

Abstraction Functions

Announcements

- Exam 1 on Tuesday October 6th
 - Closed book/phone/laptop
 - 2 cheat pages allowed (handwritten or typed)
 - 1 double-sided sheet or 2 single-sided
 - Reasoning about code, Specifications, ADTs
 - Review slides and practice tests available off Announcements page
 - More review problems on Friday
 - Office hours on Monday
- HW3 due Friday

2

Announcements

- HW0 and HW1 grades and feedback reports in Homework Server
- Quiz 3 grade in LMS
- Currently grading HW2

Fall 15 CSCI 2600, A. Milanova

3

Quiz 3 Problem

Which one is stronger? A B Neither

Spec A: Spec B:
 $P_A: \text{arg} \geq 1$ $P_B: \text{arg} \geq 0$
 $Q_A: 0 \leq \text{result} \leq 10$ $Q_B: 0 \leq \text{result} \leq 15$

- What's a minimal change that makes B stronger?
 - One example: change $Q_B: 0 \leq \text{result} \leq 10$
- What's a minimal change that makes A stronger?
 - One example: change $P_A: \text{arg} \geq 0$

4

Slightly different problem

Spec A
 $P_A: \text{arg} \geq 1$
 $Q_A: 0 \leq \text{result} \leq 10$

Spec B
 $P_B: \text{arg} \geq 0$
 $Q_B: 0 \leq \text{result} \leq 5 \text{ if } \text{arg} \geq 1$
 $0 \leq \text{result} \leq 15 \text{ otherwise}$

Fall 15 CSCI 2600, A. Milanova

5

Outline

- Reasoning about ADTs
 - Representation invariants (rep invariants)
 - Representation exposure
 - Checking rep invariants
 - Abstraction functions
- Review and practice problems

Fall 15 CSCI 2600, A. Milanova

6

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 representation really **means**
 - E.g., array [2, 3, -1] represents $-x^2 + 3x + 2$
 - How the data structure is to be interpreted

Fall 15 CSCI 2600, A Milanova (based on slides by Michael Ernst)

7

Rep Invariant Excludes Meaningless Concrete Values

- Disallows meaningless values
 - E.g., a `RightTriangle` cannot have `base = -2`
- Ensures fields are properly synchronized
 - `volume` and `contents` in `BallContainer`
 - `amount` and `transactions` list in `Account`
 - `degree` and `coeffs` array in `Poly`
- Ensures specific data structure constraints
 - A `Tree` cannot have cycles
- Other correctness constraints
 - `0 <= index < names.length, names != null`

Checking Rep Invariant

- **checkRep()** or **repOK()**
- Checking “no duplicates” rep invariant of `IntSet`

```
private void checkRep() {
    for (int i=0; i<data.size; i++)
        if (data.indexOf(data.elementAt(i)) != i)
            throw RuntimeException("duplicates!");
}
```
- Always check if rep invariant holds when debugging
- Leave checks anyway, if they are inexpensive

9

Practice Defensive Programming

- Assume that you will make mistakes
- Write code to catch them
 - On method entry
 - Check rep invariant (i.e., call `checkRep()`)
 - Check preconditions (requires clause)
 - On method exit
 - Check rep invariant (call `checkRep()`)
 - Check postconditions
- Checking rep invariant helps **find bugs**
- Reasoning about rep invariant helps **avoid bugs**

Fall 15 CSCI 2600, A Milanova (based on notes by Michael Ernst)

10

Outline

- Reasoning about ADTs
 - Representation invariants (rep invariants)
 - Representation exposure
 - Checking rep invariants
 - **Abstraction functions**
- Review and practice problems

Fall 15 CSCI 2600, A Milanova

11

Abstraction Function: rep \rightarrow abstract value

- The abstraction function maps **valid** concrete data representation to the abstract value it represents
- AF: Object \rightarrow abstract value
- The abstraction function lets us reason about behavior from the **client perspective**

Fall 15 CSCI 2600, A Milanova (based on a slide by Michael Ernst)

