Designing Programs --- Understanding Data

Viera K. Proulx

College of Computer and Information Science

Northeastern University

vkp@ccs.neu.edu

- **Introduction**
  - Student Expectations: The Four Problems
- **Understanding Data**
- **Designing Computations**
- **Designing Abstractions**
- **Conclusion and Acknowledgements**

Student Expectations

Students want to know what computers can do

- How do the programs work
- How to design their own programs

I have this neat idea to make computer do ...

- Where do I start?
- How do I proceed?

TeachScheme! --- ReachJava

An Introduction to Programming and Computing

The team:

- Matthias Felleisen, Robert Bruce Findler, Matthew Flatt
- Kathryn E. Gray, Shriram Krishnamurthi, Viera K. Proulx

Part 1: TeachScheme! curriculum

- programming environment: DrScheme/HtDP
- textbook: How to Design Programs, MIT Press 2001

Part 2: ReachJava! curriculum

- programming environment: DrScheme/ProfessorJ
- textbook: How to Design Classes, MIT Press 2007 ???

Sample Problems

A Rat Race

- A rat in a cage looks for food, dies if not fed in time, or if food is poisoned.

Geometric Shapes

- Circles, squares, combinations of them: within bounds?, contains a point?, distance to the top, closer to the top?, ...

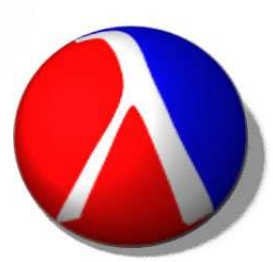
A College Registrar System

- Students, Instructors, Courses, Transcripts, Rosters, Schedules: e.g. Have all students in a course completed a prerequisite?

The Boston Marathon

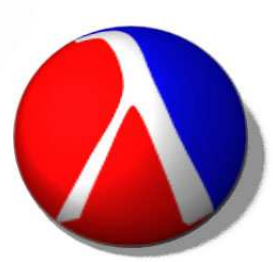
- Runners, Timing: produce results in various forms

Goals



Teach students to think through the problems:

- Analyze the information available
- Learn to represent the information as data
- Analyze the problem statement that requires a programmatic solution
- Design the programming solution systematically



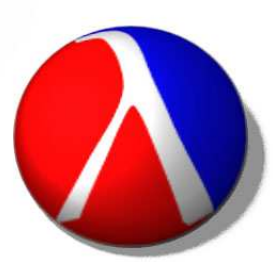
- Introduction
- Understanding Data
  - Information vs. Data
- Designing Computations
- Designing Abstractions
- Conclusion and Acknowledgements

Understanding Data

Information:

- A hungry rat in the middle of a cage 20 wide and 40 tall with 5 days to live
- A red circle of radius 10 with center at (30, 50) on top of blue square of size 20 with NW corner at (40, 20)
- Student John Dunne, id 2345, math major ... + the current schedule and a transcript
- Bill Rogers, 52 years old, male, bib number 7733 ... + the timing info: start and end times

Understanding Data



Data:

- rat-location, rat-lifespan, cage-width, cage-height
- circle-location, circle-radius, circle-color
- location-x, location-y
- square-location, square-size, square-color
- student-name, student-id, student-major, student-schedule...
- runner-name, runner-age, runner-gender, runner-bib, ...

Design Recipe for Data Definition

Does the information consists of several parts?

- Design a class of data, one field per part
- Identify the type of data needed to represent each field

If the field itself has parts, design a new class of data

... containment

- Rat: location, lifespan
- Location: x-coordinate, y-coordinate

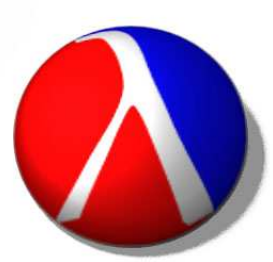
- Student: name, id, major, schedule, transcript
- Schedule: list of current-courses

Design Recipe for Data Definition

Data Definition in Scheme:

```
;; to represent a rat ...
;; A Rat is given by Posn Number
(define-struct rat (location lifespan))

(define a-rat make-rat (make-posn 10 20) 5))
( rat? a-rat) ➧ true
( rat-location a-rat) ➧ (make-posn 10 20)
( rat-lifespan a-rat) ➧ 5
```

Design Recipe for Data Definition



Data Definition in Java:

```
// to represent a rat ...
class Rat {
   Point location;
   int lifespan;
 ... constructor, methods ...}

Rat aRat = new Rat(new Point(10 20), 5);
aRat.location ➡ new Point(10 20)
aRat.lifespan ➡ 5
```

Design Recipe for Data Definition

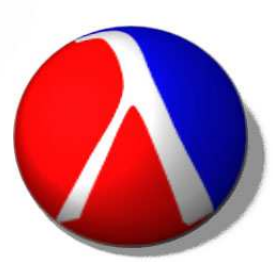
Does the information consists of several variants?

