Design of Class Hierarchies: An Introduction to OO Program Design

Viera K. Proulx
College of Computer Science
Northeastern University
Boston, MA
vkp@ccs.neu.edu

Kathryn E. Gray
School of Computing
University of Utah
Salt Lake City, UT
kathyg@cs.utah.edu

ABSTRACT

We report on the experience of teaching an introductory second semester computer science course on Fundamentals of Computer Science that uses our curriculum ***How to Design Class Hierarchies*** and the **ProfessorJ** programming languages implemented within the **DrScheme** programming environment.

This comprehensive curriculum for an introductory course focuses on principled design of class based programs in an object-oriented language (Java) with a carefully structured gradual increase in the complexity of the class structure and the programming language.

The curriculum includes extensive lecture notes, programming assignments, closed lab plans, exams, and the first part of a textbook. The curriculum is supported by a programming environment *ProfessorJ* with a series of gradually more complex teaching languages that support a novice learner. The pedagogy focuses on teaching the students problem solving and design skills that transcend the study of programming. The organization of the topics draws its strength from the theory of programming languages by focusing on the structure of data rather than on algorithms, user interactions, or arcane details of the programming language syntax.

Categories and Subject Descriptors: K.3 Computer and Information Science Education; D.1.5 Programming Techniques; D.3.3 Programming LanguagesLanguage Constructs and Features[Classes and objects]

General Terms: Program Design, Pedagogy, Programming Languages and Environments

Keywords: CS1/2, Design, Pedagogy, Programming Education, Systematic Programming

1. INTRODUCTION

Typical introductory curricula overwhelm students with a number of concepts and tricks that must be understood just to write their first program. In an object-oriented language, specifically Java, this means defining a class, its methods, and an instance of the class that is used to invoke the correct methods — all while learning to use the programming environment, which uses an industrial strength language and compiler with error messages incomprehensible to a novice, as well as the environment's mechanism for interaction. Some pedagogical programming environments (especially BlueJ [4] and DrJava [2, 14]) provide support for novice-user interactions. Other approaches have been events-first [7, 8], elementary patterns [5], graphics first [3], concrete to abstract [16], and test first [9].

Our curriculum, How to Design Class Hierarchies (**HtDCH**) addresses this problem through the structure of the programs students work on, the programming environment *ProfessorJ* [13] within *DrScheme* that provides support for the novice programmer through a series of Java-like learning languages, and through a pedagogy that focuses on disciplined program design from the first day. This curriculum is a natural follow-up to the *TeachScheme!* [11] curriculum supported by the *DrScheme* [12] series of languages and the textbook *How to Design Programs* [10]. The pedagogy of this (and our) curriculum is based on the use of the Design Recipe.

The curriculum has been used in our classrooms for the past three years resulting in a noticeable improvement of students' abilities to write programs and to reason about them [17]. It has also been successfully implemented by several high school and college instructors who have participated in our summer workshops.

1.1 HtDCH: The Structure and the Function

The key premise of object-oriented programming is that interacting objects communicate with each other and perform tasks in response to method invocation. The emphasis is on the class hierarchies that support these interactions, while the methods are typically much simpler. This is also the key premise of our curriculum. Students first focus on understanding the structure of data, and design classes and class hierarchies that represent different kinds of relationships among them.

Once students can design quite complex class hierarchies and understand how to represent information as instances of the classes they designed, they proceed with the design of methods.

Another original premise of object-oriented programming was to write programs that favor immutability [6]. Indeed, there has been a quest to eliminate the assignment statement altogether [15]. To follow this quest, for the first several

SIGCSE'06, March 1–5, 2006, Houston, Texas, USA.

weeks our students only write programs free of side effects. This is enforced by the programming environment, *ProfessorJ*, which requires that every method produces a value (not `void`), and which prohibits the use of assignment within a method body. As a consequence of these restrictions our students can very easily design tests for all methods.

1.2 HtDCH: The Pedagogy

The TeachScheme! Project [1] introduced the pedagogy of teaching program design through the use of Design Recipes. Design Recipe is a pedagogical tool that promotes self-regulatory learning [18, 19] and provides the opportunity for pedagogical interventions.

Self-regulatory learning research shows that students learn better when the task is divided into small steps, where at every step the learner has clear instructions on how to proceed, a goal to accomplish, and a way to determine if the goal has been achieved.

The Design Recipe for functions in the TeachScheme! curriculum describes such steps:

1. Analyze the problem, identify the available information, represent it as data.

2. Write down a concise purpose statement, a contract and a header for the function.

3. Make examples of the function use, with expected outcomes.

4. Write down the template: a list of all data available for your function. (For example, if an argument is a structure, list all of its components.)

