Computer Science:

Designing Programs for High Schools

Viera Kr anová Proulx

College of Computer Science
Northeastern University
vkp@ccs.neu.edu

Pedagogy

Design Recipe

steps in the design process:

-- pedagogical intervention
-- self-regulatory learning

-- enforces documentation
-- enforces test first approach

Focus on Design

Design data hierarchies first

Design methods:
-- data driven
-- test first

Immutable data first
-- using structural recursion

Design of abstractions

Software: DrScheme

Language levels

Scheme-like & ProfessorJ

Interactive environment

Targeted error-messages

Test design is supported

Computer Science:

Designing Programs for High Schools

Viera Kr anová Proulx

Northeastern University

vkp@ccs.neu.edu

- **Introduction**
  - ○ Introductory Informatics in the USA and the World
  - ○ Principles vs. Artifacts
- **TeachScheme! and HtDC**
- **Design Recipe**
- **Programming Environment Support**
- **Scaling Up**
- **Conclusion**

Introduction

Introductory Informatics in the World

- Variety of curricula
- Some countries more successful than others
- Main concerns - make it relevant, yet not fashion driven
- Bring Informatics to the level of Physics and Biology

- USA is not the leader ...

Introduction

Introductory Informatics in the USA

- Local control of schools and curricula
- No nationwide certification of teachers
- CSTA - two years old

- The only common force: AP Curriculum
  - Keep up with the latest language and its features
  - Make sure it is compatible with 100 (bad) textbooks
  - Universal Introductory College Curriculum
  - No room for alternatives, innovation

  - Colleges and Universities -- same problems

Introduction

Principles vs. Artifacts

Designing a car:

- start with just the engine
- -- no gears
- -- no controls
- -- no steering
- -- no brakes
- -- no transmission

First understand the engine design well

Introduction

Principles vs. Artifacts

Learning to fly an airplane:

- start with just the simple flight control
- -- no take-offs
- -- no landings
- -- no high winds
- -- no fancy maneuvers
- -- one engine, or no engine

First understand the flight control well

Introduction

Principles vs. Artifacts

Learning to design a computer program:

- Start with the full scale commercial language
  - Syntax, complex constructs
  - Algorithms and Complex data
  - IDE e.g. Eclipse
  - I/O, GUIs, Events/Actions, Graphics
  - running, debugging, seeing the output, providing input

Student is confused from the beginning

Introduction

A Bit of History

Assembly Language Programming 15 years ago:

- Books focused on Vax, Motorola, Intel
- Details of a particular architecture -- No common principles

1994: Patterson & Hennessy: Computer Organization and Design

- End of language wars -- focus on the concepts
- Still relevant today -- even with the advances of architecture

Principles, not Artifacts

Written by top researchers

Introduction

What are the Principles of Computation?

Theory:

- Turing machines, Automata theory
    - state transitions - change the state of the world
- Church, Lambda calculus
    - functional approach: function consumes data, produces data
    - compositional: always known output for the given inputs
    - easier to understand, to test
    - programs follow the structure of data
    - Solid underlying theory -- preferred even for OO programming

Introduction

What Should we Teach?

Understand Computation

- represent information as data
- interpret data as information
- design operations that transform data
    - either imperative or functional

Design Programs

- convert reasoning about information, data, and data manipulation into a working program

-- regardless of the language

- **Introduction**
- **TeachScheme! and HtDC**
  - Overview
  - Principles
  - Resources
  - Philosophy
  - Structure
- **Design Recipe**
- **Programming Environment Support**
- **Scaling Up**
- **Conclusion**

TeachScheme! and HtDC: Overview

TeachScheme!

- Introductory curriculum with over 10 years of experience
- Used in over 300 high schools in the USA
- First semester in universities
- Summer camp for high school students
- Book: How to Design Programs -- used in Mexico, Germany, Poland, China...
  - free with online support at http://www.teach-scheme.org/

How to Design Classes (HtDC)

- builds on the TeachScheme! foundation --> OO design
- Java-like language -- over 4 years in the classrooms

TeachScheme! and HtDC: Overview

TeachScheme!

