CSE 331
Software Design & Implementation

James Wilcox & Kevin Zatloukal

Fall 2022

Abstract Data Types (ADTs)

Comparing specifications

- Occasionally, we need to compare different specification:
  - comparing potential specifications of a new class
  - comparing new version of a specification with old
    - recall: most work is making changes to existing code

- For that, we often consider *stronger* and *weaker* specifications...

Satisfaction of a specification

Let M be an implementation and S a specification

M satisfies S if and only if

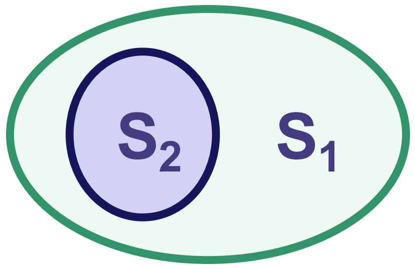
- for every input allowed by the spec precondition,
  M produces an output allowed by the spec postcondition

If M does not satisfy S, either M or S (or both!) could be “wrong”

- *“one person’s feature is another person’s bug.”*

Stronger vs Weaker Specifications

- **Definition 1**: specification S_2 is stronger than S_1 iff
  - for any implementation M: M satisfies S_2 => M satisfies S_1
  - i.e., S_2 is harder to satisfy

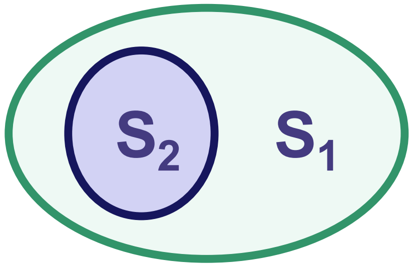


(satisfying **implementations**)

- Two specifications may be *incomparable*
  - but we are usually choosing between stronger vs weaker

Stronger vs Weaker Specifications

- An implementation satisfying a stronger specification can be **used anywhere** that a weaker specification is required
  - can **use** a method satisfying **S_2** anywhere **S_1** is expected

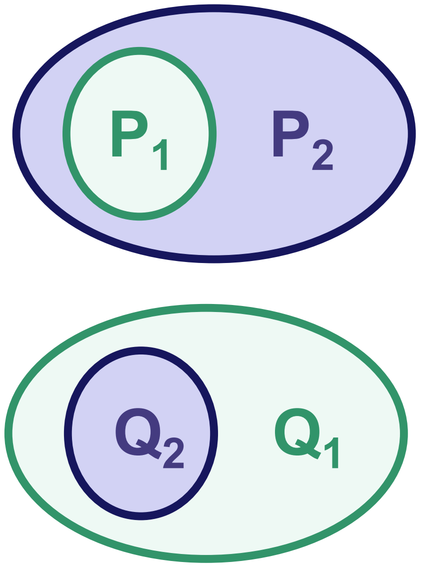


Making changes to a specification...

- changing from **S_2** to **S_1** should not break implementation
  – but it could break clients!
- changing from **S_1** to **S_2** should not break clients
  – but it could break implementation

Stronger vs Weaker Specifications

- **Definition 2**: specification S_2 is stronger than S_1 iff
  - precondition of S_2 is weaker than that of S_1
  - postcondition of S_2 is stronger than that of S_1 (on all inputs allowed by both)



- A **stronger** specification:
  - is harder to satisfy
  - gives more guarantees to the client

- A **weaker** specification:
  - is easier to satisfy
  - gives more freedom to the implementer

Example 1 (stronger postcondition)

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

Which is stronger?

- Specification A
  - requires: value occurs in **`a`**
  - returns: **`i`** such that **`a[i]`** = **`value`**
- Specification B
  - requires: value occurs in **`a`**
  - returns: *smallest* **`i`** such that **`a[i]`** = **`value`**

Example 2 (weaker precondition)

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

Which is stronger?

- Specification A
  - requires: value occurs in **`a`**
  - returns: **`i`** such that **`a[i]`** = **`value`**

- Specification C
  - returns: **`i`** such that **`a[i]`** = **`value`**, or **`-1`** if value is not in **`a`**

Example 3

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

Which is stronger?

