Specifications, conclusion.
Abstract Data Types (ADTs)

Based on notes by Michael Ernst,
University of Washington

Announcements

- Quiz 1 and 2 graded
- Labs 1 and 2 graded
  - Feedback in Submitty
- Currently grading HW0 and HW1
  - Feedback in Submitty
- HW2 is up
  - Questions: LMS discussion board or mailing list

So far

- Specifications
  - Benefits of specifications
  - Specification conventions
    - Javadoc
    - JML/Dafny
  - PoS specifications

Outline

- Specifications, conclusion
  - Specification style
  - Specification strength
  - Substitutability
  - Comparing specifications, informally
  - Comparing specifications with logical formulas
  - Converting PoS specification into logical formulas
- Abstract Data Types

PoS Specifications

- Specification convention due to Michael Ernst
- The precondition
  - requires: clause spells out constraints on client
- The postcondition
  - modifies: lists objects (typically parameters) that may be modified by the method. Any object not listed under this clause is guaranteed untouched
  - throws: lists possible exceptions
  - effects: describes final state of modified objects
  - returns: describes return value

Specification Style

- A method is called for its side effects (effects clause) or its return value (returns clause)
- It is bad style to have both effects and return
- There are exceptions. Can you think of one?
  - E.g., HashMap.put returns the previous value
  - E.g., Box.add returns true if successfully added
- Main point of spec is to be helpful
  - Being overly formal does not help
  - Being too informal is not great either
Specification Style

- A specification should be
  - Concise: not too many cases
  - Specific (strong) enough: to make guarantees
  - General (weak) enough: to permit (efficient) implementation

- Too weak: a spec imposes too many preconditions and/or gives too few guarantees
- Too strong: a spec imposes too few preconditions and/or gives too many guarantees. Burden on implementation and may hinder efficiency (e.g., is input array sorted?)

Specification Strength

- Sometimes, we need to compare specifications (we’ll see why a little later)
- “A is stronger than B” (i.e. A => B) means
  - For every implementation I
    - “I satisfies A” implies “I satisfies B”
    - The opposite is not necessarily true
  - For every client C
    - “C meets the obligations of B” implies “C meets the obligations of A”
    - The opposite is not necessarily true

Which One is Stronger?

```java
int find(int[] a, int value) {
    for (int i=0; i<a.length; i++) {
        if (a[i] == value) return i;
    }
    return -1;
}
```

- Specification A:
  - requires: a is non-null and value occurs in a
  - returns: the smallest index i such that a[i] = value
- Specification B:
  - requires: a is non-null and value occurs in a
  - returns: i such that a[i] = value

Which One is Stronger?

```java
String substring(int beginIndex)
```

- Specification A:
  - requires: 0 <= beginIndex <= length of this String object
  - returns: new string with same value as the substring beginning at beginIndex and extending until the end of the current string
- Specification B:
  - requires: nothing
  - returns: new string with same value as the substring beginning at beginIndex and extending until the end of the current string
  - throws: IndexOutOfBoundsException --- if beginIndex is negative or > length of this String object

Why Care About Specification Strength?

- Because of substitutability!

- Principle of substitutability
  - A stronger specification can always be substituted for a weaker one
  - I.e., an implementation that conforms to a stronger specification can be used in a client that expects a weaker specification

Fall 17 CSCI 2600, A Milanova (slide based on slides by Michael Ernst)
Substitutability

- Substitutability ensures correct hierarchies
- Client code: `X x; ... x.foo(index);`
  - Client is "polymorphic": written against `X`, but it is expected to work with any subclass of `X`
  - A subclass of `X`, say `Y`, may have its own implementation of `foo`, `Y.foo(int)`. Client must work correctly with `Y.foo(int)` too!
- If the spec of `Y.foo(int)` is stronger than the spec of `X.foo(int)` then we can safely substitute `Y.foo(int)` for `X.foo(int)`!