- Book: How to Design Programs -- used in Mexico, Germany, Poland, China...
    - ○ free with online support at http://www.teach-scheme.org/

How to Design Classes (HtDC)

- builds on the TeachScheme! foundation --> OO design
- Java-like language -- over 4 years in the classrooms

The team:

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt

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

TeachScheme! and HtDC: Principles

- Simple and Fun
  - Students program the actions, not I/O
- Design Discipline
  - Student learn to design systematically
- Solid Pedagogy
- Supporting Environment
  - Language levels - based on pedagogy
  - Tools: Interactions, Stepper, Test support
  - Libraries at the correct abstraction level
- Principles That Scale Up to the Real World

TeachScheme! and HtDC: Resources

Teacher Support for TeachScheme!

- Book: How to Design Programs free at http://www.teach-scheme.org/
- Summer workshops for teachers
- New exercises online - with solutions (password protected)
- New libraries added
- Mailing list - help, discussions, online community

- Testimonials

TeachScheme! and HtDC: Resources

HtDC: Curriculum for program design using OO language

- Book: How to Design Classes under development
    - Preliminary version used in several schools
    - Expected completion - this summer
- Lecture notes, labs, exercises online
- ProfessorJ series of language levels
- Libraries for simple graphics, animation, events

- Curriculum tested in classrooms for four years
- Summer workshops for teachers
- Mailing list - help, discussions, online community

TeachScheme! and HtDC: Philosophy

Game of Pong

- Ball falling down - timer controlled
- Paddle moving left/right - key controlled
- Display the ball and paddle
- Detect out-of-bounds or collision

We need to represent the ball and the paddle

- But also:
  - Frame, canvas, timer
  - key listener, graphics, speed choice

Remember the car and airplane - focus on the key idea!

TeachScheme! and HtDC: Philosophy

Focus on the Principles

- How the ball moves
- How the paddle moves
- When do they collide
- How does the game start
- How does the game resume after ball is out of play

The rest is irrelevant to the program design for a novice

- Design the game logistics systematically
- Provide simple interface for user interactions
- Program the Model, not the View

TeachScheme! and HtDC: Philosophy

Focus on the Principles

- How do you represent information as data?
- How do you interpret data as information?
- What is the operation (function, method, action) you want to model?
- What information/data do you need to perform the action?
- What do you expect as the outcome of this operation?
- What are the sub-parts of the information/data the operation uses?

Compose all of the above into a program

- At the introductory level - like the game of Pong
    - Can be done with 13 year old children ...
    - In one week ...

TeachScheme! and HtDC: Structure

The Design Recipe: The Pedagogy

- Structured description of the design process
  - Program design is divided into steps
  - Questions to ask at each step
  - Clearly defined outcome for each step
  - Enforces test-driven design, documentation

- Pedagogy
  - Self-regulatory learning: independent learner
  - Pedagogical intervention

TeachScheme! and HtDC: Structure

The Language Levels

- Full programming language is too complex
  - Start only with the necessary language constructs
  - Motivate each additional construct with need
  - At each level provide user-appropriate feedback
  - Enforce constraints that are appropriate for the novice

- TeachScheme!
  - Scheme-like series of languages - own syntax
- How to Design Classes: ProfessorJ
  - Java-like series of languages

TeachScheme! and HtDC: Structure

The Programming Environment: DrScheme

- Language levels: Scheme-like and ProfessorJ
  - error messages designed for each language level
  - Scheme syntax adjusted for novices:
  - first car rest cdr define let
- Interactions window
- Stepper for Scheme languages
- Test support for ProfessorJ

- **Introduction**
- **TeachScheme! and HtDC**
- **Design Recipe**
  - The Basics
  - Design Recipe for Data
  - Design Recipe for Functions/Methods
- **Programming Environment Support**
- **Scaling Up**
- **Conclusion**

Design Recipe: The Basics

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.

Design Recipe: The Basics

Problem: Program design involves two complex tasks

- the design of data and data hierarchies
- the design of functions/methods to manipulate the data

Our solution: **Designing data** before designing functions

Design Recipe for data hierarchies

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

Design Recipe: The Basics

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 ...