- Specification B
  - requires: value occurs in **`a`**
  - returns: *smallest* **`i`** such that **`a[i]`** = **`value`**

- Specification C
  - returns: **`i`** such that **`a[i]`** = **`value`**, or **`-1`** if value is not in **`a`**

Strengthening a specification

- Strengthen a specification by:
  - Promising more (stronger postcondition):
    - returns clause harder to satisfy
    - effects clause harder to satisfy
    - fewer objects in modifies clause
    - more specific exceptions (subclasses)
  - Asking less of client (weaker precondition)
    - requires clause easier to satisfy
- Weaken a specification by:
  - (Opposite of everything above)

“Strange” case: @throws

Compare:

S1:

  @throws FooException if x<0

  @return x+3

S2:

  @return x+3

S3:

  @requires x >= 0

  @return x+3

- S1 & S2 are *stronger* than S3
- S1 & S2 are *incomparable* because they promise different, incomparable things when x<0

Which is better?

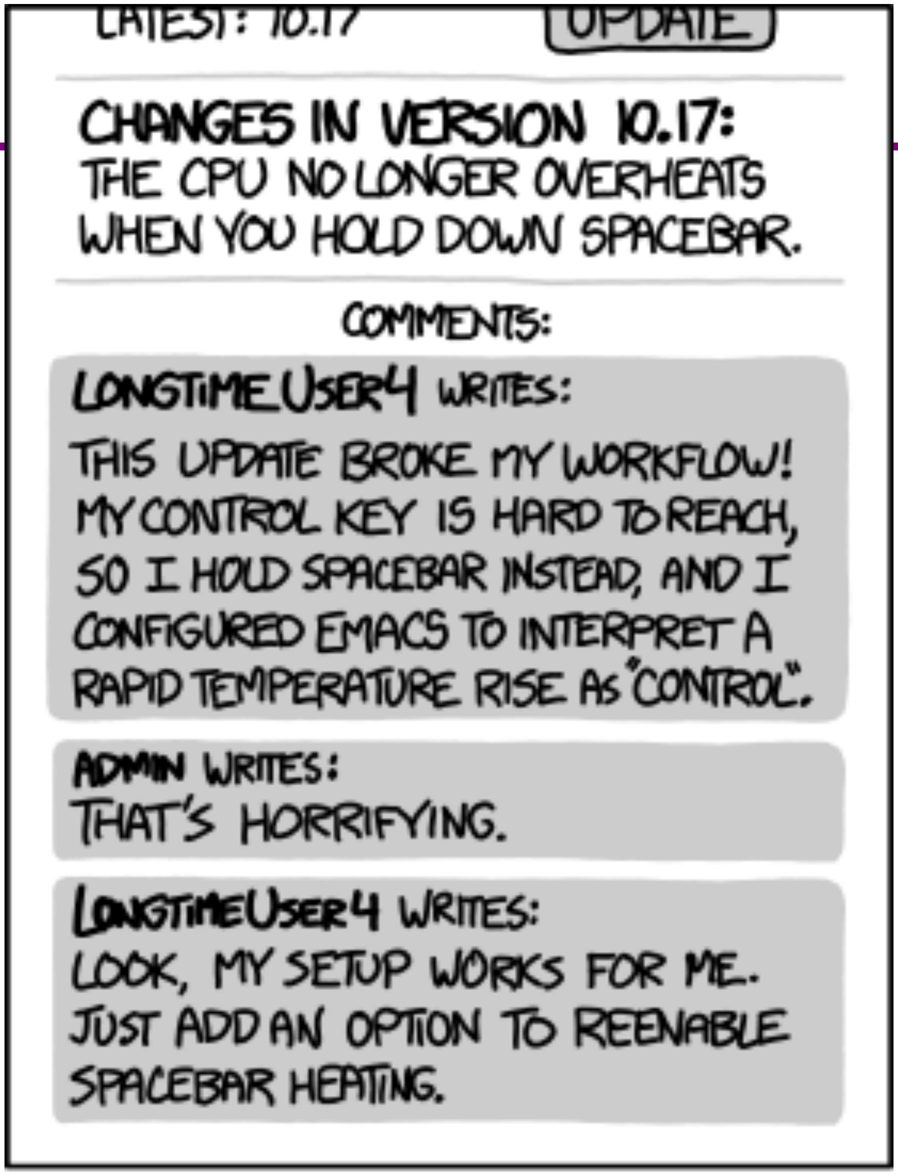
- Stronger does not always mean better!
- Weaker does not always mean better!
- Strength of specification trades off:
  - usefulness to client
  - ease of simple, efficient, correct implementation
  - promotion of reuse and modularity
  - clarity of specification itself
- "It depends"

Warnings on Specifications

Specifications are also the products of human design, so...

