CSE 331
Software Design & Implementation

Kevin Zatloukal

Spring 2020

Abstract Data Types (ADTs)

Outline

Previously looked at writing specificationss for methods.
The situation gets more complex with object-oriented code...

This lecture:

1. What is an Abstract Data Type (ADT)?
2. How to write a specification for an ADT
3. Design methodology for ADTs

Next lecture:

- Documenting an *implementation* of an ADT

Why we need Data Abstractions (ADTs)

Manipulating and presenting data is pervasive

- choosing how to organize that data is key design problem
- inventing and describing algorithms is less common

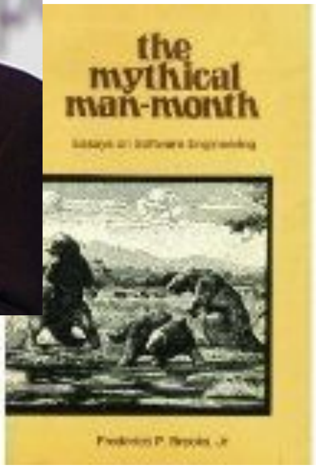
Often best to start your design by designing data...

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



Designing Around Data

Brooks says it is enough to decide what your data looks like

- (don’t even need to say how it is organized)
- can figure out the data structures & code from that

In fact, even that is possibly too detailed...

- all we really need to know is what operations we need to perform with the data

An *abstract data type* defines a class of abstract objects which is completely characterized by the operations available on those objects …

When a programmer makes use of an abstract data object, he [sic] is concerned only with the behavior which that object exhibits but not with any details of how that behavior is achieved by means of an implementation…

Programming with Abstract Data Types
by Barbara Liskov and Stephen Zilles

Why we need Data Abstractions (ADTs)

Manipulating and presenting data is pervasive

- choosing how to organize that data is key design problem
- inventing and describing algorithms is less common

Often best to start your design by designing data

- first, what **operations** will be permitted on the data (for clients)
- next, decide how data be organized (data structures)
  - see CSE 332 & CSE 344
- lastly, write the code

Why we need Data Abstractions (ADTs)

Manipulating and presenting data is pervasive

- choosing how to organize that data is key design problem

Hard to always choose the right data structures ahead of time:

- hard to know ahead of time what will be too slow
- programmers are “notoriously” bad at this (Liskov)

ADTs give us the freedom to **change** data structures later on

- data structure details are hidden from the clients

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

Abstract Data Type (ADT)

- invented by Barbara Liskov in the 1970s
- one of the fundamental ideas of computer science

ADTs in Java

An ADT is a set of **operations**

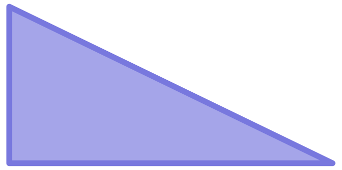
ADT abstracts from the *organization* to *meaning* of data

- details of data structures are hidden from the client
- client see only the operations that provided

An ADT is a set of **operations**

ADT abstracts from the *organization* to *meaning* of data

- details of data structures are hidden from the client
- client see only the operations that provided



```
class RightTriangle {
  float base, altitude;
}
```

```
class RightTriangle {
  float base, hypot, angle;
}
```

