Abstract Data Types (ADTs)

Thanks to Michael Ernst,
University of Washington
So Far

• Specifications
  • Benefits of specifications
  • Specification conventions
    • Javadocs, PoS specifications, JML/Dafny

• Specification style
• Specification strength
• Comparing specifications
• Comparing specifications with logical formulas
Outline

• What is an abstract data type (ADT)?
• Specifying an ADT
  • Immutable
  • Mutable
• The ADT design methodology
• Next: reasoning about ADT implementations
Abstraction

• Abstraction: hiding unnecessary low-level details
  • E.g. Information hiding

• Control abstraction (procedural abstraction)
  • A procedure (method) implements the details of an algorithm
  • One part of abstraction: signature, provides name, parameter types, return type. Not enough!
    • E.g., int binarySearch(int[] a, int key)
  • Another part: specification, provides detail about behavior and effects
  • Reasoning about code connects implementation to specification
Abstraction

• Data abstraction
  • Types: abstract away from the details of data representation
  • E.g., type String is an abstraction
  • E.g., C type struct Person is an abstraction

• Abstract Data Type (ADT): higher-level data abstraction
  • The ADT is operations + object
  • A specification mechanism
  • A way of thinking about programs and design
Abstract Data Types are Important

• ADTs are about organizing and manipulating data
• Organizing and manipulating data is pervasive. Inventing and describing algorithms is not
• Start your design by designing data structures. Write code to access and manipulate data
• Chose data structures carefully!
ADT is a way of thinking about programs and design

• From domain concept
  • E.g., the math concept of the polynomial, the integer set, the concept of a library item, etc.

• through ADT
  • Describes domain concept in terms of specification fields and abstract operations

• to implementation
  • Implements ADT with representation fields and concrete operations
Example: Polynomial with Int Coefficients, Domain Concept & ADT

ADT:
Overview description:
A Poly is an **immutable** polynomial with int coefficients. A Poly is:
• $c_0 + c_1x + c_2x^2 + ...$
  • Specification fields $c_0$, $c_1$ etc.

Set of abstract operations:
add, mul, eval, etc. with PoS style specs referencing abstract specification fields
Example: Polynomial with Int Coefficients, Implementation

class Poly {
    // rep. invariant: d = coeffs.length - 1
    private int d; // degree of the polynomial
    private int[] coeffs; // coefficients
    // operations add, sub, mul,
    // eval, in terms of rep. fields coeffs, d.
}

class Poly {
    // rep. invariant: ...
    private List<Term> terms; // terms of poly
    // operations add, sub, mul, eval, etc. in
    // terms of rep. field terms.
}
Another Example: A Meeting: Domain Concept & ADT

ADT:
Overview description:
An appointment for a meeting.
date : Date // the time
room : Integer // room number
with : Set<Person> // appt with

Set of abstract operations:
e.g., addAttendee, etc. with PoS style
specs referencing abstract specification fields

Too specific about types!
Why ADTs?

• Bridges gap between domain concept and implementation

• Formalizes domain concept, provides basis for reasoning about correctness of the implementation

• Shields client from implementation. Implementation can vary without affecting client!
An ADT is a Set of Operations

• Operations operate on data
• ADT abstracts from **organization to meaning** of data
• ADT abstracts from **structure to use**
• Data representation (implementation) doesn’t matter!

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

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

• Instead, think of a type as a **set of operations**: create, getBase, getAltitude, getBottomAngle, etc.
• Force clients to call operations to access data
Are These Types Same or Different?

- They are **different**!
- They are the **same**! Both implement the concept of the 2-d point. Goal of ADT methodology is to **express sameness**
  - Clients depend only on the **set of operations**: x(), y(), r(), theta(), etc.
  - Data representation can be changed: to change algorithms, to delay decisions, to fix bugs

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

class Point {
    float r;
    float theta;
}
```
Are These Types Same or Different?

Clients depend only on the set of operations: `add(Poly)`, `mul(Poly)`, etc.
Abstraction Barrier

Clients access the ADT through its operations. They never access the data representation.
2-d Point as an ADT

```java
class Point {
    // A 2-d point in the plane
    public float x();
    public float y();
    public float r();
    public float theta();