- They will contain **bugs**
  - (recall the central dogma of this course)
- Creating them requires **judgement**
  - no mechanical way to produce good specs (or invariants)
  - harder but good for job security
- Harder to fix the more people that use it
  - Medusa effect: “turns to stone” a bit more with each look
  - widely used parts become impossible to change

XKCD
1172


LATEST: 10.17
UPDATE
CHANGES IN VERSION 10.17:
THE CPU NO LONGER OVERHEATS WHEN YOU HOLD DOWN SPACEBAR.
COMMENTS:
LONGTIMEUSER4 WRITES:
THIS UPDATE BROKE MY WORKFLOW! MY CONTROL KEY IS HARD TO REACH, SO I HOLD SPACEBAR INSTEAD, AND I CONFIGURED EMACS TO INTERPRET A RAPID TEMPERATURE RISE AS "CONTROL".
ADMIN WRITES:
THAT'S HORRIFYING.
LONGTIMEUSER4 WRITES:
LOOK, MY SETUP WORKS FOR ME. JUST ADD AN OPTION TO REENABLE SPACEBAR HEATING.

EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

Back to Correctness…

Correctness Toolkit

- Learned forward and backward reasoning for
  - assignment
  - if statement
  - while loop

- One missing element: function calls
  - we needed specifications for that
  - now we have them

Reasoning about Function Calls

```
static int f(int a, int b) { … }
```

requires	P(a,b)	-- some assertion about a & b
returns	R(a,b,c)	-- some assertion about a, b, & c (returned)

Forward

{{ P1 }}

```
 c = f(a, b);
```

Reasoning about Function Calls

```
static int f(int a, int b) { … }
```

requires	P(a,b)	-- some assertion about a & b
returns	R(a,b,c)	-- some assertion about a, b, & c (returned)

Forward

{{ P1 }}
```
 c = f(a, b);
```
{{ P1 and R(a,b,c) }}

if P1 implies P(a,b)

Reasoning about Function Calls

```
static int f(int a, int b) { … }
```

requires	P(a,b)	-- some assertion about a & b
returns	R(a,b,c)	-- some assertion about a, b, & c (returned)

Backward

```
 c = f(a, b);
{{ Q }}
```

Reasoning about Function Calls

```
static int f(int a, int b) { … }
```

requires	P(a,b)	-- some assertion about a & b
returns	R(a,b,c)	-- some assertion about a, b, & c (returned)

Backward

```
 c = f(a, b);
```
{{ Q1 and Q2(a,b,c) }}

Reasoning about Function Calls

```
static int f(int a, int b) { … }
```

requires	P(a,b)	-- some assertion about a & b
returns	R(a,b,c)	-- some assertion about a, b, & c (returned)

Backward

```
                                  {{ Q1 and P(a,b) }}
                              ↑     c = f(a, b);
if R(a,b,c) implies Q2(a,b,c) |   {{ Q1 and Q2(a,b,c) }}
```

Importance of Specifications

Specifications are essential to **correctness**

They are also essential to **changeability**

- need to know what changes will break code using it

They are also essential to **understandability**

- need to tell readers what it is supposed to do

They are also essential to **modularity**

- need to tell clients what it will do so they can start building their own parts of the system

Reasoning about Objects

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*
- way of thinking about programs and design

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

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

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

Need a way to make our data structures **changeable**

- have this for *code* now, but not yet for data

Abstract Data Types (ADTs)

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, they are 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)

Abstract Data Type (ADT)

- invented by Barbara Liskov in the 1970s
- one of the fundamental ideas of computer science
- reduces data abstraction to procedural abstraction

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

- data structure details are hidden from the clients

Also critical for **understandability** and **modularity**

Outline

Previously looked at writing specifications 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(s):