12

Abstraction Function Example

```
class Poly {
    // Rep invariant: ...
    private int[] coeffs;
    private int degree;
    // Abstraction function: coeffs [a0, a1, ..., adegree]
    // represents polynomial
    // adegreexdegree + ... + a1x + a0
    // E.g., array [-2, 1, 3] → 3x2 + x - 2
    // Empty array represents the 0 polynomial
    ...
}
```

The valid rep is the domain of the AF

The polynomial, described in terms of (abstract) specification fields, is the range of the AF

Fall 15 CSCI 2600, A. Milanova

13

Another Abstraction Function Example

```
class IntSet {
    // Rep invariant:
    // data contains no nulls and no duplicates
    private List<Integer> data;
    // Abstraction function: List [a1, a2, ..., an]
    // represents the set { a1, a2, ..., an }.
    // Empty list represents {}.
    ...
    public IntSet() ...
}
```

Fall 15 CSCI 2600, A. Milanova

14

Abstraction Function: mapping rep to abstract value

- Abstraction function: Object → abstract value
 - I.e., the object's rep maps to abstract value
 - IntSet e.g.: list [2, 3, 1] → { 1, 2, 3 }
 - Many concrete reps map to the same abstract value
 - IntSet e.g.: [2, 3, 1] → { 1, 2, 3 } and [3, 1, 2] → { 1, 2, 3 } and [1, 2, 3] → { 1, 2, 3 }
- Not a function in the opposite direction
 - One abstract value maps to many objects

Fall 15 CSCI 2600, A. Milanova

15

The IntSet Implementation

```
class IntSet {
    // Rep invariant:
    // data has no nulls and no duplicates
    private List<Integer> data
    = new ArrayList<Integer>();
    public void add(int x) {
        if (!contains(x)) data.add(x);
    }
    public void remove(int x) {
        data.remove(new Integer(x));
    }
    public boolean contains(int x) {
        return data.contains(x)
    }
    public int size() { return data.size(); }
}
```

The representation (the rep)

16

Another (Implementation of) IntSet

- What if we dropped the “no duplicates” constraint from the rep invariant
- ```
class IntSet {
 // Rep invariant: data contains no nulls
 private List<Integer> data;
 ...
}
```
- Can we still represent the concept of the IntSet? (Remember, an IntSet is a mutable set of integers with no duplicates.)

Fall 15 CSCI 2600, A. Milanova

17

## Yes. But we must change the abstraction function

```
class IntSet {
 // Rep invariant: data contains no nulls
 private List<Integer> data;
 // Abstraction function: List data
 // represents the smallest set
 // { a1, a2, ..., an } such that each ai is
 // in data. Empty list represents {}.
 ...
 public IntSet() ...
}
```

Fall 15 CSCI 2600, A. Milanova

18

## Another IntSet

```
class IntSet {
 // Rep invariant: data contains no nulls
 private List<Integer> data;
 ...
}
```

- $[1,1,2,3] \rightarrow \{1, 2, 3\}$
- $[1,2,3,1] \rightarrow \{1, 2, 3\}$

etc.

There are many objects that correspond to the same abstract value

Fall 15 CSCI 2600, A. Milanova

19

## Another IntSet

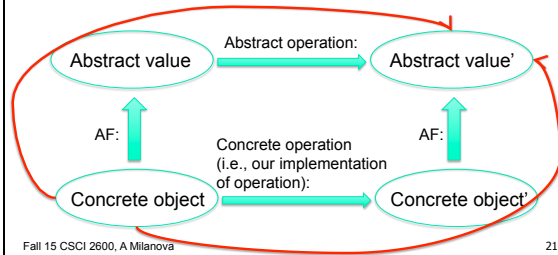
- We must change the implementation of the concrete operations as well
- What is the implication for `add(int x)` and `remove(int x)`? For `size()`?
- `add(int x)` no longer needs to check `contains(x)`. Why?
- `remove(int x)` must remove all occurrences of `x` in `data`. Why?
- What about `size()`? What else?