    // ... can be created
    public Point(); (0,0)
    public Point(float x, float y);
    public Point centroid(Set<Point> points);
}
```

Observers

Creators/Producers
2-d Point as an ADT

// class Point continued

…

// … can be moved
public translate(float delta_x, float delta_y);
public scaleAndRotate(float delta_r, float delta_theta);

}
Outline

- What is an abstract data type (ADT)?
- **Specifying an ADT**
  - immutable
  - mutable
- The ADT design methodology
- Next: reasoning about ADT implementations
Specifying an ADT

<table>
<thead>
<tr>
<th>immutable</th>
<th>mutable</th>
</tr>
</thead>
<tbody>
<tr>
<td>class TypeName</td>
<td>class TypeName</td>
</tr>
<tr>
<td>1. overview</td>
<td>1. overview</td>
</tr>
<tr>
<td>2. specification fields</td>
<td>2. specification fields</td>
</tr>
<tr>
<td>3. creators</td>
<td>3. creators</td>
</tr>
<tr>
<td>4. observers</td>
<td>4. observers</td>
</tr>
<tr>
<td>5. producers</td>
<td>5. producers (rare!)</td>
</tr>
<tr>
<td>6. mutators</td>
<td>6. mutators</td>
</tr>
</tbody>
</table>
Poly, an immutable datatype: overview

/**
 * A Poly is an **immutable** polynomial with
 * integer coefficients. A Poly is:
 * \[ c_0 + c_1x + c_2x^2 + \ldots \]
 */

class Poly {

**Overview:** Always state whether **mutable** or **immutable**

Define **abstract model** for use in specification of

operations. In ADTs state is **abstract**, not concrete (i.e.,

these are NOT actual, implementation fields of Poly, just

what we call specification fields.)
Poly, an immutable datatype: **creators**

// modifies: none  
// effects: makes a new Poly = 0
public Poly()

// modifies: none
// effect: makes a new Poly = c \times x^n
// throws: NegExponentException if n < 0
public Poly(int c, int n)

**Creators:** This is example of overloading, two Poly constructors with different signatures. New object is part of **effects** not preexisting state. Hence, **modifies** is none.
Poly, an immutable datatype: observers

// returns: degree of this polynomial
public int int degree()

// returns: the coefficient of the term of
// this polynomial, whose power is d
public int int coeff(int d)

Observers: Used to obtain information about this polynomial. Return values of other types. Never modify the abstract state!

this: the current Poly object. Also called the receiver

Poly x = new Poly(...)
c = x.coeff(3);
Poly, an immutable datatype: producers

```java
// modifies: none
// returns: a new Poly with value this + q
public Poly add(Poly q)
```

```java
// modifies: none
// returns: a new Poly with value this*q
public Poly mul(Poly q)
```

**Producers:** Operations on a type that create other objects of the same type. Common in immutable types. **No side effects**, i.e., cannot change _abstract_ values of any existing object
IntSet, a mutable datatype: overview, creators and observers

/*
 * Overview: An IntSet is a mutable, unbounded set of integers. E.g.,
 * \{ x_1, x_2, \ldots x_n \} with no duplicates
 */
class IntSet {

    // effects: makes a new empty IntSet
    public IntSet()

    // returns: true if x in this IntSet, else false
    public boolean contains(int x)
IntSet, a mutable datatype: **mutators**

```java
public void add(int x)
```

```java
public void remove(int x)
```

**Mutators**: operations that modify receiver **this**. Rarely modify anything other than this. Must list **this** in **modifies:** clause. Typically, mutators have no return value.
Exercise: String, an immutable datatype

• Overview?
• Creators?
  \[ \text{String()}, \text{String(char[]} \text{[] value}), \]
  \[ \text{String(String original)}, \ldots \]
• Observers?
  \[ \text{charAt, compareTo, contains, endsWith,} \]
  \[ \ldots \]
• Producers?
  \[ \text{concat, format, substring,} \]
  \[ \ldots \]
• Mutators?
  • None!
Exercise: The Stack datatype