Design Recipe: The Basics

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 and the size
  - combo: given by the top shape and the bottom shape

Design Recipe: The Basics

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: The Basics

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

Design Recipe: Designing Data

```
;; to represent a geometric shape
;; A Shape is one of
;; -- (make-circle Posn Number)
;; -- (make-square Posn Number)
;; -- (make-combo Shape Shape)
```

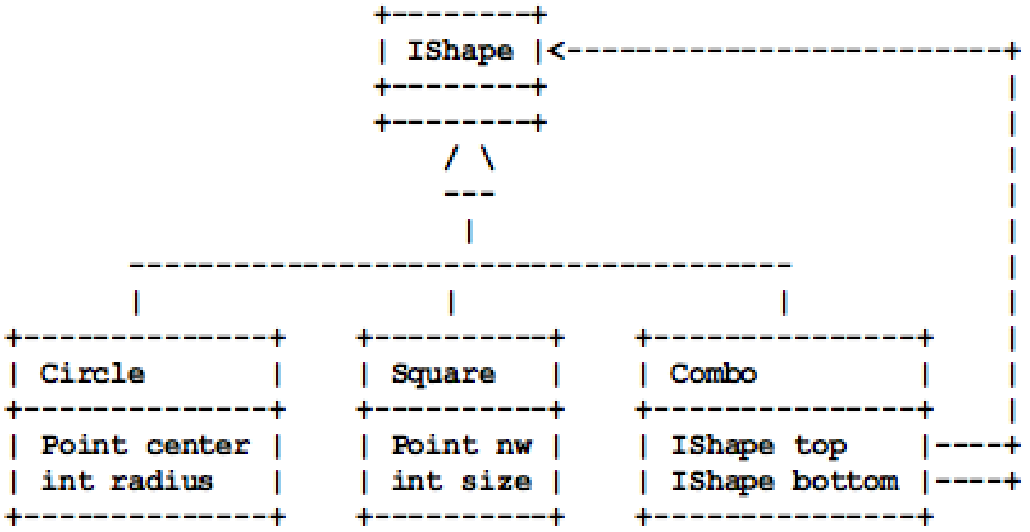
```
 (define-struct circle (center radius))
 (define-struct square (nw size))
 (define-struct combo (top bot))
```

Example:

```
 (define center (make-posn 100 100))
 (define c (make-circle center 50))
```

Design Recipe: Designing Data

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

In Scheme **`Posn`**, in Java **`Point`**

Design Recipe: Designing Data

```
// 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: Designing Data

Examples of **`Shape`** data

```
(define center (make-posn 100 100))
(define nw (make-posn 120 100))
(define c (make-circle center 50))
(define s (make-square nw 150))))
(define cs (make-combo c s))))
```

Translation of data into information:

- **`s`** is a square with the nw corner at coordinates **`(120, 100)`**, width **`150`** and height **`50`**

Design Recipe: Designing Data

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`**

Design Recipe: Designing Functions/Methods

Design Recipe: describes the steps in the design process

- Helps the student to work systematically
- Enforces good design discipline
- Build up complexity in parallel with the complexity of data
- Encourages to focus at one task per functions

Steps in the Design Recipe:

- 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 Recipe: Designing Functions/Methods

The problem statement:

We need to find out whether a point is contained in a shape.

Design recipe for functions/methods: function/method **`contains`**

Step 1: Problem analysis and data definition

- The problem deals with two pieces of data -- the point and the shape.
    - Point is a known class of data (**`Posn`** in Scheme, **`Point`** in Java) with the fields **`x`** and **`y`**
    - Shape is represented by the class of data defined earlier.
- The function/method produces a boolean value **`true`** or **`false`**

Design Recipe: Designing Functions/Methods

Step 2: Purpose statement and the header

In Scheme: the function consumes a `Posn` (predefined) and a `Shape` and produces a `Boolean`

```
;; does the given shape contain the given point?
;; Shape Posn -> Boolean
(define contains (s p) ... )
```

In Java: the method is defined in the interface `IShape` (and all of the classes that implement it). It is then invoked by an instance of a class that implements the `IShape` interface. It consumes a `Point` and produces a `boolean` value.