5. Design the function body.

6. Convert your examples into test cases and run the tests.

Students proceed in a very structured, disciplined way, providing documentation for each method as well as practicing test-driven design. When a student encounters a problem the instructor can intervene by asking at which step of the design recipe the student got stuck. Asking further questions about that particular step in the Design Recipe guides the student in finding the solution. The intervention is focused, effective, and empowering.

Our curriculum builds on the *TeachScheme!* curriculum by defining Design Recipes tailored to the design of classes and class hierarchies as well as methods for these interacting classes.

1.3 HtDCH: Abstractions

In order to take advantage of the vast libraries of programs available in nearly every programming language one has to understand how to design and use abstractions. The Design Recipe for abstractions guides our students in moving from simple concrete solutions for specific problems to producing general solutions for a class of problems. In the process students learn the principles behind the design of abstractions, the language support for building abstractions, and the techniques for implementing the abstractions in their programs.

The specific techniques we present are interfaces, generics, function objects, iterators, abstract data types, and combinations of these. Illustrating these principles in the context of Java libraries motivates mutation and a transition from recursive style of programming to iterative programming.

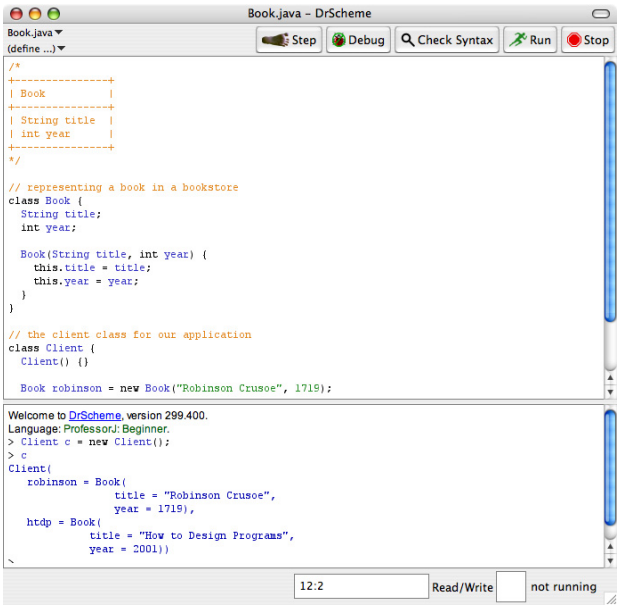

Figure 1: DrScheme – ProfessorJ.

Students are well prepared for understanding the principles and the use of the Java Collections Framework and other Java libraries, and transition easily to working with the full Java language.

2. THE CURRICULUM

2.1 Designing Class Hierarchies

Our curriculum first focuses on the design of classes that represent information. There are no methods. The Design Recipe for classes describes the questions to ask about the available information that guide the student in designing the appropriate class structure.

We begin with simple classes where all fields are either primitive types or *String*s. The class is represented both as a UML-like class diagram and as Java code. We do not introduce the visibility modifiers, and the only constructors are full constructors that initialize every field. As a result, once the student decides on the fields, their types and their names, the remainder of the task of producing the Java code can be automated. *ProfessorJ* provides a tool that generates the class diagram and the corresponding Java code from the information supplied in a GUI dialog. However, the design Recipe requires that the class definition must be followed immediately by defining examples of instances of this class in the `Client` class. *ProfessorJ*'s `Interactions` window can then display the values of these instances in a readable form.[Figure 1]

Next, we introduce classes that contain instances of other classes (e.g. a `Book` class with a field of type `Author`), and a union of classes that represent variants of a common core, (e.g. a `Shape` that can be either a `Circle` or a `Rectangle`).

The Design Recipe for designing simple classes has three steps:

1. Read the problem statement. Identify the fields needed to represent the given information. Write down your findings

as a class diagram. It will serve as our data definitions when designing classes.
2. Translate the class diagram into a class definition, adding a purpose statement to each class. The purpose statement explains to future readers what kind of information the class represents and how.
3. Make up examples of the information and represent them with instances of the class. Conversely, make up instances of the class and interpret them as information.

Similar DESIGN RECIPEs for classes with containment and classes with unions help students understand how to choose the right representation for the available information. The class diagrams translate directly into Java syntax. Additionally, the words *'consists of fields'* or *'is one of'* are used consistently to describe either a single class or a union of classes.