Strengthening and Weakening Specification

- Strengthen a specification
  - Require less of client: fewer/weaker conditions in `requires` clause AND/OR
  - Promise more to client: `throws`, `modifies/effects`, `returns`
- Weaken a specification
  - Require more of client: more/stronger conditions to `requires` AND/OR
  - Promise less to client: `throws`, `modifies/effects`, `returns` clauses are weaker, easier to write into code

Ease of Use by Client vs. Ease of Implementation

- A stronger specification is easier to use
  - Client has fewer preconditions to meet
  - Client gets more guarantees in postconditions
  - But a stronger spec is harder to implement!
- Weaker specification is easier to implement
  - Larger set of preconditions, relieves implementation from the burden of handling different cases
  - Easier to guarantee less in postcondition
  - But weaker spec is harder to use

Specification Strength

- Let specification `A` consist of precondition `P_A` and postcondition `Q_A`
- Let specification `B` consist of precondition `P_B` and postcondition `Q_B`
  - `A` is stronger than `B` if (but not only if!)
    - `P_B` is stronger than `P_A` (stronger specifications require less)
    - `Q_A` is stronger than `Q_B` (stronger specifications promise more)

Exercise: Order by Strength

Spec A: `requires`: a positive int argument
  `returns`: an int in `[1..10]`
Spec B: `requires`: a non-negative int argument
  `returns`: an int in `[2..5]`
Spec C: `requires`: `true` // the weakest condition
  `returns`: an int in `[2..5]`
Spec D: `requires`: an int in `[1..10]`
  `returns`: an int in `[1..20]`
Group Exercise

Spec A: "returns: an integer ≥ its argument"
Spec B: "returns: a non-negative integer ≥ its argument"
Spec C: "returns: argument + 1"
Spec D: "returns: argument^2"
Spec E: "returns: Integer.MAX_VALUE"

Implementations:

<table>
<thead>
<tr>
<th>Code</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Review, so far

- "A is stronger than B" means
  - For every implementation I
    - "I satisfies A" implies "I satisfies B"
    - The opposite is not necessarily true!
  - A larger world of implementations satisfy the weaker spec B than the stronger spec A
  - Consequently, it is easier to implement a weaker spec!
  - Weaker specs require more AND/OR
  - Weaker specs guarantee (promise) less

Outline

- Specifications, conclusion
  - Specification style
  - Specification strength
  - Substitutability
  - Comparing specifications, informally
  - Comparing specifications with logical formulas
  - Converting PoS specification into logical formulas

Comparison by Logical Formulas

The following is a sufficient condition:
If $P_B \Rightarrow P_A$ and $Q_A \Rightarrow Q_B$, then A is stronger than B

$p \Rightarrow q \Rightarrow \neg p \lor q$

A is stronger than B

Too strict a requirement

Example: int find(int[] a, int val)

- Specification B:
  - requires: a is non-null and val occurs in a
  - returns: i such that a[i] = val

- Specification A:
  - requires: a is non-null
  - returns: i such that a[i] = val if value val occurs in a and -1 if value val does not occur in a

Clearly, $P_B \Rightarrow P_A$. But $Q_A$, which states "val occurs in a => returns i such that a[i]=val AND val does not occur in a => returns -1" does not imply $Q_B$!
Comparing by Logical Formulas

(P_a => Q_a) => (P_b => Q_b) =

!(P_a => Q_a) V (P_b => Q_b) = [ due to law p => q = !p V q ]

!(P_a V Q_a) V (P_b V Q_b) = [ due to p => q = !p V q ]

(P_a ∧ !Q_a) V (P_b V Q_b) = [ due to commutativity of V ]

(P_b V Q_b) V (P_a ∧ !Q_a) = [ due to distributivity ]

([ P_b => (Q_b V P_A) ] ∧ [( P_b ∧ Q_A ) => Q_b])

A is stronger than B if and only if P_b => Q_b is true trivially or P_b implies P_A AND Q_b together with P_b imply Q_b (i.e., for the inputs permitted by P_b, Q_b holds).