```
// does this shape contain the given point?
boolean contains(Point p);
```

Design Recipe: Designing Functions/Methods

Step 3: Examples

Show examples of the use of this function/method with expected outcomes.

- In Scheme --- using the earlier examples of data:
  - (contains(make-posn 90, 110) c) ---> true
    (contains(make-posn 90, 110) s) ---> false
    (contains(make-posn 90, 110) sc) ---> true

- In Java --- using the earlier examples of data:
  - this.c.contains(new Point(90, 110)) ---> true
    this.s.contains(new Point(90, 110)) ---> false
    this.sc.contains(new Point(130, 110)) ---> true

Notice the use of **`this`** to refer to the instances that invoke the method

Design Recipe: Designing Functions/Methods

Step 4: Template -- an inventory of available data

```
;; Shape Posn -> Boolean
(define contains (s p)
   ... (circle? s) ...                              ;; Boolean
     ... (circle-center s) ...                      ;; Posn
     ... (circle-radius s) ...                      ;; Number

   ... (square? s) ...                               ;; Boolean
       ...

   ... (combo? s) ...                                  ;; Boolean
     ... (combo-top s) ...                              ;; Shape
     ... (combo-bottom s) ...                           ;; Shape
     ... (contains (combo-top s) p) ...                 ;; Booelan
     ... (contains (combo-bottom s) p)...               ;; Boolean

   ... (posn-x p) ...                                ;; Number
   ... (posn-y p) ...                                ;; Number
```

Design Recipe: Designing Functions/Methods

Step 5: Function Body

```
;; Shape Posn -> Boolean
(define contains (s p)
   (cond
     [(circle? s)
         (<= (distance (circle-center s) p) (circle-radius s))]

     [(square? s)
           ...]

     [(combo? s)
         (or (contains (combo-top s) p)
               (contains (combo-bottom s) p)]))
```

Step 6: Tests

- turn the examples into tests and evaluate them

Design Recipe: Designing Functions/Methods

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

  ... // does the top shape contain the given point?
  ... this.top.contains(p) ...        -- boolean

  ... // does the bottom shape contain the given point?
  ... this.bottom.contains(p) ... -- boolean
      p                               Point
  ```

Design Recipe: Designing Functions/Methods

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

Step 5: Function 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

Design Recipe: Designing Functions/Methods

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*

Design Recipe: Designing Functions/Methods

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

Design Recipe: Designing Functions/Methods

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...

- **Introduction**
- **TeachScheme! and HtDC**
- **Design Recipe**
- **Programming Environment Support**
  - The Goals
  - DrScheme
  - Language Levels: HtDP
  - Language Levels: HtDC
  - Libraries: Graphics, Events, Timer, GUI, Web
  - Test Support
- **Scaling Up**
- **Conclusion**

Programming Environment Support

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

Programming Environment Support

DrScheme

- Full scale, yet very simple environment
- Definitions window
- Interactions window
    - Exploratory interactions: examples of data, function application
    - Test outcomes
- Language choices
    - R5S5, EOPL, Swindle, MzScheme, MrEd, FrTime, ...
    - Beginner Scheme Languages
    - ProfessorJ Languages
- Wizards, tools, libraries to help in program design

Programming Environment Support

How to Design Program Languages

Beginning Student

- a pedagogical version of Scheme that is tailored for beginning computer science students.
  - syntax forms that make the meaning clear
  - syntax forms that support clear program design

Beginning Student with List Abbreviations

- extends Beginning Student with convenient (but potentially confusing) ways to write lists, including quasiquote.

Programming Environment Support

How to Design Program Languages

Intermediate Student

- adds local bindings and higher-order functions.

Intermediate Student with Lambda

- adds anonymous functions.

Advanced Student

- adds mutable state.