The design of self-referential class hierarchies such as recursively defined lists and trees follows in a straightforward manner:

```
An ancestor tree ATree is one of:
  - empty tree  (of the type MTTree)
  - a Node that contains the fields
       ancestor (of the type Ancestor)
       left     (of the type ATree)
       right    (of the type ATree)
```

At this early point in the course, students can design quite complex class hierarchies, such as representation of files and directories that contain files and directories, web pages with their components that can again be web pages, student records with schedules, transcripts, and course information, etc., as long as the data is not circularly referential.

Before writing the first method students are comfortable with a key part of the Java syntax and have a large collection of data examples that can be used to invoke and test the methods they design.

2.2 Designing Methods

The DESIGN RECIPE for methods not only guides the student through the design process, but also instills early on the need for *documenting a program* and enforces the discipline of *test-driven design*.

The DESIGN RECIPE for a method in a simple class is almost the same as the DESIGN RECIPE for functions in *TeachScheme!*. The contract and header are replaced by the method signature and the object that invoked the method is referred to as an implicit argument named *this*. Initially, students design only methods that return a value (not `void`). It is easy to design tests for these methods. Students program in a safe environment, learn good program design skills, and can easily understand the meaning of the values produced by the methods.

There are additional DESIGN RECIPEs for class hierarchies such as classes with containment, unions, or self-referential data. They include an expanded guide on how to design the template, specifically by including the methods for which at least the stubs have been already defined.

One additional step that applies to the design of all methods is the use of a wish list. If a method seems to be too complex, or it contains a task that is best performed by another class, the DESIGN RECIPE instructs students to make a wish list of the methods they may need. It is sufficient to write down the purpose statement and the header for a method in the wish list — the rest of the work can be delayed till later. The *Chain of Responsibility* design pattern is introduced and practiced early on.

Students design methods to traverse binary trees, to sort lists of objects, to analyze pollution in a river system, to produce lists of only those objects that satisfy some constraint, and many others. We also provide a pedagogical library for the design of interactive graphics such as an animated game, where the program processes key events and responds to timer ticks. The methods for drawing images produce new images that become the new scene in the game world. Similarly, the events are handled by methods *onKey(String ke)* and `onTick()` which produce new scenes in the game world.

2.3 Designing Abstractions

At this point students realize that many of the methods they write look similar. They also observe similarities between some class hierarchies, especially those that represent lists of instances of various classes. These observations lead naturally to designing abstractions. The DESIGN RECIPE for abstractions over methods asks the student to compare two methods, identify the differences, represent the difference as an additional parameter — and when done, run the tests for the original methods on the new abstracted method. Abstracting over classes, i.e. implementing a common interface for classes that are similar, follows a similar DESIGN RECIPE. Abstract classes then carry the abstractions to another level.

By this time students have seen a number of lists of different objects, such as list of `Book`s, `Person`s, `Shape`s, `WeatherRecord`s. The similarity between the structure of these classes is obvious. Introducing the abstraction that replaces a list of specific objects with a list of `Any` (or a list of `Object`) is just a natural thing to do. With Java 1.5 this leads to using generics.

The next abstraction is over the *hook* methods needed for algorithms such as sorting. When the class implements the `Comparable` interface, the sorting algorithm becomes the template part of the *Template and Hook* design pattern. If a class implements an interface that represents a predicate to select objects within a class, students can design methods to find all items that satisfy the predicate (a *filter*), methods that determine whether all items satisfy the predicate (an *andMap*), etc.

When it becomes clear that a `Book` class cannot implement the `Comparator` interface in two different ways (by title, by year), students readily embrace the function object abstraction through a class that implements the desired `compare` method. The sort method does not change, except for how it invokes the *hook* that is now supplied as a function object argument.

Abstracting the traversal of a list (or of other structures) through a functional iterator introduces more complex interfaces. (Note: The use of a functional iterator still avoids mutation.) We then add an external implementation using the *Decorator* design pattern. Exceptions are now needed to handle the attempts to invoke `current()` or `next()` on an empty iterator. An interface that represents an abstract data type arises naturally from our examples.

Having defined an iterator and function objects that provide *hook*s for algorithms, we can define a class that represents a collection of algorithms such as `filter`, `andMap`, `orMap`, `sort`, `map`, etc. It is possible to cover all these con-

cepts without introducing mutation. The focus is on the design: additional language features are added when needed.

3. PROFESSORJ

3.1 Interactions Window

The *ProfessorJ* **Interactions** window allows the user to instantiate an object in any of the classes defined in the **Definitions** window. It also allows the student to invoke methods on the instances that have been defined in the **Interactions** window.