Example: int find(int[] a, int val)

- Specification B:
  - requires: a is non-null and val occurs in a [ P_b ]
  - returns: i such that a[i] = val [ Q_b ]

- Specification A:
  - requires: a is non-null [ P_a ]
  - returns: i such that a[i] = val if val occurs in a and -1 if val does not occur in a [ Q_a ]

Intuition: Q_A by itself, does not imply Q_b because A may return -1. But Q_b does imply Q_b for the inputs permitted by B. Thus, it’s still ok to substitute A for B.

Converting PoS Specs into Logical Formulas

- PoS specification has 5 clauses
  - Precondition:
    - requires: ...
  - Postcondition:
    - modifies: ..., effects: ..., returns: ..., throws: ...
- To reason about PoS specs, we must first convert specs into P => Q logical formulas
  - Step 1: absorbs returns and throws into effects
  - Step 2: converts into logical formula

Convert Spec to Formula, step 1:

- PoS specification convention
  - requires: (unchanged)
  - modifies: (unchanged)
  - effects:
    - absorbed into effects
  - returns:
    - absorbs and returns are absorbed into effects E
set from java.util.ArrayList<T>

T set(int index, T element)

requires: true
modifies: this[index]
effects: this[index] = element
throws: IndexOutOfBoundsException if index < 0 || index ≥ size
returns: this

denote effects expression by E. Resulting formula is:

true \implies (E \land (\text{foreach } i \neq \text{index}, \text{this}_\text{post}[i] = \text{this}_\text{pre}[i]))
Outline

- Specifications, conclusion
  - Specification style
  - Specification strength
  - Substitutability
  - Comparing specifications, informally
  - Comparing specifications with logical formulas
  - Converting PoS specification into logical formulas
- What is an Abstract Data Type (ADT)?

Abstraction

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

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 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:
\[ c_0 + c_1x + c_2x^2 + \ldots \]

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

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

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.

```java
class Poly {
    // rep. invariant: ...
    private List<Term> terms; // terms of poly
    // operations add, sub, mul, eval, etc. in
    // terms of rep. field terms.
}
```

Why ADTs?
- Bridge 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;
}
```

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

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

```java
class Poly {
    private int d;
    private int[] coeffs;
}
```

```java
class Poly {
    private List<Term> terms;
}
```
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

Outlines

- Specifications
  - What is an Abstract Data Type (ADT)?
  - Specifying an ADT
    - immutable
    - mutable

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:

```java
/**
 * A Poly is an immutable polynomial with integer coefficients. A Poly is:
 * \[ c_0 + c_1 x + c_2 x^2 + \ldots \]
 */
class Poly {
    // Abstract state (specification fields).
    // More on this later.
}
```

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., this are NOT actual, implementation fields of Poly, just specification.)
Poly, an immutable datatype:

**creators**

*public Poly()*

// modifies: none
// effects: makes a new Poly = 0

*public Poly(int c, int n)*

// modifies: none
// effect: makes a new Poly = cx^n
// throws: NegExponentException if n < 0

_Creators:_ This is an 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**

*public int degree()*

// returns: degree of this polynomial

*public int coeff(int d)*

// returns: the coefficient of the term of // this polynomial, whose power is d

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

---

Poly, an immutable datatype:

**producers**

*public Poly add(Poly q)*

// modifies: none
// returns: a new Poly with value this + q

*public Poly mul(Poly q)*

// modifies: none
// returns: a new Poly with value this*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., *
 ** { x₁, x₂, ... xₙ }
 **
 ** class IntSet {
 **
 ** // effects: makes a new empty IntSet
 ** public IntSet() *
 **
 ** // returns: true if x in this IntSet, else false
 **
 ** public boolean contains(int x)
 **
 **
 **}
 ***/

**class IntSet**

// returns: true if x in this IntSet,
else false

(public boolean _contains_(int x))

**IntSet, a mutable datatype:**

**mutators**

// modifies: this
// effects: this_post = this_pre U { x }

*public void add(int x)*

// modifies: this
// effects: this_post = this_pre - { x }

*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?
  - String()
  - String(char[] value)
  - String(String original), ...
- Observers?
  - charAt, compareTo, contains, endsWith, ...
- Producers?
  - concat, format, substring, ...
- 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()