Design of Class Hierarchies:

An Introduction to OO Program Design

Viera K. Proulx and Kathryn E. Gray

Northeastern University and Univesity of Utah
vkp@ccs.neu.edu and kathyg@cs.utah.edu

Pedagogy

Design Recipe

steps in the design process:

-- pedagogical intervention
-- self-regulatory learning

-- enforces documentation
-- enforces test first approach

Focus on Design

Design class hierarchies first

Design methods:
-- data driven
-- test first

Immutable data first
-- using structural recursion

Design of abstractions

Software: ProfessorJ

Language levels

Interactive environment

Targeted error-messages

Test design is supported

Design of Class Hierarchies:

An Introduction to OO Program Design

Viera K. Proulx and Kathryn E. Gray

Northeastern University and University of Utah

vkp@ccs.neu.edu and kathyg@cs.utah.edu

- **Overview**

  Our Goals, Our Team, Our Work

- **Curriculum: The Foundation**
- **ProfessorJ Languages**
- **Curriculum: The Broad View**
- **Summary**

Our Goals

Students should

- Learn to design programs
- Understand program evaluation
- Be introduced to language features as they are needed

- ... using a class-based language (such as Java)

OO Program Design: Focus on Class Hierarchies

The project:

- Comprehensive curriculum for program design using OO language
- Lecture notes, assignments, labs available; Book in prepapration, supported by software (ProfessorJ)
- Classroom tested (including software) for four years
- Summer workshops 2003 and 2004, 2006?
- CCSCNE 2005 tutorial --- **SIGCSE 2006 workshop**
- Piloted in several secondary schools and colleges

The team:

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt

Kathryn E. Gray, Shriram Krishnamurthi, Viera K. Proulx

OO Program Design: Focus on Class Hierarchies

The project:

- Comprehensive curriculum for program design using OO language
- Lecture notes, assignments, labs available; Book in prepapration, supported by software (ProfessorJ)
- Classroom tested (including software) for four years
- Summer workshops 2003 and 2004, 2006?
- CCSCNE 2005 tutorial --- **SIGCSE 2006 workshop**
- Piloted in several secondary schools and colleges

A follow up to **TeachScheme!** curriculum with **DrScheme** languages and the book **How to Design Programs**, MIT Press 2001

Our Solution

Design discipline + **Languages** and environment + **Pedagogy**

The complexity of programs grows in a systematic way:

- **The structure of the data ➡ the structure of the program**

The pedagogy: self-regulatory learning and intervention support

- **Design Recipes** guide the student and help the instructor

The tools for program design and user interactions

- **ProfessorJ** within **DrScheme**: designed to support design

Learning to design abstractions

- **Design recipe for abstractions**: rules based on examples

- **Overview**
- **Curriculum: The Foundation**

  The Focus on the Design and Pedagogy

- **ProfessorJ Languages**
- **Curriculum: The Broad View**
- **Summary**

Focus on the Design and Pedagogy

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

Pedagogical foundation:

- Self-regulatory learning:
  - Steps in the design process with clear goals, instructions on how to reach the goals, and a way to assess success.
- Support for pedagogical intervention:
  - Instructor asks at which step the student is stuck - then follows with the questions for that step.

Focus on the Design and Pedagogy

Problem: Class-based design involves two complex tasks

- the design of classes and class hierarchies
- the design of methods for these classes

Our solution: **Designing classes** before designing methods

Design Recipe for classes

- analyze the problem
- represent the information as data
- design classes of data
- define examples of instances of classes
- interpret the data as information

Focus on the Design and Pedagogy

Design recipe for designing classes:

The problem statement

- we would like to paint geometric shapes -- circles, squares, and combo-shape; see if they overlap and see if a point is inside a shape ...

Data Definition- in (key)words

- 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

Focus on the Design and Pedagogy

Design recipe for designing classes:

The problem statement

- we would like to paint geometric shapes -- circles, squares, and combo-shape; see if they overlap and see if a point is inside a shape ...

Data Definition- in (key)words

- 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

Design Recipe: class, containment, union, self-reference

Focus on the Design and Pedagogy

Class diagram for the IShape class hierarchy:

```
                     +--------+
                     | IShape |<-------------------------+
                     +--------+                          |
                     +--------+                          |
                        / \                              |
                        ---                              |
                         |                               |
       ------------------------------------              |
       |                |                 |              |
+--------------+  +----------+  +---------------+        |
| Circle       |  | Square   |  | Combo         |        |
+--------------+  +----------+  +---------------+        |
| Point center |  | Point nw |  | IShape top    |----+   |
| int radius   |  | int size |  | IShape bottom |----+   |
+--------------+  +----------+  +---------------+
```