- Documenting the *implementation* of an ADT
- Reasoning about the implementation of an ADT

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

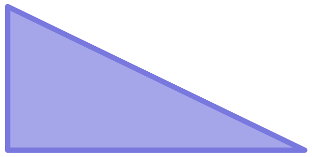
An ADT is a set of **operations**

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

- hide details of data structures such as

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

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



Think of each object as a mathematical triangle
Usable via a set of operations

`create`, `getBase`, `getArea`, …

Force clients to use operations to access data

Another Example

```
class Point {          class Point {
  float x;               float r;
  float y;               float theta;
}                      }
```

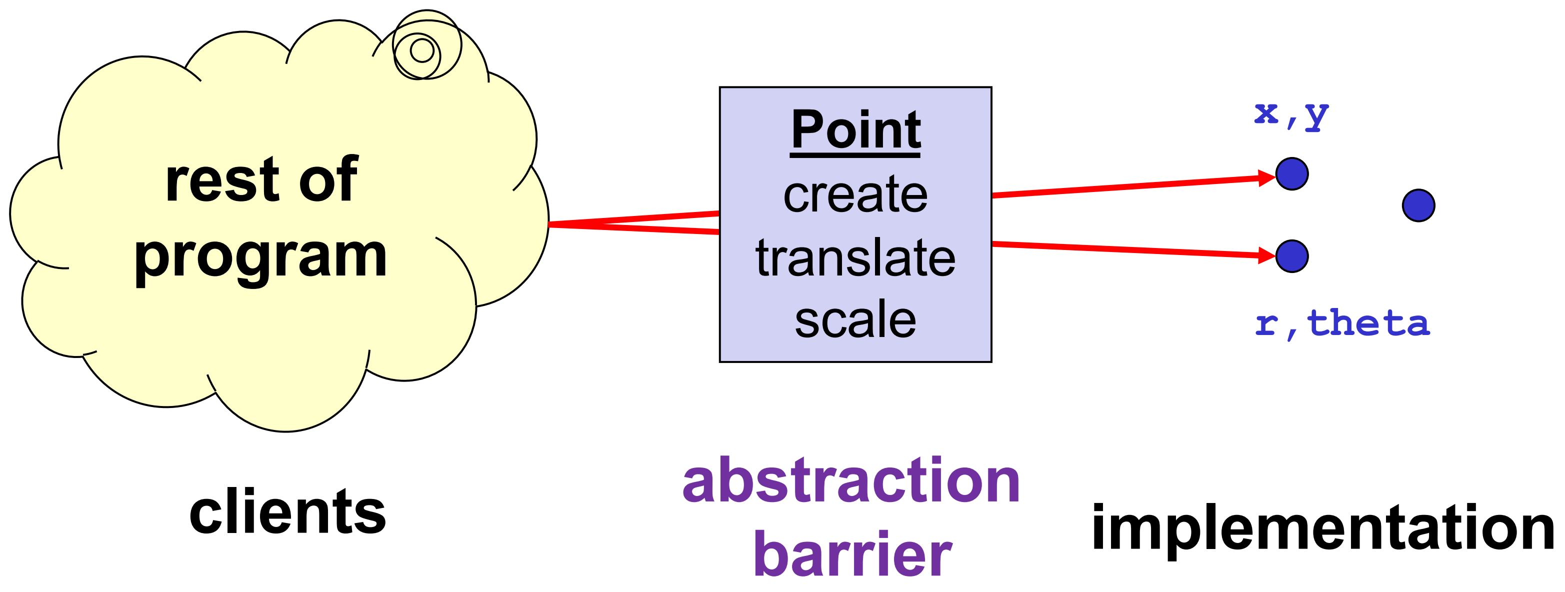
Different representations of the same concept

- both classes implement the concept “2D point”

Goal of Point ADT is to express the sameness:

- clients should think in terms of the concept “2D point”
- work with objects via operations not the representation
- produces 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

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

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`

- No information about the implementation details
  - latter called the “concrete representation”
- Note that `Point` has both field `x` and method `x()`
  - appears since it is part of the “2D point” concept
  - we are still able to change representations

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

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()
```

$\leftarrow$ "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

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
- 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