CSE 331
Software Design & Implementation

James Wilcox & Kevin Zatloukal

Fall 2022

ADT Implementation: Abstraction Functions

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

- Need a way write specifications for these procedures
  - need a vocabulary for talking about what the operations do (other than 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

- For familiar ideas from math (point, triangle, number, set, etc.), we can use those concepts as our abstract state
  - otherwise, we need to invent a concept for them

Specifying an ADT

Immutable	Mutable
`1. overview`	**`1. overview`**
`2. abstract state`	**`2. abstract state`**
`3. creators`	**`3. creators`**
`4. observers`	**`4. observers`**
`5. producers`	**`5. producers`** **`(rare)`**
~~**`6. mutators`**~~	**`6. mutators`**

Described in terms of how they change the **abstract state**

- abstract description of what the object means
  - difficult (unless concept is already familiar) but vital
- specs have no information about concrete representation
  - leaves us free to change those in the future

Poly, an immutable data type: 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: provide high level information about the type

- state if immutable (default not)
- define abstract states for use in operation specifications
  - easy here, but sometimes difficult — always vital!
- 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

- obtains information about objects of that type

- Specification uses the abstract state from the overview

- **Never** modifies the abstract state

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

- creates 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 state 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()
```

(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

  - modify the abstract state of the object

- 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

Is everything an ADT?

- Purpose of an ADT is to hide the representation details

- Some classes are not trying to hide their representation
  - Example: `Pair` with fields `first` and `second`
  - representation is very unlikely to change
  - reasonable to expose every field via a method

- Some classes do not have a representation
  - they are more "processes" than data
  - Example: `PrinterController` with various `print` methods
  - it may store data, but client does not need to think about it

Implementing a Data Abstraction (ADT)

To implement an ADT:

- select the representation of instances
- implement operations using the chosen representation

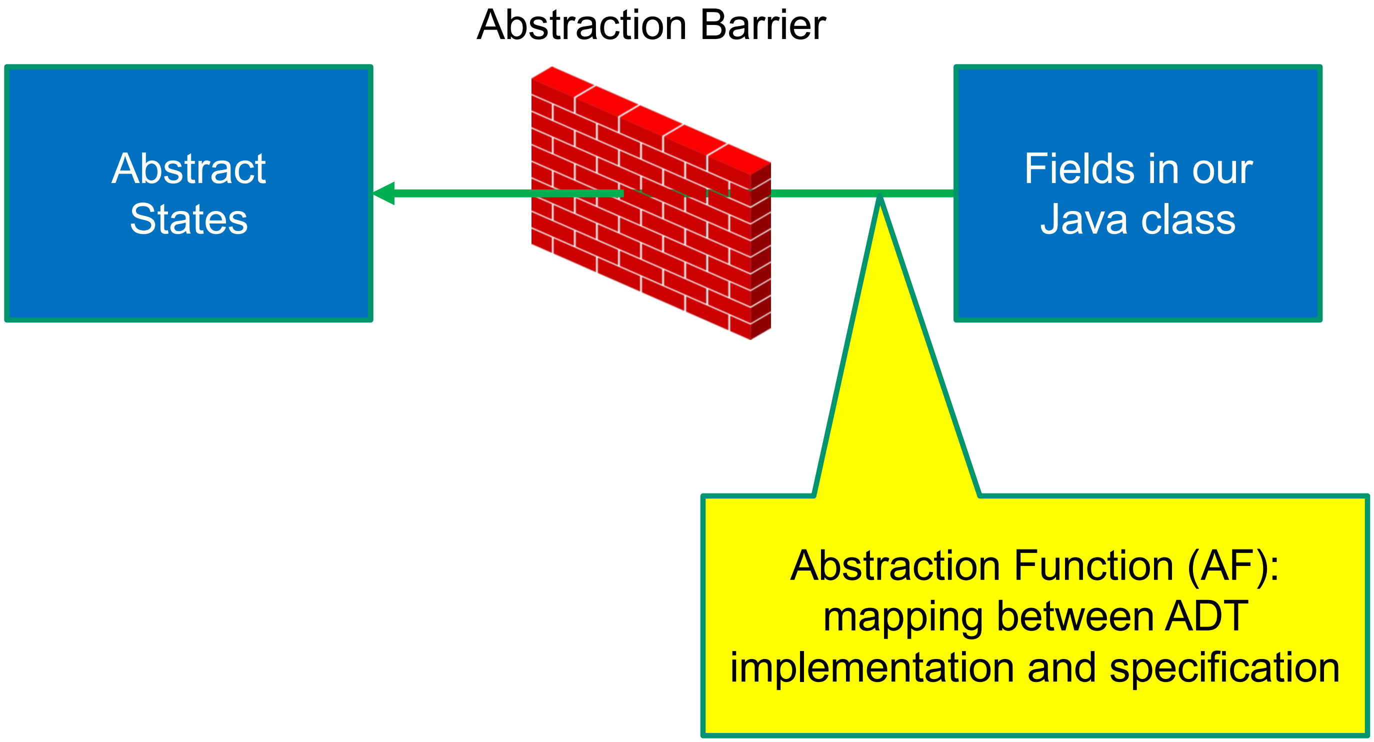
Choose a representation so that:

- it is possible to implement required operations
- the most frequently used operations are efficient / simple / …
    - abstraction allows the rep to change later
    - almost always better to start simple

Use **reasoning** to verify the operations are correct

- specs are written in terms of *abstract states* not *actual fields*
- need a new tool for this...

Data abstraction outline



ADT specification

ADT implementation

Abstraction Barrier

Abstract States

Fields in our Java class

Abstraction Function (AF): mapping between ADT implementation and specification

Connecting implementations to specs

For implementers / debuggers / maintainers of the implementation:

Abstraction Function: maps Object → abstract state

- says what the data structure *means* in vocabulary of the ADT
- maps the fields to the abstract state they represent
  - can check that the abstract value after each method meets the postcondition described in the specification

Representation Invariant: (next lecture)

Example: Circle

