CSE 331

Software Design & Implementation

Topic: ADTs + Rep. Invariants

Discussion: What did you struggle with on HW2?

Reminders

- Great work on HW2!
- We won’t have lecture on Monday ☹

Upcoming Deadlines

- Prep. Quiz: HW3 due Tuesday (7/5)
- HW3 due Thursday (7/7)

Last Time…

- Why Specifications?
- JavaDoc
- Comparing Specifications
  - weaker benefits implementer
  - stronger benefits client
- Reasoning about Functions

Today's Agenda

- Abstract Data Types
- ADTs in Java
- Representation Invariants

Function Calls

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

{{ A }}

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

{{ A }}

c = f(a, b);

{{ A and R(a,b,c) }}

if A 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);

{{ B and Q(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

{{ B and P(a,b) }}

c = f(a, b);

{{ B and Q(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

{{ B and P(a,b) }}

c = f(a, b);

if R(a,b,c) implies Q(a, b, c) {{ B and Q(a,b,c) }}

Reasoning about Function Calls

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

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

Similar to assignment statements when the specification has `@requires` and `@return`

– Gets a little trickier when we have `@modifies` or `@effects`

Reasoning about Objects

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
4. Reasoning about the implementation of an ADT

Next lecture(s):

- Documenting the *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

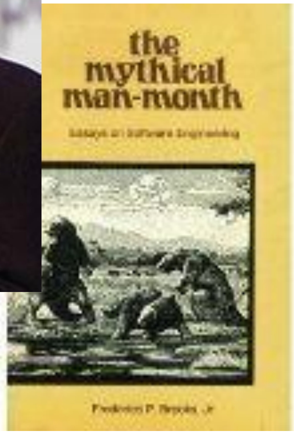
Often best to start your design by designing data…

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...

- leave room to change data structures over time
- all we really need to know is what **operations** we need to perform with the data
- the specs for those operations are the spec for 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

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

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

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 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

- data structure details are hidden from the clients

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**

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: `Math` with various mathematical methods
  - it may store data, but client does not need to think about it

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

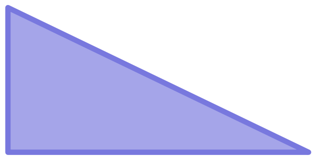
An ADT is a set of **operations**

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 {
  public float x;        public float r;
  public float y;        public float theta;
}                      }
```

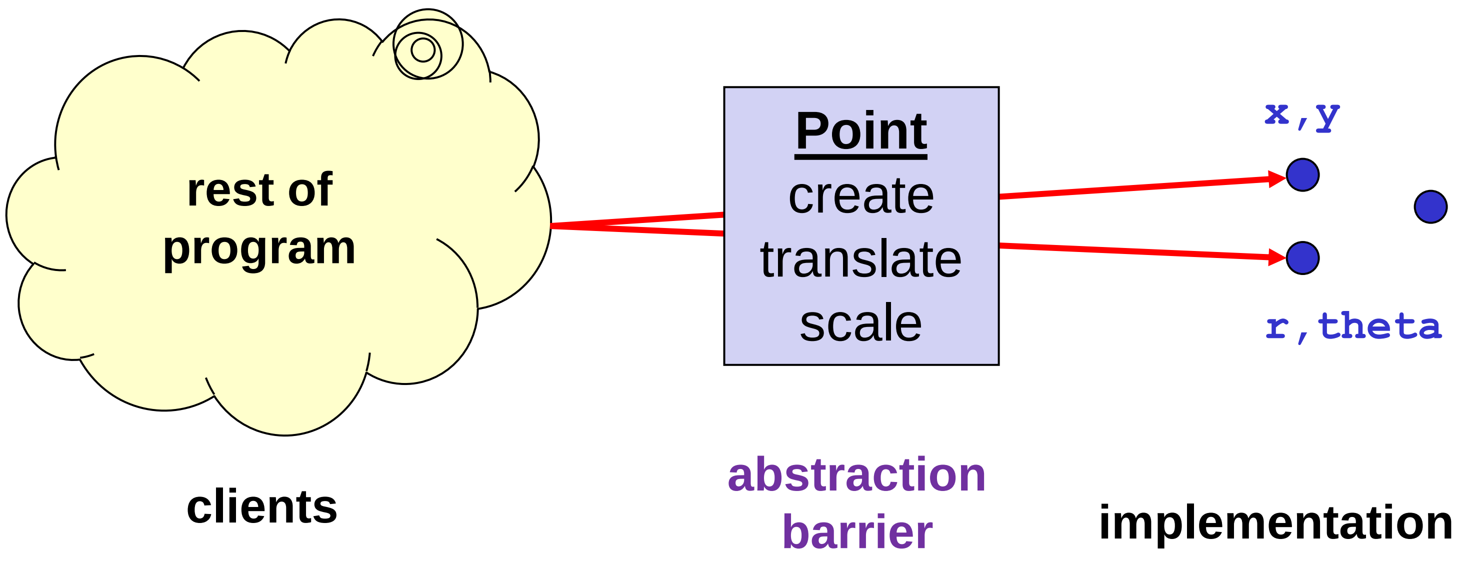
Different representations of the same concept

- 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

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();
  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	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`

- 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 (immutable): overview