This provides support for experimentation and quick verification of student's understanding of the expected program behavior. So, for example, students define a `Client` class that contains instances of other classes in their program. The `Client` class also contains tests for the methods defined in student's program's classes. The student can then instantiate the `Client` class in the **Interactions** window, to display the instances and run the tests. We show a sample user's interaction for the program shown in section 2.1:

```
> Client c = new Client();
> c
Client(
   htdp = Book(
             title = "How to Design Programs",
             year = 2001),
   gof = Book(
            title = "Design Patterns",
            year = 1995))
> c.htdp
Book(
   title = "How to Design Programs",
   year = 2001)
> (new Book("Effective Java", 2001)).before2000()
false
>
```

3.2 Beginner Language

When designing classes and methods, students work in a very supportive environment. *ProfessorJ* at the *Beginner* level does not allow methods that return void, does not allow mutation or local variables, does not allow overloading, and does not allow (or require) access modifiers or static members. Students can create fields that are initialized either in the constructor or at their declaration site, but these values cannot change. Finally, every field or method access within the class definition must be qualified with *this*, which helps students distinguish between method arguments and the current object.

While the structure of the class hierarchies they work with is nearly on par with the full language, the programs are restricted to the safety of immutable world with minimum of ambiguity or syntax overhead.

3.3 Intermediate Language

The *Intermediate* language of *ProfessorJ* provides support for the abstractions described in the previous section. Abstract classes are added to complete the class hierarchy infrastructure. The abstraction over lists of *any kind* is currently supported through *cast*s and the `instanceof` operator.

There are still no visibility modifiers, or `static` fields or methods. Though we use mathematical functions such as `Math.sqrt(x)`, even in the *Beginner* level, we postpone the explanation of this syntax till later.

The *Intermediate* language adds mutation. The assignment statement can now be used within method bodies and methods may have the return type `void`. Though mutation is not needed for the abstractions described in the previous section it is added to support circularly referential data.

3.4 Advanced and Beyond

Currently, the *Advanced* language level is undergoing testing, and so at this point our students transition to a commercial Java compiler and IDE. By now the students have an appropriate context for the discussion of visibility modifiers, the need for classes to be responsible for the integrity of its data, as well as the need for separating the API from the implementation. After having worked with class hierarchies with more than a dozen classes and interfaces students can confidently navigate and work with an IDE project.

4. THE TRANSITION TO *FULL* JAVA

To transition to the full Java language, students need to understand mutation, iterative (as opposed to recursion based) loops, and the use of `static` fields and methods. We also aim to guide students to become effective users of existing libraries.

The first step in this transition introduces mutation. The motivation for the mutation is presented in two different contexts. The first one is the need to define circularly referential data. If a book has several authors, our representation needs a field that contains a list of authors; at the same time, our representation of an author needs a field that represents all books written by this author. We can no longer define constructors that initialize both books and authors. The list of books written by an author must be initialized to an empty list, and as each new book is defined, the list is modified to give the author the credit for the newly published book. The effects of adding a book to an author's list of books can still be easily tested.

The second motivation for mutation comes from using a direct access data structure (either a `Vector`, or the `ArrayList`, or the `Array`). We start with `ArrayList` because it is similar to the lists we have used. We define an iterator for `ArrayList` that implements our interface for an immutable functional iterator. This allows us to define all of our earlier algorithms without modifications.

We then present the direct access methods for `ArrayList` and how to transform the recursively defined methods to iteration using either a `while` loop or a `for` loop. The DESIGN RECIPE for this transformation is a simplified version of the CPS (Continuation Passing Style) transformation.

Students are then ready to learn about the *Java Collections Framework*. It is easy to explain the need for the `Collection` interface and the `AbstractCollection` implementation of most of the methods. Students read the documentation with confident understanding of the description of the class hierarchies.

The introduction of the Java mutating `Iterator` interface and the `iterator()` method in the `Collection` interface provides the context for introducing inner classes and static fields and methods. We also design an adapter that implements our functional iterator using the methods provided by the Java `Iterator` interface — a beautiful and useful illustration of the *Adapter* design pattern.

To introduce other classes in the *Java Collections Framework* we discuss the algorithm complexity. We present prob-

lems that highlight the need for specialized data structures such as hash tables, sets, trees, and algorithms such a heapsort/priority queue, or the union/find algorithm. Our algorithm framework that allows us to select independently the specific sorting algorithm with its data representation, the source and the size of the data, and the `Comparator` used to sort the data, provides the infrastructure for stress tests of sorting algorithms. Students experience on concrete examples the differences between the algorithms, not only based on the structure of the algorithm, but also the structure of the data and the limitation of the language (such as the lack of support for tail recursion in Java).