Programming Environment Support

ProfessorJ

- Definitions window
  - All class definitions in one file at the start
  - Libraries/packages provided and used
- Test support
  - Compare two objects for their contents, not identity
  - Summarize the test results and diagnostics
- Interactions window
  - Exploratory interactions: examples of objects, method invocations

Programming Environment Support

ProfessorJ

- Wizards to eliminate mechanical typing tasks
- Language levels
  - Gradual increase in the complexity of the syntax and the language features
  - Students see the need for new features before they are introduced
- Library to support simple graphics and event programming
  - Copies the design of library for HtDP
  - Also available for commercial Java for easy transition

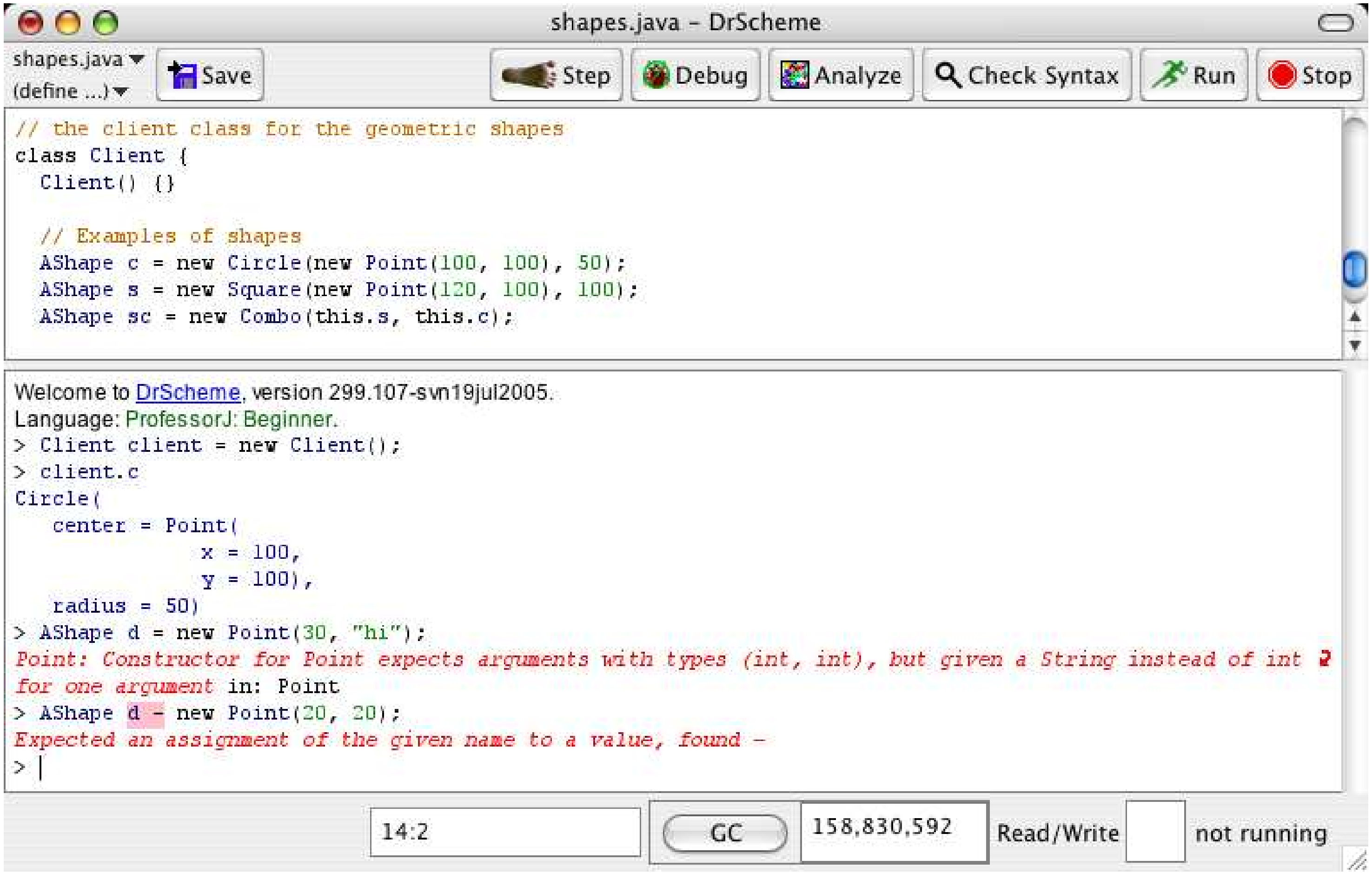
Programming Environment Support

ProfessorJ Language Levels

- Beginner Language: Classes & Methods
  - no mutation, **`static`**, access modifiers, loops, arrays, overloading, inner classes, reflection
- Intermediate Language: Polymorphism & Abstraction
  - adds inheritance and overriding methods, casts, imperative programs
- Advanced Language: Iterative programming & APIs
  - adds loops & arrays, access controls and packages, overloading, **`static`**
- Full Language
  - No plans to implement - students move on to 'real world'