```
public Stack()
public boolean empty()
public E peek()
public int search(Object o)
public E push(E item)
public E pop()
```
ADTs and Java Language Features

• Both classes and interfaces are appropriate
• Java classes
  • Operations in the ADT are public
  • Other operations are private
  • Clients can only access ADT operations
• Java interfaces
  • Clients only see the ADT operations
  • Cannot see fields
Outline

• Comparing specifications with logical formulas

• What is an abstract data type (ADT)?
• Specifying an ADT
  • immutable
  • mutable

• The ADT design methodology

• Next: reasoning about ADT implementations
Reasoning About ADTs

- ADT is a specification, a set of operations
  - E.g., contains(), add(), etc., (the IntSet ADT)
  - add(Poly q), mul(Poly q), etc., (the Poly ADT)

- When specifying ADTs, there is no mention of data representation!
- When implementing the ADT, we must select a specific data representation

- Reasoning connects implementation to specification:
  - is our implementation correct?
Implementation of an ADT is Provided by a Class

• To implement the ADT
  • We must select the representation, the rep
  • Implement concrete operations in terms of that rep
  • E.g., the rep of our Poly can be
    • a) `int[] coeffs` or
    • b) `List<Term> terms`

• Choose representation such that
  • It is possible to implement all operations
  • Most frequently used operations are efficient
Connecting Implementation to Specification

• **Representation invariant**: Object $\rightarrow$ boolean
  • Indicates whether data representation is well-formed. Only well-formed representations are meaningful
  • Defines the set of valid values

• **Abstraction function**: Object $\rightarrow$ abstract value
  • What the data structure really means
    • E.g., array [2, 3, -1] represents $-x^2 + 3x + 2$
    • How the data structure is to be interpreted
IntSet ADT

/** Overview: An IntSet is a mutable set
 * of integers. E.g., \{ x_1, x_2, \ldots \ldots x_n \}, \{\}. 
 * There are no duplicates. 
 */

// effects: makes a new empty IntSet 
public IntSet();

// modifies: this 
// effects: this_post = this_pre U \{ x \} 
public void add(Integer x);

// modifies: this 
// effects: this_post = this_pre - \{ x \} 
public void remove(Integer x);

// returns: (x in this) 
public boolean contains(Integer x)

// reruns: cardinality of this 
public int size()
class IntSet {
    private List<Integer> data
        = new ArrayList<Integer>();
    public void add(Integer x) { data.add(x); }  
    public void remove(Integer x) {
        data.remove(x);
    }
    public boolean contains(Integer x) {
        return data.contains(x)
    }
    public int size() { return data.size(); }
}
The Representation Invariant

• $s = \text{new IntSet}();$
  $s.\text{add}(1); \ s.\text{add}(1); \ s.\text{remove}(1);$

• What is wrong with this code?
• Representation invariant tells us
  ```java
  class IntSet {
      // Rep invariant: data has no nulls and no duplicates
      private List<Integer> data; ...
  }
  ```
The Representation Invariant

• States data structure **well-formedness**
  • E.g., IntSet objects, whose data array contains duplicates are not valid values

• Must hold **before** and **after** every operation!

• Correctness of operation implementation (methods in the class) depends on it!
The Representation Invariant

class IntSet {
    // Rep invariant:
    // data has no nulls and no duplicates
    private List<Integer> data = new ArrayList<Integer>();

    public void add(Integer x) {
        data.add(x);
    }
    // Rep invariant does not hold after add!
}

The Representation Invariant

class IntSet {
    // Rep invariant:
    // data has no nulls and no duplicates
    private List<Integer> data = new ArrayList<Integer>();
    public void add(Integer x) {
        if (!data.contains(x))
            data.add(x);
    }
    // Rep invariant now holds after add
}
The Representation Invariant

- Rep invariant excludes values that do not correspond to abstract values

```java
class LineSegment {
    // Rep invariant: !(x1 = x2 && y1 = y2)
    private float x1, y1; // start point
    private float x2, y2; // end point
}
```

- Conceptually, a line segment is defined by two distinct points. Thus, values with same start and end point (e.g., x1=x2=1, y1=y2=2), are meaningless. Rep invariant excludes them
The Representation Invariant

• Rep invariant excludes data representation values that do not correspond to abstract values

```java
class RightTriangle {
    // Rep invariant: 0 < angle < 90 &
    // base > 0 & base = hypot * cos(angle)
    float base, hypot, angle;
    // Objects that don’t meet the above constraints are
    // not right triangles
```
Additionally...

• Rep invariant states constraints imposed by specific data structures and algorithms
  • E.g., Tree has no cycles, array must be sorted

• Rep invariant states constraints between fields that are synchronized with each other
  • E.g., degree and coefficients fields in Poly (if we choose the array data representation)

• In general, rep invariant states correctness constraints --- if not met, implementation may behave incorrectly!
Rep Invariant Example

class Account {
    // Rep invariant:
    // balance >= 0
    // balance = Σᵢtransactions.get(i).amount
    // transactions != null
    // no nulls in transactions

    private int balance;
    // history of transactions
    private List<Transaction> transactions;
    ...
}
Another Rep Invariant Example

class NameList {
    //Rep invariant: 0 ≤ index < names.length
    int index;
    String[] names;
    
    void addName(String name) {
        index++;
        if (index < names.length)
            names[index] = name;
    }
}
More Rep Invariant Examples

```java
class Polynomial {
    //Rep. invariant: degree = coeffs.length - 1
    //... and more ...
    private int degree;
    private int[] coeffs;
    // operations add, sub, mul, eval, etc.
}
```

```java
class Polynomial {
    //Rep. invariant:
    //Poly is sum of terms...
    private List<Term> terms;
    // operations add, sub, mul, eval, etc.
}
```
Subtyping and Substitutability

• Remember, principle of substitutability: a stronger specification can be substituted for a weaker one
  • Applies to individual methods as well as ADTs

• Note: Work on ADTs and substitutability largely due to Barbara Liskov, Turing Award 2009