Corresponds exactly to the narrative data definition

Students use the diagrams to represent the data definition

Focus on the Design and Pedagogy

```
// 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

Focus on the Design and Pedagogy

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, 50);

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)`**, width **`150`** and height **`50`**

Focus on the Design and Pedagogy

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
  ```

Focus on the Design and Pedagogy

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
  ```

Focus on the Design and Pedagogy

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)
        || this.bottom.contains(p);
  }
  ```

Step 6: Tests

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

Focus on the Design and Pedagogy

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*

Focus on the Design and Pedagogy

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

Focus on the Design and Pedagogy

Example of a more complex problem students can solve:

- **River with tributaries:** *pollution, lengths*
- **Binary trees:** *search trees, ancestor trees*
- **Drawing fractal curves:** *Sierpinski triangles, savannah trees*
    - using our *Canvas* and graphics library
- **Interactive games with timer and key events:** *Worm, UFO, Pong*
    - using our *World* library
- **Classes that represent Java programs:** *are the definitions valid*
- **Sorting lists, constructing sublists:** *easy tasks in our context*

and more...

Focus on the Design and Pedagogy

Programming language needs to support of the learner:

Example of a problem:

- Every method produces a value -- not void
- Assignment not needed (not allowed) at the beginning
  - however, every field has to be initialized
  - e.g. the method to move a shape image produces a new shape image:
  - // produce a shape moved by the given distance
    IShape move(int dx, int dy){...
- Testing is made easier
  - test whether the result value is as expected

- **Overview**
- **Curriculum: The Foundation**
- **ProfessorJ Languages**
  - The Languages and the Environment
- **Curriculum: The Broad View**
- **Summary**

The Languages and the Environment: The Goals

- Reduce the syntax to what is necessary
- Allow the student to focus on the key concepts
- Feedback / error messages at user's level of understanding
- Prevent misuse of advanced features
- Support a well documented test design
- Provide tools to understand program evaluation

Add new features when the need becomes compelling

The Languages and the Environment

ProfessorJ

- Within the **DrScheme** environment
- Definitions window
- Interactions window
  - Exploratory interactions: examples of objects, method invocations
  - Test outcomes
- Language levels
- Wizards to eliminate mechanical typing tasks
- Test environment
- Library to support simple graphics and event programming

The Languages and the Environment

ProfessorJ

- Within the **DrScheme** environment
- Definitions window
- Interactions window
- Language levels
    - Restricted syntax
    - Enforcement of some conventions
    - Error messages appropriate for the level.
- Wizards to eliminate mechanical typing tasks
- Test environment
- Library to support simple graphics and event programming

The Languages and the Environment

Concepts Taught in Language Levels

- Beginner
  - Classes & Methods
- Intermediate
  - Polymorphism & Abstraction
- Advanced
  - Iterative programming & APIs
- Full
  - Professional features: inner classes & exceptions

The Languages and the Environment

Beginner

- Object-oriented functional programming
  - classes and interfaces
  - recursive methods
- Removes
  - mutation
  - static
  - access modifiers -- public, private, protected
  - loops, arrays, overloading
  - inner classes & reflection

The Languages and the Environment

Intermediate

- Polymorphic Object-oriented programming
  - inheritance and overriding methods
  - casts
  - imperative programs
- Removes
  - static, access modifiers, loops & arrays
  - overloading
  - inner classes & reflection