```
/** Represents a mutable circle in the plane. For example,
  * it can be a circle with center (0,0) and radius 1. */
public class Circle {

  // Abstraction function:
  // AF(this) = a circle with center at this.center
  //   and radius this.rad
  private Point center;
  private double rad;

  //  ...

}
```

Example: Circle 2

```
/** Represents a mutable circle in the plane. For example,
  * it can be a circle with center (0,0) and radius 1. */
public class Circle {

  // Abstraction function:
  // AF(this) = a circle with center at this.center
  //   and radius this.center.distanceTo(this.edge)
  private Point center, edge;

  //  ...

}
```

Example: Polynomial

```
/** An immutable polynomial with integer coefficients.
  * Examples include 0, 2x, and 3x^2 + 5x + 6. */
public class IntPoly {

  // Abstraction function:
  // AF(this) = sum of coeffs[i] * x^i
  //                for i = 0 .. coeffs.length-1
  private final int[] coeffs;

  //  ...

}
```

Example: Polynomial 2

```
/** An immutable polynomial with integer coefficients.
  * Examples include 0, 2x, and 3x^2 + 5x + 6. */
public class IntPoly {

  // Abstraction function:
  // AF(this) = sum of monomials in this.terms
  private final LinkedList<IntTerm> terms;

  //  ...

}
```

Example: Stack

```
/** List that only allows insert/remove at right end. */
public class IntStack {

  // AF(this) = vals[0..len-1]
  private int[] vals;
  private int len;

  // ...

}
```

Example: Stack

```
// AF(this) = vals[0..len-1]
private int[] vals;
private int len;

// Creates an empty stack.
public IntStack() {
  vals = new int[3];
  start = len = 0;
}
```

AF(this) = vals[0..-1] = []

Example: Stack

```
// AF(this) = vals[0..len-1]
private int[] vals;
private int len;

// @return number of elements in the collection
public length() {
  return len;
}
```

length of this = length of vals[0..len-1] = len

Example: Stack

```
// AF(this) = vals[0..len-1]
private int[] vals;
private int len;

// @modifies this
// @effects this = this + [value]
public push(int value) {
  ensureEnoughSpace(len+1);  // make sure vals[len] exists
  vals[len] = value
  len = len + 1;
}
```

AF(this) = vals[0 .. len -1]
  = vals_0 [0 .. len - 2] + [value]
  = vals_0 [0 .. len_0 - 1] + [value]
  = AF(this_0) + [value]

Example: Stack

```
// AF(this) = vals[0..len-1]
private int[] vals;
private int len;

// @requires length > 0
// @modifies this
// @effects this = this[0..length-2]
public pop() {
  ...
}
```

talks about “this” **not vals** and
“length” **not len**

Example: Stack

```
// AF(this) = vals[0..len-1]
private int[] vals;
private int len;

// @requires length > 0
// @modifies this
// @effects this = this[0..length-2]
public pop() {
  len = len - 1;
}
```

Example: Stack

```
// AF(this) = vals[0..len-1]
private int[] vals;
private int len;

// @requires length > 0
// @modifies this
// @effects this = this[0..length-2]
public pop() {
```

  $\{\{\ length > 0\ \}\}$

```
  len = len - 1;
```

  $\{\{\ this = this_0\,[0\ ..\ len_0 - 2]\ \}\}$

```
}
```

$\{\{\ len > 0\ \}\}$

$\{\{\ len_0 > 0 \text{ and } len = len_0 - 1\ \}\}$

$\Rightarrow \{\{\ AF(this) = vals[0\ ..\ len - 1]$

$= vals[0\ ..\ len_0 - 2]\ \}\}$

Summary: the abstraction function

- Purely conceptual (not a Java function)
- Allows us to check correctness
  - use reasoning to show that the method leaves the abstract state such that it satisfies the postcondition