The Languages and the Environment

ProfessorJ in DrScheme



```
// the client class for the geometric shapes
class Client {
  Client() {}

  // Examples of shapes
  AShape c = new Circle(new Point(100, 100), 50);
  AShape s = new Square(new Point(120, 100), 100);
  AShape sc = new Combo(this.s, this.c);
```

```
Welcome to DrScheme, version 299.107-svn19jul2005.
Language: ProfessorJ: Beginner.
> Client client = new Client();
> client.c
Circle(
   center = Point(
              x = 100,
              y = 100),
   radius = 50)
> AShape d = new Point(30, "hi");
Point: Constructor for Point expects arguments with types (int, int), but given a String instead of int
for one argument in: Point
> AShape d - new Point(20, 20);
Expected an assignment of the given name to a value, found -
> |
```

- **Introduction**
- **TeachScheme! and HtDC**
- **Design Recipe**
- **Programming Environment Support**
- **Scaling Up**
  - Desinging Abstractions Systematically
  - Understanding Mutation
  - Understanding Program and Language Design
- **Conclusion**

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 ➡ Abstract Data Types
- Comparisons ➡ interfaces that represent a function object
- Traversal of a container ➡ iterator

Designing and Understanding Abstractions

Examples of abstractions

- **Abstract classes:** *common fields, common concrete methods*
- **Generics:** *common structure of data*
  - e.g. *list of <T>*
- **Comparable, Comparator:** *common functional behavior*
- **Abstract Data Type**
  common functional representation of structures
  - *add, remove, size, contains*
- **Iterators:** *abstracting over traversals*

Understanding Mutation

Students are introduced to stateful programming when they already can design quite complex programs.

- When is mutation needed
- What are the dangers of using mutation
- Designing tests in the presence of mutation

- The need for mutation:
    - 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

Students can understand other languages, their design and structure

Understanding User Interactions

Students programmed the **model** most of the time

They see a clear separation of programming the user interactions from programming the behavior of the model

Tools to suport user interactions: **Java Power Tools**

- Clear abstractions for GUI elements design and layout
- Uniform way of reading input data from a variety of sources
- Support for data encoding for reading/writing
- Clear abstractions for event handling
- User interactions playground: Java Power Framework

Java Power Tools available at http://www.ccs.neu.edu/jpt/

- **Introduction**
- **TeachScheme! and HtDC**
- **Design Recipe**
- **Programming Environment Support**
- **Scaling Up**
- **Conclusion**
  - Our Experiences
  - Plans

Our Experiences: TeachScheme!

Over 300 high schools - student do well in following programming courses

Girls are attracted and remain in the courses

Math skills are improved

Challenges and the room to progress for the best students

Weaker students do well - learn skills and succeed

Teachers are very happy

Web site:

http://www.teach-scheme.org

Our Experiences: HtDC

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: HtDC

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

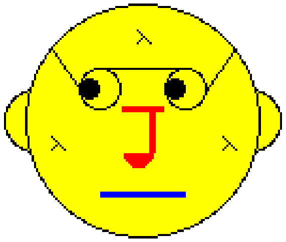
Web site:

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

Plans

- Expect to finish the HtDC textbook this year
- Plan to run one week workshops covering HtDP and HtDC for the next three years in Utah, California, New York, and Massachusetts
- Lecture notes, solutions to exercise sets, more libraties
- Full support web site
- Online community - listserve

THANK YOU



ProfessorJ

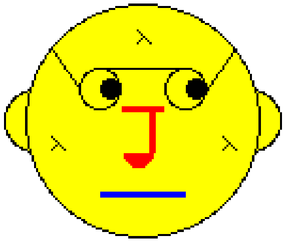
Web sites:

http://www.teach-scheme.org

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

http://www.ccs.neu.edu/jpt

THANK YOU



ProfessorJ

Web sites:

http://www.teach-scheme.org

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

http://www.ccs.neu.edu/jpt