5. CONCLUSION

5.1 Our Experiences

This curriculum has been tested in the classroom for three years, in incrementally more complete and comprehensive states. During the first year, we introduced the key design ideas and some abstractions, using the full Java language with a commercial IDE (Metrowerks). In the second year we first used *ProfessorJ* and a draft of the textbook covering the first four weeks of the course (up to abstractions). In 2005 we complemented the textbook with online lecture notes. Over the three years we have experimented with a different structure for student's test suites. Every year the course has been team taught by two or more instructors, only one of them (Viera Proulx) from the HtDCH group.

The instructors in all sections of the subsequent courses (Object-Oriented Design and Computer Organization and Programming) uniformly comment on better preparation of students who completed this curriculum. This claim is supported by data that indicate a higher success rate and a lower attrition than in similar classes prior to our change in curriculum. The most telling comment came from a student as a posting on the course newsgroup in response to some complaints about the wording of an exam question:

> Now that is completely unfair. [reply to an earlier unhappy posting] I transfered into Northeastern at the beginning of this past year. I went to a community college for a year, took 3 different programming classes there as well as an AP Computer Science class in Highschool. Now I can honestly say that in this ONE semester, I have learned more from Clement's class than all of my previous classes combined. I wish that I had no programming experiance before coming here because some old habits are hard to change.
>
> One question on one test shouldn't cause you to completely look down at an amazing course.

We additionally presented the curriculum in one-week intensive summer workshops during the summers 2003 and 2004. The participants were uniformly excited and several of the instructors proceeded to implement the curriculum in their classes - using the part of the textbook, the lecture notes, and the support of our team.

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

5.2 Acknowledgments

We would like to acknowledge Matthias Felleisen, the primary inventor of this curriculum, and Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi our co-authors on the book, for their contributions to the curriculum.

6. REFERENCES

[1] http://www.teach-scheme.org/.

[2] E. Allen, R. Cartwright, and B. Stoller. DrJava: A lightweight pedagogic environment for Java. *SIGCSE Bulletin and Proceedings*, 34(1):137–141, 2002.

[3] C. Alphonce and P. Ventura. Using graphics to support the teaching of fundamental object-oriented principles in CS1. In *Companion of the 18th annual ACM OOPSLA Conference, Anaheim, CA*, pp 156–161, Oct, 2003.

[4] D. J. Barnes and M. Koelling. *Objects First with Java: A Practical Introduction Using BlueJ*. Prentice Hall, 2003.

[5] J. Bergin. Teaching polymorphism with design patterns. In *Companion of the 18th annual ACM OOPSLA Conference, Anaheim, CA*, pp 167–169, Oct, 2003.

[6] J. Bloch. *Effective Java Programming Language Guide*. Addison Wesley, 2001.

[7] K. Bruce, A. Danyluk, and T. P. Murtagh. A library to support a graphics-based object-first approach to cs1. *SIGCSE Bulletin*, 33(1):6–10, 2001.

[8] K. B. Bruce, A. P. Danyluk, and T. P. Murtagh. *Java: An eventful approach*. Prentice-Hall, 2004.

[9] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM OOPSLA Conference, Anaheim, CA*, pp 148–155, Oct, 2003.

[10] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs*. MIT Press, 2001.

[11] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. The Teachscheme! Project: Computing and programming for every student. *Computer Science Education*, 14(1):55–77, 2004.

[12] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *J. of Functional Programming*, 12(2):159–182, 2002.

[13] K. E. Gray and M. Flatt. ProfessorJ: a gradual introduction to Java through language levels. In *Companion of the 18th annual ACM OOPSLA Conference, Anaheim, CA*, pp 170–177, Oct, 2003.

[14] J. I. Hsia, E. Simpson, D. Smith, and R. Cartwright. Taming Java for the classroom. *SIGCSE Bulletin and Proceedings*, 7(1):327–331, 2005.

[15] A. Kay. The early history of Smalltalk. *SIGPLAN Notices*, 28(3), March 1993.

[16] C. H. Nevison. From concrete to abstract: The power of generalization. In *Companion of the 19th annual ACM OOPSLA Conference, Vancouver, BC*, pp 106–108, Oct, 2004.

[17] V. K. Proulx and T. Cashorali. Calculator problem and the Design Recipe. *SIGPLAN Notices*, 40(3):4–11, 2005.

[18] D. H. Schunk and P. A. Ertmer. Self-regulatory processes during computer skill acquisition. *Journal of Educational Psychology*, 91:251–260, 1999.

[19] B. J. Zimmerman and D. H. S. (eds.). *Self-regulated learning and academic achievement: Theory, research and practice*. Springer Verlag, 1989.