Fall 15 CSCI 2600, A. Milanova

20

## Correctness

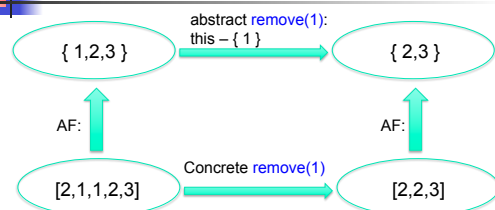
- Abstraction function allows us to reason about correctness of the implementation



Fall 15 CSCI 2600, A. Milanova

21

## IntSet Example



Creating concrete object:  
Establish rep invariant  
Establish abstraction function

After every operation:  
Maintains rep invariant  
Maintains abstraction function

Fall 15 CSCI 2600, A. Milanova

22

## Aside: the Rep Invariant

- Which implementation is better

```
class IntSet {
 // Rep invariant:
 // data has no nulls and no duplicates
 ...
 // methods establish, maintain invariant,
 // maintain consistency w.r.t. original AF
}
```

OR

```
class IntSet {
 // Rep invariant: data has no nulls
 ...
 // methods maintain this weaker invariant,
 // maintain consistency w.r.t. to new AF
}
```

23

## Aside: the Rep Invariant

- Often the rep invariant simplifies the abstraction function (specifically, it simplifies the domain of the abstraction function)
- Consequently, rep invariant simplifies implementation and reasoning!

Fall 15 CSCI 2600, A. Milanova

24

## Benevolent Side Effects

- Another implementation of `IntSet.contains`:

```
public boolean contains(int x) {
 int i = data.indexOf(x);
 if (i == -1)
 return false;
 // move-to front optimization
 // speeds up repeated membership tests
 Integer y = data.elementAt(0);
 data.set(0,x);
 data.set(i,y);
}
```
- Mutates rep, but does not change abstract value!

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

25

## ADT is a Specification

- Specification of `contains` remains  
// returns: (x in this)  
`boolean contains(int x);`
- The specification reflects modification/effects on abstract state (i.e., specification fields), not concrete state (i.e., representation fields)!

Fall 15 CSCI 2600, A Milanova

26

## Another Example: String.hashCode()

```
// returns: the hashCode value of this string
public int hashCode() {
 int h = this.hash; // rep. field hash
 if (h == 0) { // caches the hashCode
 char[] val = value;
 int len = count;
 for (int i = 0; i < len; i++) {
 h = 31*h + val[i];
 }
 this.hash = h; // modifies rep. field
 }
 return h;
}
```

Fall 15 CSCI 2600, A Milanova

27

## Writing an Abstraction Function

- The **domain** is all representations that satisfy the rep invariant
  - Rep invariant simplifies the AF by restricting its domain
- The **range** is the set of abstract values
  - Denoted with **specification fields** and **derived spec fields**
  - Relatively easy for mathematical concepts like sets
  - Trickier for “real-world” ADTs (e.g., Person, Meeting)
    - **Specification fields** and **derived specification fields** help describe abstract values

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

28

## Specification Fields

- Describe abstract values. Think of the abstract value as if it were an object with fields
- E.g., math concept of a **LineSegment**
  - **start-point** and **end-point** are **specification fields**
  - **length** is a **derived specification field**
    - Can be derived from spec fields but it's useful to have
- Range of AF and specs of operations are in terms of specification fields

29

## Specification Fields

- Often abstract values aren't clean mathematical objects
  - E.g., concept of **Customer**, **Meeting**, **Item**
  - Define those in terms of **specification fields**: e.g., a **Meeting** can be specified with specification fields **date**, **location**, **attendees**
- In general, specification fields (the specification) are different from the representation fields (the implementation)

Fall 15 CSCI 2600, A Milanova

30

## ADTs and Java Language Features

- Java classes
  - Make operations of the ADT public methods
  - Make other operations private
  - Clients can only access the ADT operations
- Java interfaces
  - Clients only see the ADT operations, nothing else
  - Multiple implementations, no code in common
  - Cannot include creators (constructors) or fields

Fall 15 CSCI 2600, A Milanova (based on slide by Michael Ernst)

31

## ADTs and Java Language Features

- Both classes and interfaces can apply
  - Write and rely upon careful specifications
- When coding, prefer interface types over specific class types
  - E.g., we used `List<Integer> data` not `ArrayList<Integer> data`.
  - Why?