- Design a union class of data, one subclass per variant
- Shape: Circle, Square, Combo

Does one of the fields represent data in another class?

- containment (reference)
- ... possibly a self-reference (recursive definition)

Design Recipe for Data Definition



;; to represent a shape

;; A Shape **is one of:**

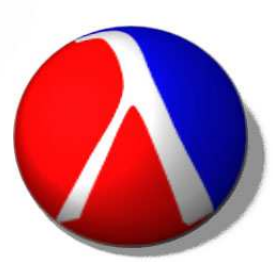
;; -- Circle: **given by** a center Point and the radius

;; -- Square: **given by** the NW Point the size

;; -- Combo: **given by** the top Shape and the bottom Shape

Data Definition - in (key)words:

Design Recipe for Data Definition



;; to represent a shape

;; A Shape **is one of:**

;; -- Circle: **given by** a center Point and the radius

;; -- Square: **given by** the NW Point the size

;; -- Combo: **given by** the top Shape and the bottom Shape

Data Definition - in (key)words:

- is given by ➡ class,
- reference to other class ➡ containment,
- is one of ➡ union
- reference to itself ➡ self-reference

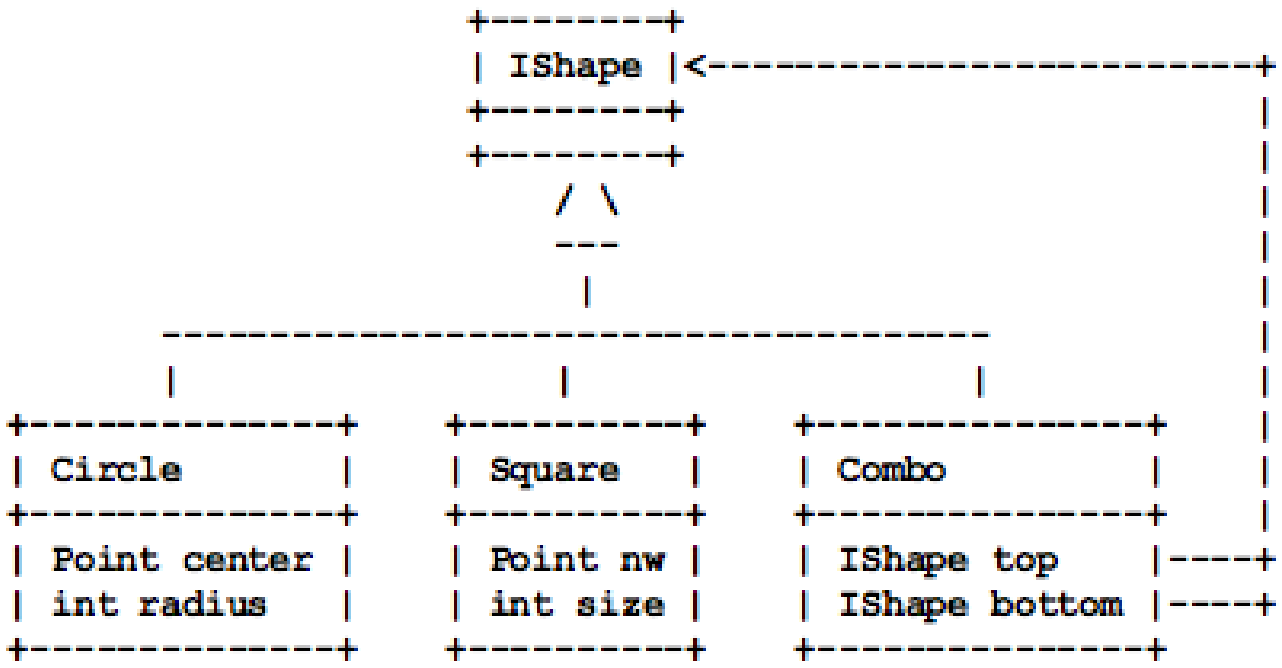
Design Recipe for Data Definition

```
;; to represent a shape
;; A Shape is one of:
;; -- Circle
;; -- Square
;; -- Combo
    ;; to represent a circle
    ;; A Circle is given by Posn Number
    (define-struct circle (center radius))
    ;; to represent a combinaton of two shapes
    ;; A Combo is given by Shape Shape
    (define-struct combo (top bottom))
```

... and include examples of data as well ...