```
/**
 * A Poly is an immutable polynomial with
 * integer coefficients.  A typical Poly is
 *         c0 + c1x + c2x^2 + ...
 */
class Poly {
```

$c_0 + c_1x + c_2x^2 + \ldots$

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: NegExponentException 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()

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

"this" means the abstract state

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 (mutable): 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

Specifying an ADT

Different types of methods:

1. **`creators`**
2. **`observers`**
3. **`producers`**
4. **`mutators`** (if mutable)

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

Implementing a Data Abstraction (ADT)

To implement an ADT:

- select the representation of instances
- implement operations in terms of that representation

Choose a representation so that:

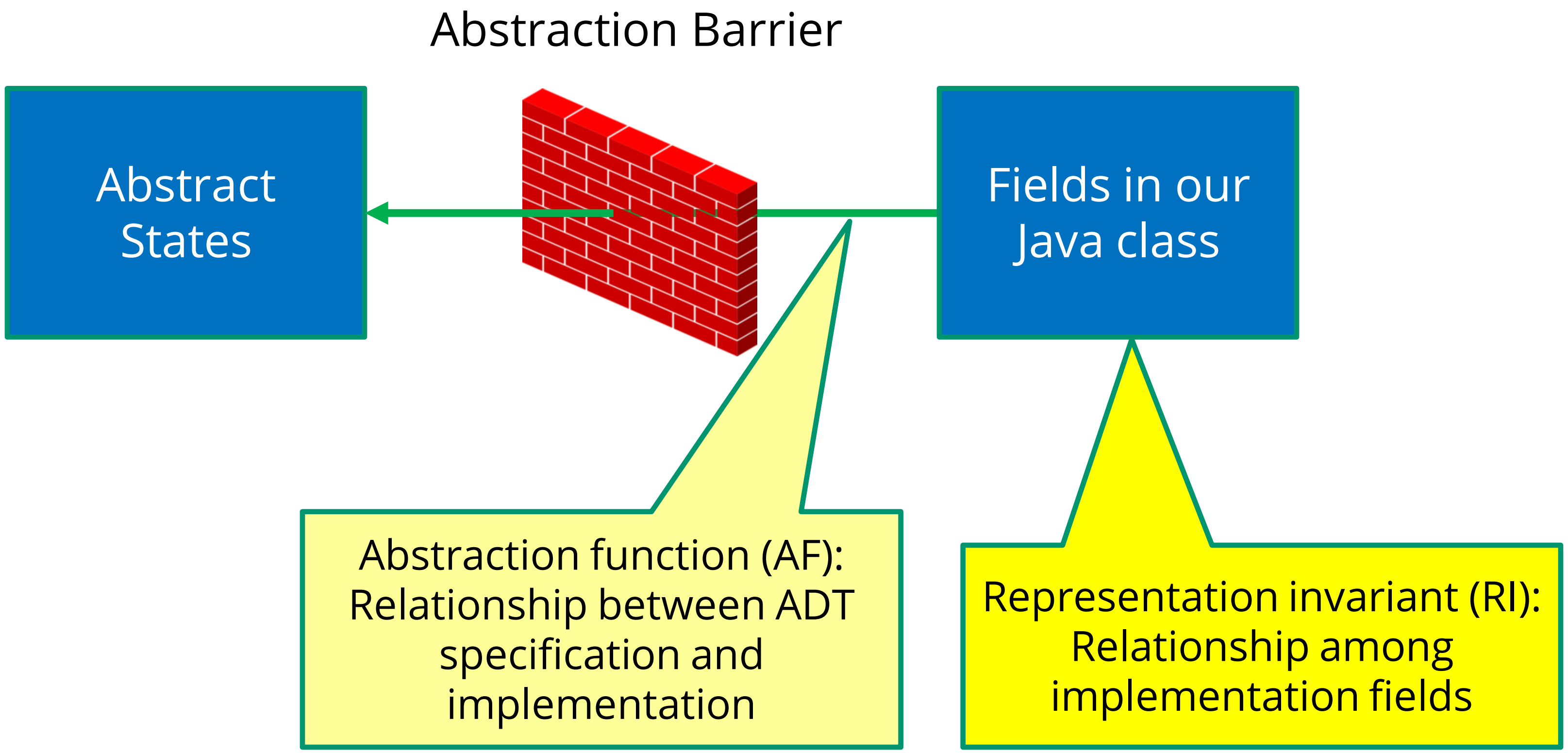
- it is possible to implement required operations
- the most frequently used operations are efficient / simple / …
  - abstraction allows the rep to change later
  - almost always better to start simple

Then use **reasoning** to verify the operations are correct

- two intellectual tools are helpful for this…

Data abstraction outline

ADT specification **ADT implementation**



Connecting implementations to specs

For implementers / debuggers / maintainers of the implementation:

Representation Invariant: maps Object → boolean

- defines the set of valid concrete values
- must hold before and after any public method is called
- **no object should *ever* violate the rep invariant**
  - such an object has no useful meaning

Abstraction Function: maps Object → abstract state

- we'll discuss this more next time!

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 {

  // Rep invariant: center != null and rad > 0
  private Point center;
  private double rad;

  // Abstraction function:
  // AF(this) = a circle with center at this.center
  //   and radius this.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 {

  // Rep invariant: center != null and edge != null
  //   and !center.equals(edge)
  private Point center, edge;

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

  //  ...
}
```