Instead, think of a type as a set of operations

`create, getBase, getArea, …`

Force clients 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;
}                      }
```

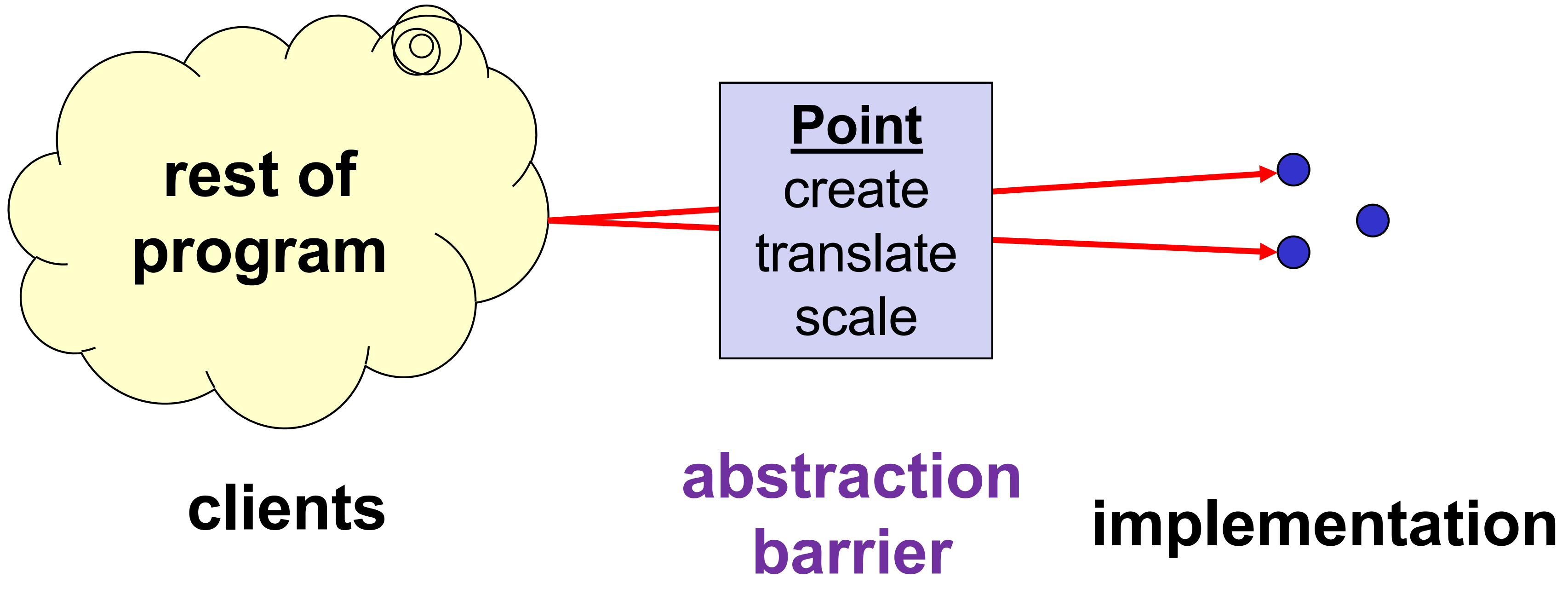
Different Details: cannot replace one with the other in a program

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

Abstract data type = objects + operations



We call this an “abstraction barrier”

- a good thing to have and not *cross* (a.k.a. *violate*)
- prevents clients from depending on implementation details

Benefits of ADTs

If clients are forced to respect data abstractions, ...

- Can change how data is stored (and data structures)
  - fix bugs
  - improve performance

- Can also change algorithms

- Can delay decisions on how ADT is implemented

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();                       Observers / Getters
  public float theta();

  // ... can be created, ...
  public Point(); // new point at (0,0)          Creators /
  public Point centroid(Set<Point> points);      Producers

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

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

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`

- Should have no information about the implementation
  - (latter called the “concrete representation”)
  - leave ourselves free to change it later
- A collection of procedural abstractions — not procedures

Specifying an ADT

- 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” (when possible) not actual fields to describe the state
  - abstract description of a state is called an **abstract state**
  - describes what the state “means” not the implementation
    - give clients an abstract way to think about the state
  - each operation described in terms of “creating”, “observing”, “producing”, or “mutating” the abstract state

Poly, an immutable datatype: overview

```
/**
 * A Poly is an immutable polynomial with
 * integer coefficients.  A typical Poly is
 *           $c_0 + c_1x + c_2x^2$ + ...
 */
class Poly {
```

Abstract state

Overview

- defines the abstract states for use in operation specifications

Notes on overview

Overview

- state if immutable (default not)
- define abstract states for use in operation specifications
  - difficult and vital!
  - appeal to math if appropriate
  - **never** make reference to concrete representation
- give an example (reuse it in operation definitions)

Poly: creators

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

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

Creators

- creates a new object

Note: Javadoc above omits many details...

- should be `/** ... */` not `// ...`
- should be `@spec.effects` not `effects`

Poly: observers

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

← "this" means the abstract state

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

Observers

- obtains information about objects of that type

Notes on observers

Observers

- **Never** modify the abstract state
- Specification uses the abstraction from the overview

Poly: producers

```
// returns: this + q
public Poly add(Poly q)

// returns: this * q
public Poly mul(Poly q)

// returns: -this
public Poly negate()
```

Producers

- creates other objects of the same type

Notes on producers

Producers

- 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

Poly, example

```
Poly x = new Poly(4, 3);
Poly y = new Poly(5, 3);
Poly z = x.add(y);

System.out.println(z.coeff(3));   // prints 9
```

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()
```

(Note: Javadoc is highly simplified...)

IntSet: observers

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

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

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

IntSet: mutators

```
// modifies: this
// effects:  change this to this + {x}
public void add(int x)

// modifies: this
// effects:  change this to this - {x}
public void remove(int x)
```

Mutators

- modify the abstract state of the object

Notes on mutators

Mutators

- 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