Design Recipe for Data Definition

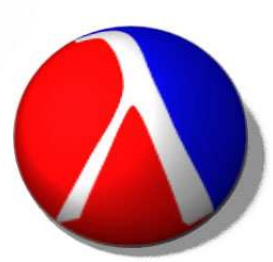
Class diagram for the IShape class hierarchy:



Corresponds exactly to the narrative data definition

Students use the diagrams to represent the data definition

Design Recipe for Data Definition



```
// to represent geometric shapes
interface IShape {
}

// to represent a circle
class Circle implements IShape {
  Point center;
  int radius;

  Circle(Point center, int radius){
    this.center = center;
    this.radius = radius;
  }
}
```

Code can be generated automatically

Design Recipe for Data Definition

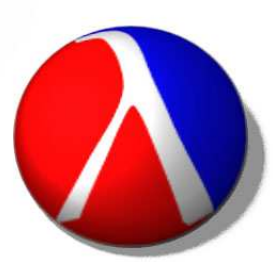
Examples of `IShape` objects

```
// Examples of geometric shapes - in the Client class

Point center = new Point(100, 100);
Point nw = new Point(120, 100);

IShape c = new Circle(this.center, 50);
IShape s = new Square(this.nw, 150);

IShape sc = new Combo(this.s, this.c);
```

Translation of data into information:

- `s` is a square with the nw corner at coordinates `(120, 100)`, size `150`

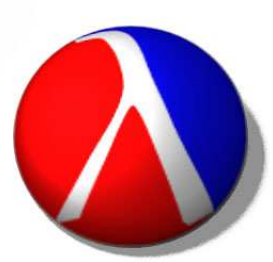
Understanding Data



Information vs. Data:

- A hungry rat in the middle of a cage 20 wide and 40 tall with 5 days to live
- (make-rat (make-posn 10 20) 5)
- A red circle of radius 10 with center at (30, 50) on top of a blue square of size 20 with NW corner at (40, 20)
- new Shape(new Circle(new Point(30, 50), 10, Color.red),
  new Square(new Point(40, 20), 20, Color.blue))

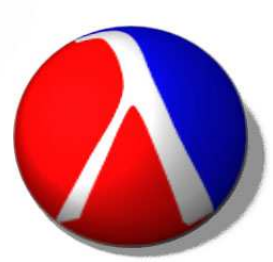
Understanding Data



Information vs. Data:

- Student John Dunne, id 2345, math major ... + the current schedule and a transcript
- (make-student "John Dunne" 2345 "math" ...)
- Bill Rogers, 52 years old, male, bib number 7733 ... + the timing info: start and end times
- new Runner("Bill Rogers", 52, true, 7733, ...)

Understanding Data



Program design depends on the data we use

Increase the complexity of both gradually

Design systematically --- student in the driver seat

Understanding data is essential regardless of the language

- **Introduction**
- **Understanding Data**
- **Designing Computations**
  - The Design Recipe
- **Designing Abstractions**
- **Conclusion and Acknowledgements**

Designing Computations

Functions vs. Procedures:

Start by focusing on programs (functions/methods) that consume data and produce a new value

- no accessors, mutators, change of state is needed
- no input, no output, just test cases
- user interaction through DrScheme and interactive graphics

Design Recipe: the steps in the design process

- Clear set of questions to answer for each step
- Outcomes that can be checked for correctness and completeness

Designing Computations

Design recipe for methods: method `contains`-- Part 1

Step 1: Problem analysis and data definition

a shape is the object that invokes the method

the user supplies the desired point

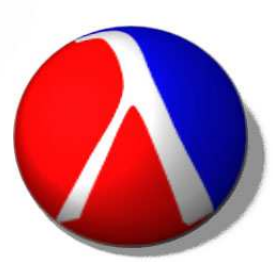
Step 2: Purpose statement and the header