Example: Polynomial

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

  // Rep invariant: coeffs != null
  private final int[] coeffs;

  // Abstraction function:
  // AF(this) = sum of this.coeffs[i] * x^i
  //   for i = 0 .. this.coeffs.length

  // ... coeff, degree, etc.
```

Example: Polynomial 2

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

  // Rep invariant: terms != null and
  //     no two terms have the same degree and
  //     terms is sorted in descending order by degree
  private final LinkedList<IntTerm> terms;

  // Abstraction function:
  // AF(this) = sum of monomials in this.terms

  //  ... coeff, degree, etc.
```

Example: Container

```
/** A container which can reach but not exceed a given capacity */
public class Container {

  // RI: 0 <= curr <= capacity
  private int curr;
  private int capacity;

  // requires: x > 0
  // modifies: this
  // effects: adds x to this if doing so does not exceed the capacity
  public void add(int x) {
      {{ pre and RI }}
      // your code here
      {{ post and RI }}
  }
```

Before next class...

1. Start on Prep. Quiz: HW3 as early as possible!
   - Reminds you of integer base conversion
     - E.g. binary, decimal, hexadecimal
   - Reminds you how to submit your homework assignment

2. Enjoy the Monday holiday!
   - July 4^{th}, U.S. Independence Day
   - No lecture