Fall 15 CSCI 2600, A Milanova

32

## Implementing an ADT: Summary

- Rep invariant
  - Defines the set of valid objects (concrete values)
- Abstraction function
  - Defines, for each valid object, which abstract value it represents
- Together they modularize the implementation
  - Can reason about operations in isolation
  - Neither is part of the ADT abstraction!!!

Fall 15 CSCI 2600, A Milanova (slide by Michael Ernst)

33

## Implementing an ADT: Summary

- In practice
  - **Always write a rep invariant!**
  - Write an abstraction function when you need it
    - Write a precise but informal abstraction function
    - A formal one is hard to write, and often not that useful
    - As always with specs: we look for the balance between what is "formal enough to do reasoning" and what is "humanly readable and useful"

Fall 15 CSCI 2600, A Milanova (slide by Michael Ernst)

34

## Exercise

- The mathematical concept of the **LineSegment**
  - Choose spec fields (abstraction)
  - Choose rep fields
  - Write rep invariant
  - Write abstraction function

Fall 15 CSCI 2600, A Milanova

35

## Exercise

- Suppose we decided to represent our polynomial with a list of terms:  
`private List<Terms> terms;`
- Write the abstraction function
  - Use `e(terms[i])` to refer to exponent of  $i^{\text{th}}$  term
  - Use `c(terms[i])` to refer to coefficient of  $i^{\text{th}}$  term
- Be concise and precise

36

## Review Problems: IntMap Specification

The Overview:

```
/** An IntMap is a mapping from integers to integers.
 * It implements a subset of the functionality of Map<int, int>.
 * All operations are exactly as specified in the documentation
 * for Map.
 *
 * IntMap can be thought of as a set of key-value pairs:
 *
 * @specfield pairs = { <k1, v1>, <k2, v2>, <k3, v3>, ... }
 */
```

Fall 15 CSCI 2600, A Milanova (spec by Michael Ernst)

37

## Review Problems: IntMap Specification

```
interface IntMap {
 /** Associates specified value with specified key in this map. */
 bool put(int key, int val);
 /** Removes the mapping for the key from this map if it is present.
 */
 int remove(int key);
 /** Returns true if this map contains a mapping for the specified
 key. */
 bool containsKey(int key);
 /** Returns the value to which specified key is mapped, or 0 if this
 map contains no mapping for the key. */
 int get(int key);
}
```

Fall 15 CSCI 2600, A Milanova (spec by Michael Ernst)

38

## Review Problems: IntStack Specification

```
/**
 * An IntStack represents a stack of ints.
 * It implements a subset of the functionality of Stack<int>.
 * All operations are exactly as specified in the documentation
 * for Stack.
 *
 * IntStack can be thought of as an ordered list of ints:
 *
 * @specfield stack = [a_0, a_1, a_2, ..., a_k]
 */
```

Fall 15 CSCI 2600, A Milanova (spec by Michael Ernst)

39

## Review Problems: IntStack Specification

```
interface IntStack {
 /** Pushes an item onto the top of this stack.
 * If stack_pre = [a_0, a_1, a_2, ..., a_(k-1), a_k]
 * then stack_post = [a_0, a_1, a_2, ..., a_(k-1), a_k, val].
 */
 void push(int val);
 /**
 * Removes the int at the top of this stack and returns that int.
 * If stack_pre = [a_0, a_1, a_2, ..., a_(k-1), a_k]
 * then stack_post = [a_0, a_1, a_2, ..., a_(k-1)]
 * and the return value is a_k.
 */
 int pop();
}
```

40

## Review Problems: Rep Invariants and Abstraction Functions

- Willy Wazoo wants to write an `IntMap` but only knows how to use an `IntStack`!
- So he starts like this before he gets stuck

```
class WillysIntMap implements IntMap {
 private IntStack theRep;
 ...
}
```

- Help Willy write the rep invariant and abstraction function

Fall 15 CSCI 2600, A Milanova (problem due to Mike Ernst)

41

## Review Problems

- Now help Willy implement an `IntStack` with an `IntMap`

```
class WillysIntStack implements IntStack {
 private IntMap theRep;
 int size;
 ...
}
```

- Write a rep invariant and abstraction function

Fall 15 CSCI 2600, A Milanova

42