- // is the given point within this shape
  boolean contains(Point p);

Step 3: Examples

- this.c.contains(new Point(90, 110)) ---> true
  this.s.contains(new Point(90, 110)) ---> false
  this.sc.contains(new Point(130, 110)) ---> true

Designing Computations



Design recipe for methods: method `contains`-- Part 2

Step 4: Template -- an inventory of available data

- // in the class Circle
  ... this.center ... -- Point
  ... this.center.distTo(p)... -- int
  ... this.radius ... -- int
  ... p ... -- Point
  ... p.distTo(Point ...) ... -- int

- // in the class Combo
  ... this.top ... -- IShape
  ... this.bottom ... -- IShape
  ... this.top.contains(p) ... -- boolean
  ... this.bottom.contains(p) ... -- boolean
  ... p ... -- Point

Designing Computations

Design recipe for methods: method `contains`-- Part 3

Step 5: Body

- // in the class Circle

```
boolean contains(Point p) {
  return this.center.distTo(p) <= this.radius;
}
```

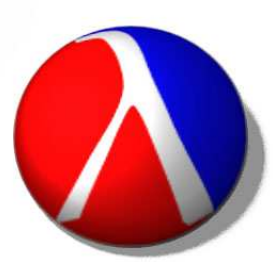
- // in the class Combo

```
boolean contains(Point p) {
  return this.top.contains(p)
      ll this.bottom.contains(p);
}
```

Step 6: Tests

- turn the examples into tests in the `Client` class and evaluate them

Designing Computations



Design Recipe: the steps in the design process:

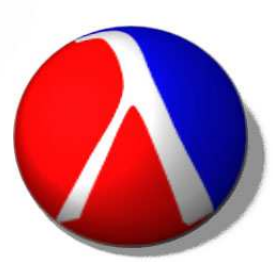
- Problem Analysis and Data Definition **-- understand**
- Purpose & Header **-- interface and documentation**
- Examples **-- show the use in context: design tests**
- Template **-- make the inventory of all available data**
- Body **-- only design the code after tests/examples**
- Test **-- convert the examples from before into tests**

Clear set of questions to answer for each step

Outcomes that can be checked for correctness and completeness

Opportunity for *pedagogical intervention*

Designing Computations

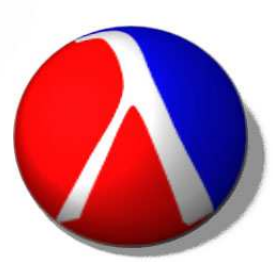


Design Recipe: the steps in the design process:

- Problem Analysis and Data Definition **-- understand**
- Purpose & Header **-- interface and documentation**
- Examples **-- show the use in context: design tests**
- Template **-- make the inventory of all available data**
- Body **-- only design the code after tests/examples**
- Test **-- convert the examples from before into tests**

Design foundation:

- Required documentation from the beginning
- Test-driven design from the beginning
- Focus on the structure of data and the structure of programs



- **Introduction**
- **Understanding Data**
- **Designing Computations**
- **Designing Abstractions**
  - Reusable Software Components
- **Conclusion and Acknowledgements**

Designing and Understanding Abstractions

Abstractions --- both in Scheme and in Java

- motivated by observing repeated code patterns
- students are taught to design abstractions

Designing and Understanding Abstractions

Abstractions --- both in Scheme and in Java

- motivated by observing repeated code patterns
- students are taught to design abstractions

Design Recipe for Abstractions:

- Identify the differences between similar solutions
- Replace the differences with parameters and rewrite the solution
- Rewrite the original examples and test them again

Designing and Understanding Abstractions

Abstracting over similarities:

- Classes with similar data ➡ super classes

Person: name, id

Student: name, id, major, transcript, schedule

Instructor: name, id, title, schedule

- (define-struct person (name id))
- (define-struct student (p-data major transcript schedule))
- (define-struct instructor (p-data title schedule))

Designing and Understanding Abstractions

Abstracting over similarities:

- Classes with similar data ➡ super classes

Person: name, id

Student: name, id, major, transcript, schedule

Instructor: name, id, title, schedule

- class Person{ String name; int id; ...}
- class Student extends Person{ String major; ...}
- class Instructor extends Person{ String dept; ...}