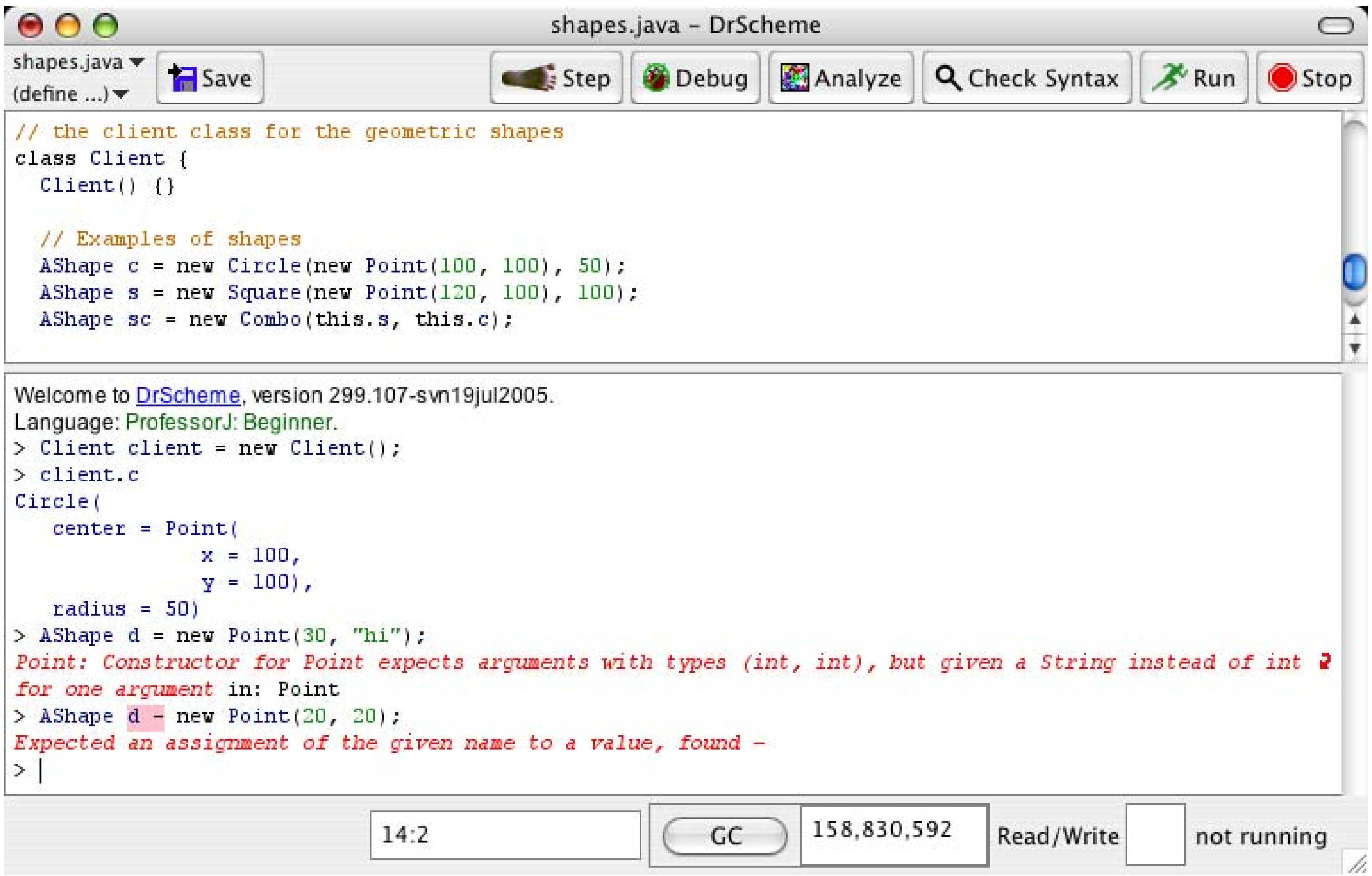
The Languages and the Environment

Advanced

- Iterative programs
  - loops & arrays
  - access controls and packages
  - overloading
  - statics
- Removes
  - inner classes & reflection
  - exceptions

The Languages and the Environment

ProfessorJ in DrScheme



- Overview
- Curriculum: The Foundation
- ProfessorJ Languages
- **Curriculum: The Broad View**
    - Abstractions, Mutation, Real Java
- Summary

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

Designing and Understanding Abstractions

Motivating abstractions

Abstracting over similarities:

- Classes with similar data ➡ abstract classes/interfaces
- Lists of different data ➡ list of **`<T>`** ➡ generics
- Classes with similar structure and methods ➡ ADTs
- Comparisons ➡ interfaces that represent a function object
- Traversal of a container ➡ iterator

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

- Overview
- Curriculum: The Foundation
- ProfessorJ Languages
- Curriculum: The Broad View
- **Summary**

  Our Experiences and Plans

Our Experiences

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

Very low attrition rate (<5%)

Students are much more confident in their understanding of program design

Two very successful summer workshops for secondary school and university teachers

Workshop planned for summer 2006

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

Web site:

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

Our Experiences

A growing number of followers:

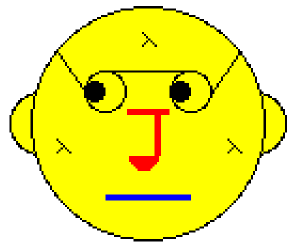
- Northeastern University, University of Utah
- University of Chicago, Worcester Polytechnic Institute
- Worcester State College, Colby College
- University of Waterloo, University of Washington
- Knox College IL, Richard Stockton College, NJ

- Weston High School, MA; Spacenkill High School, NY
- Viewpoint High School, CA; Owatonna High School, MN
- Omaha High School, NB; Oregon High School, WI

Web site:

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

How to Design Class Hierarchies

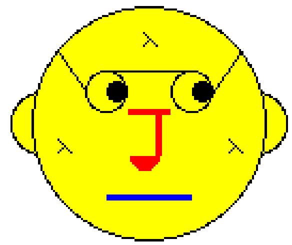


ProfessorJ

Web site:

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

How to Design Class Hierarchies



ProfessorJ

Web site:

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