general or in specialized situations
- Can delay decisions on how ADT is implemented
- ...

We talk about an “abstraction barrier”
- a good thing to have and not cross (a.k.a. violate)
Concept of 2D point, as an ADT

class Point {
    // A 2D point exists 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 / Getters

Creators/Producers

Mutators
Abstract data type = objects + operations

• Implementation is hidden

• The only operations on objects are those provided by the abstraction
Specifying a data abstraction

- A collection of procedural abstractions
  - not a collection of procedures!

- Need a way write specifications for these procedures
  - need a vocabulary for talking about what the operations do
  - need to avoid referencing the actual implementation

- Use “math” to specify these procedures
  - mathematical description of a state is called an abstract state
  - describes what the state “means” not the implementation
  - each operation described in terms of “creating”, “observing”, “producing”, or “mutating” the abstract state
Specifying an ADT

Immutable

1. overview
2. abstract state
3. creators
4. observers
5. producers
6. mutators

Mutable

1. overview
2. abstract state
3. creators
4. observers
5. producers (rare)
6. mutators

- Creators: return new ADT values (e.g., Java constructors)
- Observers / Getters: Return information about an ADT
- Producers: ADT operations that return new values
- Mutators: Modify a value of an ADT
Implementing an ADT

Next lecture will be about implementations of ADTs

This lecture is about the ADTs themselves
  – these are specifications
  – should have *no information* about the implementation
    • (latter called the "concrete representation")
Poly, an immutable datatype: overview

/**
 * A Poly is an immutable polynomial with integer coefficients. A typical Poly is
 * \[ c_0 + c_1 x + c_2 x^2 + \ldots \]
 **/

class Poly {

Overview:
- state if immutable (default not)
- define abstract states for use in operation specifications
  - difficult and vital!
  - appeal to math if appropriate
  - give an example (reuse it in operation definitions)
  - make no reference to concrete representation
Poly: creators

// effects: makes a new Poly = 0
public Poly()

// effects: makes a new Poly = cx^n
// throws: NegExponent if n < 0
public Poly(int c, int n)

Creators
- new object, so no pre-state: only effects, no modifies
- overloading: distinguish procedures of same name by parameters
  - use with care (see Effective Java)
  - will see alternative design patterns later on

(Note: Javadoc above omits many details.)
Poly: observers

// returns: the degree of this,
// i.e., the largest exponent with a
// non-zero coefficient.
// Returns 0 if this = 0.
public int degree()

// returns: the coefficient of the term
// of this whose exponent is d
// throws: NegExponent if d < 0
public int coeff(int d)

(Note: Javadoc above omits many details.)
Notes on observers

Observers
– used to obtain information about objects of that type
– return values of other types
– never modify the abstract state
– specification uses the abstraction from the overview

this
– abstract value of particular Poly object being accessed
  • target of the method call (object on which the call was made)

Poly x = new Poly(4, 3);
int c = x.coeff(3);
System.out.println(c); // prints 4
Poly: producers

// returns: this + q (as a Poly)
public Poly add(Poly q)

// returns: the Poly equal to this * q
public Poly mul(Poly q)

// returns: -this
public Poly negate()
Notes on producers

• Operations on a type that create other objects of the same type

• Common in immutable types like `java.lang.String`
  – `String substring(int offset, int len)`

• No side effects
  – *never* modify the abstract value of existing objects
IntSet, a mutable datatype: overview and creator

// Overview: An IntSet is a mutable, unbounded set of integers. A typical IntSet is { x1, ..., xn }.

class IntSet {

    // effects: makes a new IntSet = {}
    public IntSet()
IntSet: observers

// returns: true if and only if x in this
public boolean contains(int x)

// returns: the cardinality of this
public int size()

// returns: some element of this
// throws: EmptyException when size()==0
public int choose()

(Note: Javadoc above omits many details.)
IntSet: mutators

// modifies: this
// effects: this_{post} = this_{pre} + \{x\}
public void add(int x)

// modifies: this
// effects: this_{post} = this_{pre} - \{x\}
public void remove(int x)

(Note: Javadoc above omits many details.)
Notes on mutators

• Operations that modify an element of the type

• Rarely modify anything (available to clients) other than this
  – list this in modifies clause

• Typically have no return value
  – “do one thing and do it well”
  – (sometimes return “old” value that was replaced)

• Mutable ADTs may have producers too, but that is less common