Designing and Understanding Abstractions

Abstracting over similarities: abstract - if methods are all different

```
abstract int distToTop();
```

In the class Circle:

```
int distToTop() {return this.location.y - this.radius;}
```

In the class Square:

```
int distToTop() {return this.location.y;}
```

In the class Combo:

```
int distToTop() {
        return min(this.top.distToTop(),
                   this.bottom.distToTop());}
```

Designing and Understanding Abstractions

Abstracting over similarities: concrete - if methods are the same

In the class Circle:

```
boolean closerToTop(IShape that){
        return this.distToTop() < that.distToTop();}
```

In the class Square:

```
boolean closerToTop(IShape that){
        return this.distToTop() < that.distToTop();}
```

In the class Combo:

```
boolean closerToTop(IShape that){
        return this.distToTop() < that.distToTop();}
```

Designing and Understanding Abstractions

Abstracting over similarities:

- Classes with the same methods ➧ methods in abstract classes

```
boolean closerToTop(IShape that) {
    return this.distToTop() < that.distToTop();}
```

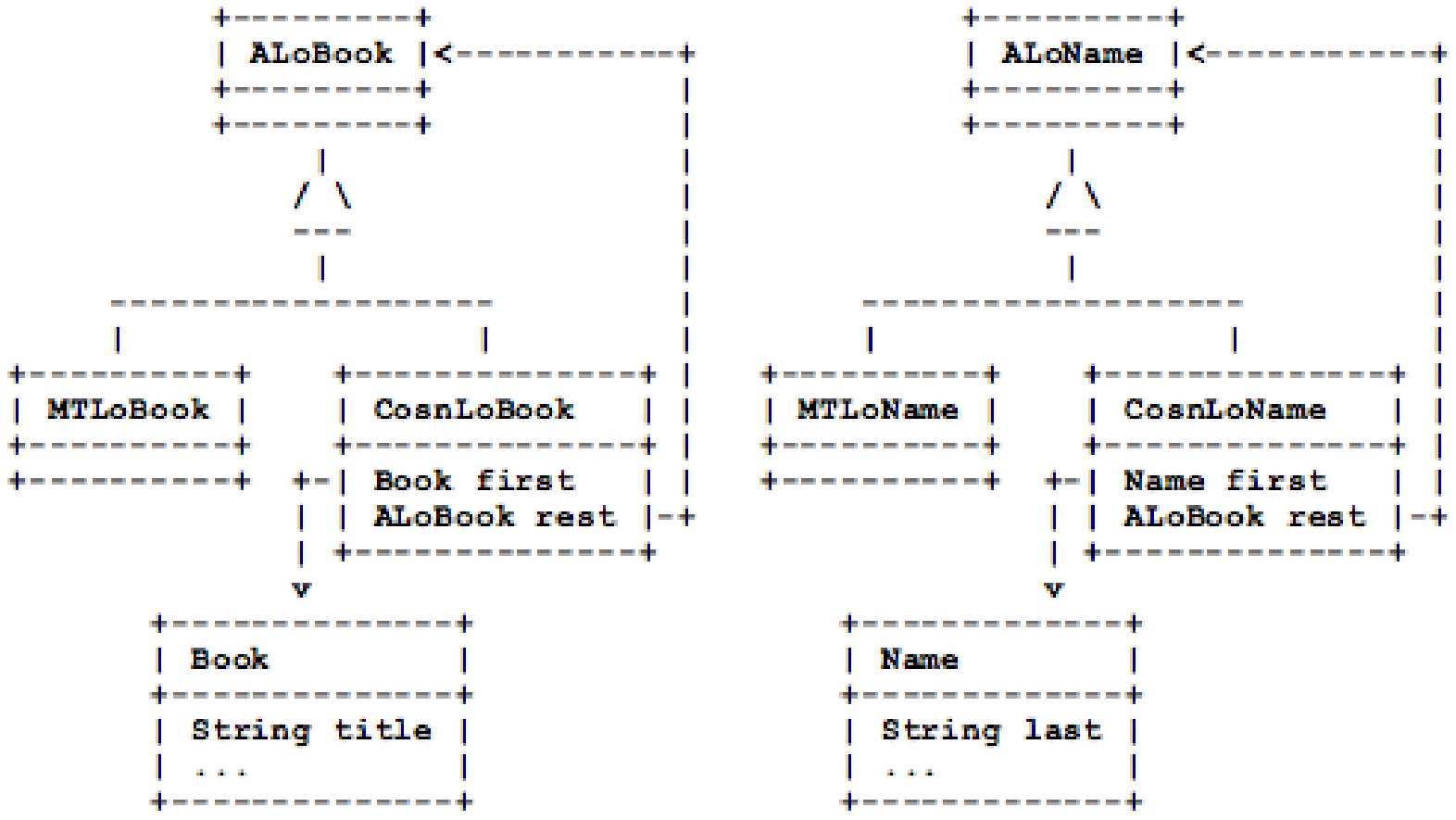
Same method in all subclasses of IShape

➧ lift as concrete method in the abstract class

Designing and Understanding Abstractions

Abstracting over similarities:

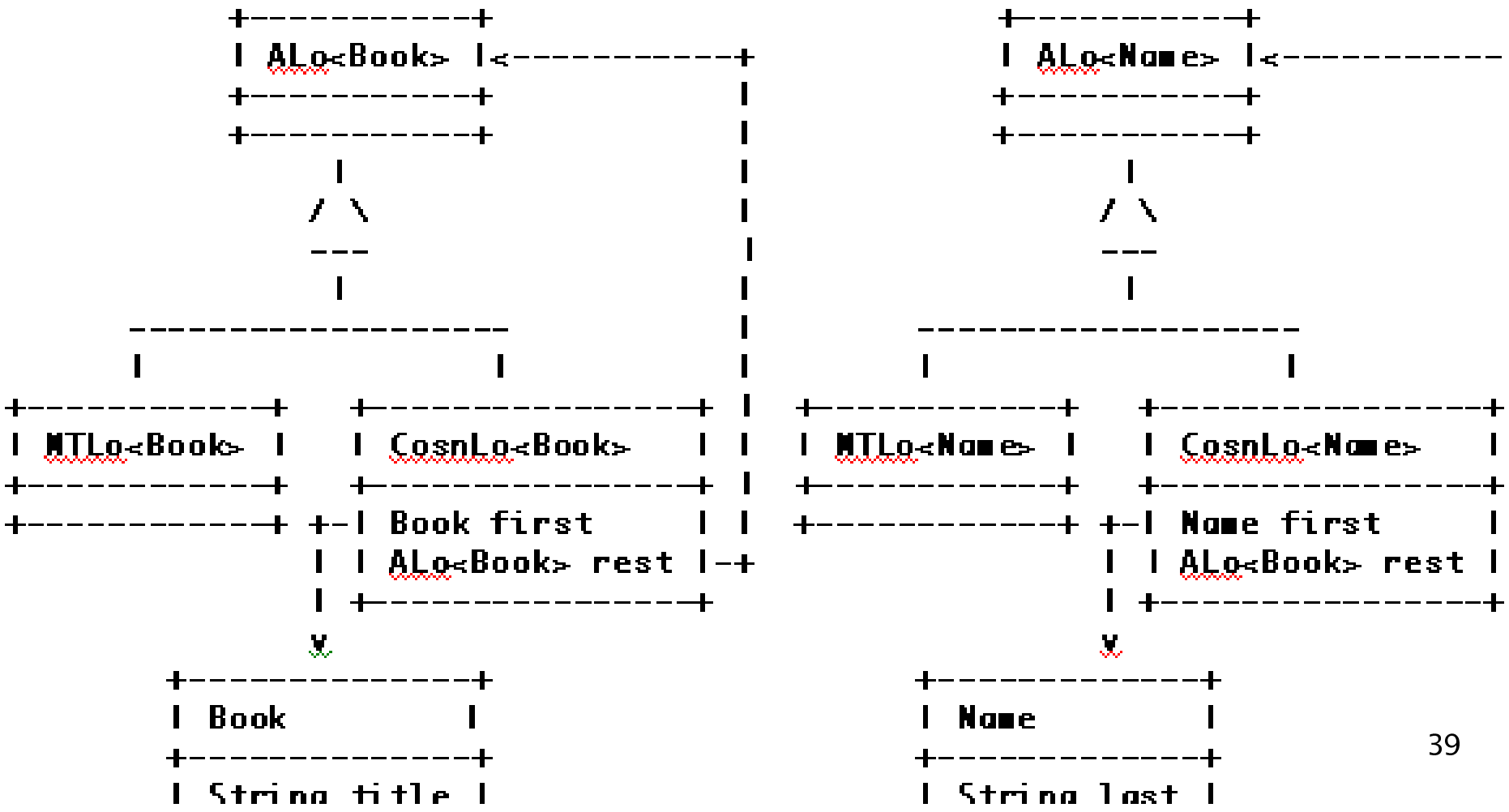
- Lists of different data ➡ list of **<T>** ➡ generics



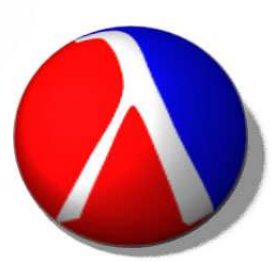
Designing and Understanding Abstractions

Abstracting over similarities:

- Lists of different data ➡ list of <T> ➡ generics



Designing and Understanding Abstractions



Abstracting over similarities:

- Classes with similar structure and methods ➧ ADTs

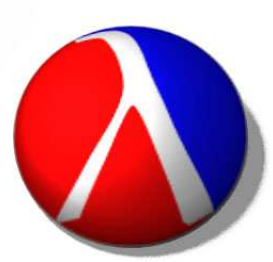
Methods for a sorted list SList:

- SList insert (Data data), Data getFirst (),
- SList removeFirst (), boolean isEmpty ()

Methods for a binary search tree BST:

- BST insert (Data data), Data getFirst (),
- BST removeFirst (), boolean isEmpty ()

Designing and Understanding Abstractions



Abstracting over similarities:

- Classes with similar structure and methods ➧ ADTs

Interface for a sorted list SortedData:

```
interface SortedData {
        SortedData insert(Data data);
        Data getFirst();
        SortedData removeFirst()
        boolean isEmpty()

abstract class SList implements SortedData{...
abstract class BST implements SortedData{...
```

Designing and Understanding Abstractions

Abstracting over similarities:

- Functions used as parameters ➧ function objects in Java

In the Boston Marathon Problem:

- make a list of all male runners under 50 years old
- make a list of all female runners under 40 years old
- make a list of all female runners over 50 years old
- make a list of all runners who finished under 2 hrs and 30 minutes

Designing and Understanding Abstractions

Abstracting over similarities: functions as parameters

In the Boston Marathon Problem:

```
AList selectSuch(Alist allRunners){
    AList result = new Alist();
    while (!allRunners.isEmpty()){
        if ( allRunners.getFirst() ???)
            result = result.add(allRunners.getFirst();
        allRunners.getRest();
    }
    return result;
}
```

Designing and Understanding Abstractions

In Scheme:

```
;; to produce a list of all runners such that ... from the given list
;; [Listof X] (X -> Boolean) -> [Listof X]
(define (filter alist predicate?)
   (cond
      [(empty? alist) empty]
      [(cons? alist)
         (cond
            [(predicate? (first alist))
               (cons (first alist)
                     (filter (rest alist) predicate?))]
            [else  (filter (rest alist) predicate?)])]))
```

Designing and Understanding Abstractions

In Java:

```
// to produce a list of all runners such that ... from the given list
AList selectSuch(Alist allRunners, ISelect choice){
    AList result = new Alist();
    while (!allRunners.isEmpty()){
        if (choice.select(allRunners.getFirst()))
            result = result.add(allRunners.getFirst();
        allRunners.getRest();
    }
    return result;
}
```

Designing and Understanding Abstractions

Abstracting over similarities: functions as parameters

```
interface ISelect{
    boolean select(); }

class QualifyingTime implements ISelect{
    boolean select(Runner r){
        return r.runningTime() < 150; }

AListRunners qualified =
            allRunners.filter(new QualifyingTime());
```

Designing and Understanding Abstractions

Abstracting over similarities: functions as parameters

Sort all marathon runners by their names

Sort all marathon runners by their running times

Sort all marathon runners by their bib numbers

interface Comparator...

Designing and Understanding Abstractions

Abstracting over similarities:

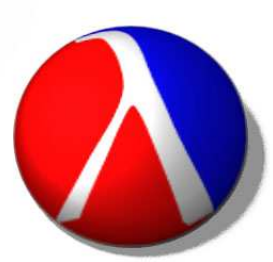
- Traversal of a container ➧ iterator

Methods:

- boolean isEmpty()
- Data getFirst()
- Traversal getRest()

Reason: provide the container class as a library -- no more changes

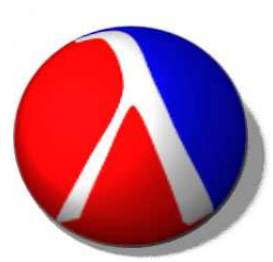
Designing and Understanding Abstractions



Abstracting over similarities:

- Classes with similar data or methods ➡ abstract classes
- Classes with same concrete methods ➡ abstract classes
- Lists of different data ➡ list of `<T>` ➡ generics
- Classes with common functional representation of structures ➡ ADTs
- Comparisons ➡ interfaces that represent a function object
- Traversal of a container ➡ iterator

Designing and Understanding Abstractions



Abstractions --- integrated throughout the course

- motivated by observing repeated code patterns
- students are taught to design abstractions

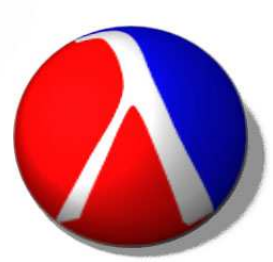
Designing and Understanding Abstractions

Abstractions --- integrated throughout the course

- motivated by observing repeated code patterns
- students are taught to design abstractions

Designing abstractions: Design Recipe for Abstractions

- Identify the differences between similar solutions
- Replace the differences with parameters and rewrite the solution
- Rewrite the original examples and test them again



- Introduction
- Understanding Data
- Designing Computations
- Designing Abstractions
- **Conclusion and Acknowledgements**

  Other Work; Our Plans

Conclusion

Solid understanding of principles

Student is the designer

Language features motivated by need

Language weaknesses exposed

Abstractions are made concrete

Exposure to alternate ways of solving problems

Understanding Mutation

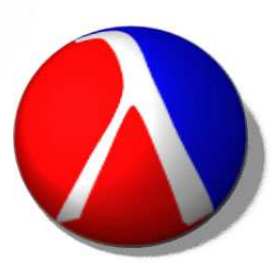
When is mutation needed

What are the dangers of using mutation

Designing tests in the presence of mutation

- The need for mutation:
  - First used to support the definition of circularly referential data
  - ArrayList - the need for mutating a structure
  - GUIs - the need to record the current state - apart from the current view
  - Efficiency - mutating sort and other algorithms

Understanding the Big Picture



The foundations are there for understanding full Java

- Study of the Java Collections Framework
- Understanding the meaning of Javadocs
- Foundations for reasoning about complexity
- Foundations for understanding the data structure tradeoffs
    - HashMap, Set, TreeMap, Linked structures
- Motivation for and using the JUnit

Our Experiences

TeachScheme! --- over 500 high schools - great results

ReachJava! --- four years at Northeastern University, many others

Instructors in follow-up courses feel students are much better prepared

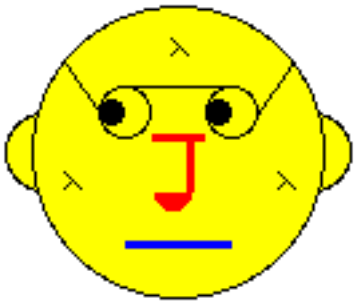
Students are much more confident in their understanding of program design

Two very successful summer workshops for secondary school and university teachers

Workshops planned for summer 2007, 2008, 2009

A growing number of followers despite the 'work in progress'

Understanding Data --- Designing Porgrams



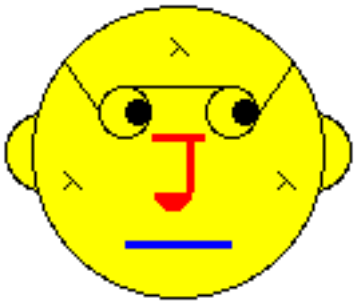
ProfessorJ

Web sites:

http://teach-scheme.org

http://www.ccs.neu.edu/home/vkp/HtDCH.html

Understanding Data --- Designing Porgrams



ProfessorJ

Web sites:

http://teach-scheme.org

http://www.ccs.neu.edu/home/vkp/HtDCH.html