Position Paper for OOPSLA 2001 Workshop
Pedagogies and Tools for Assimilating Object-Oriented Concepts

OOP as an Enrichment of FP

Robert "Corky" Cartwright and Dung "Zung" Nguyen
Department of Computer Science
Rice University
Houston, TX 77251-1892, USA
cork@rice.edu, dxnguyen@rice.edu

Abstract

At Rice University, we have a long history of teaching *functional* programming (FP) in Scheme in our first semester programming course. Five years ago, when we converted our second semester course from *object-based* programming (OBP) in C++ to *object-oriented* programming (OOP) in Java, we discovered that object-oriented program design and functional programming design are intimately connected. In both FP and OOP, the fundamental principle of program design is: the structure of a program should mirror the form of data that it processes. Moreover, for every functional program, there is a corresponding object-oriented program derived from the functional program using appropriate *design patterns*. The resulting OO program is still *functional* in the sense that all data objects are *immutable*.

In our second semester course, we leverage our students understanding of functional design to teach object-oriented design using *design patterns*. We show students that the *data-directed* approach to program design taught in the first course applies equally well to writing object-oriented programs. Design patterns provide standard templates for expressing common data definitions and operations that process them in object-oriented form. We begin by teaching the *union* (*composite* without recursion), *composite*, *interpreter*, *singleton*, *command*, *factory method*, and *visitor* patterns. Students learn that there is an OO analog for every abstraction in FP. In Java, the connection is particularly compelling because Java fully supports closures in the form of inner classes. Once students have mastered "functional programming" in Java, we introduce object mutation using the *state*, *strategy*, *iterator*, *observer*, and *model-view-controller* patterns. We also show students how OOP facilitates code factoring and re-use.

Position

Over the past decade, the Rice Computer Science Department has developed an innovative introductory computing curriculum [2,8,9] that emphasizes a high-level *algebraic* view of computation based on an extension of the familiar laws of arithmetic and algebra taught in secondary school. The curriculum consists of two semester-long courses: a first course focusing on *data-directed* functional programming in Scheme and a second course focusing on object-oriented design in Java.

The first course is more prescriptive than conventional introductory programming courses. Students are given an explicit design recipe for writing programs that stresses the correspondence between the definition of the data processed by a program and the structure of that program. This *data-directed* approach to program design provides students with a concrete methodology for designing programs--an issue that is generally ignored in traditional introductory courses. Students are given the following six-step *design recipe* for writing programs:

1. *Analyze the problem and define the data*. Determine what forms of data the program will process. Write precise definitions in English for each such form of data. Implement these definitions by data definitions in the target language.

2. *Determine the program operations required to solve the problem*. For each operation, write the signature (contract and header) and a purpose statement stating what the operation does.

3. For each program operation, *give a collection of examples* specifying how that operation should behave.

4. For each program operation, *generate a template corresponding to the definition of the data processed by the operation*. For each self-reference in the data definition, the template includes a "structural" recursive call.

5. For each operation, *write the code body* by filling in the selected template.

6. *Test* each operation using the examples from step 3.

The methodology is supported by a comprehensive pedagogic programming environment called DrScheme [3], which provides a hierarchy of language levels, from beginning to advanced, enabling the environment's syntax checking to recognize common syntactic mistakes in programs written by beginning students.

To provide students with a sound mathematical understanding of Scheme programs, the course includes a simple set of *laws* (rewrite rules) for evaluating Scheme programs akin to the laws of arithmetic. These laws are implemented as a stepper in DrScheme so students can trace the evaluation of computations step by step. The first course has recently been codified as a textbook entitled *How to Design Programs* [2].

The second course shows students how to apply the design principles from the first course to the task of writing object-oriented programs. Object-oriented design is a simple extension of *data-directed* functional design wherein functional data definitions and templates are converted to object-oriented form. The resulting object-oriented class diagrams and templates correspond to well-known *design patterns* described in the seminal book on the subject written by Gamma, Helm, Johnson, and Vlissides [1].

The first phase of the course introduces the *composite*, *interpreter*, *singleton*, *factory method*, *command*, and *visitor* patterns to show students how to express simple functional designs in object-oriented form. In this phase of the course, the *only* control structure that we use other than dynamic dispatch is the **if** statement. We motivate the introduction of each design pattern by focusing on a computation that involves the abstraction captured in the pattern. For example, to motivate the introduction of the composite, interpreter, and singleton patterns, we consider simple functional computations on algebraic data types such as lists of integers.

An `IntList` has one of two forms: (*i*) `Empty` or (*ii*) `NonEmpty`. In the latter case, a `NonEmpty` list *has* two components: a `first` element which is an `int` and a rest component which is an `IntList`. Hence, the data type `IntList` is inductively defined. To express the inductive structure of this definition in Java, we use the *composite* pattern. In this pattern, `IntList` is an abstract class with two concrete subclasses: `Empty` and `NonEmpty`. The following table shows the code skeletons embodying this data definition in Java and Scheme:

Object-Oriented Design in Java	Data-directed Design in Scheme

`abstract class IntList {`
`  abstract int first();`
`  abstract IntList rest();`
`}`
	`;; IntList ::= Empty +`
	`;;  NonEmpty(integer, IntList)`

`class Empty extends IntList {`
`  int first() {`
`  throw new IllegalArgumentException("first applied to Empty");`
`  }`
`  IntList rest() {`
`  throw new IllegalArgumentException("rest applied to Empty");`
`  }`
`}`
	`(define-struct Empty ())`

`class NonEmpty extends IntList {`
`  int first;`
`  IntList rest;`
`  int first() { return first; }`
`  int rest() { return rest; }`
`  NonEmpty(int f, IntList r) {`
`  first = f; rest = r;`
`  }`
`}`
	`(define-struct NonEmpty (first rest)`
	`;; Scheme automatically generates`
	`;; NonEmpty-first and`
	`;; NonEmpty-rest`

Assume that our task is to define a method `sum` for `IntList` that returns the sum of the elements in `this`. The standard template for defining methods on a *composite* class hierarchy is the *interpreter* pattern. In this pattern, the defined method is left abstract in `IntList` and defined in the concrete subclasses. Moreover, the new method code performs recursive calls to process any fields of type `IntList`. The following table shows the skeletons for the definition of `sum` in both Java and Scheme.

Object-Oriented Design in Java	Data-directed Design in Scheme

`abstract class IntList {`
`  ...`
`  abstract int sum();`
`}`
	`;; IntList ::= Empty +`
	`;;  NonEmpty(integer, IntList)`
	`;; sum: IntList -> integer`

`class Empty extends IntList {`
`  ...`
`  int sum() { return ...; }`
`}`
	`(define-struct Empty ())`

`class NonEmpty extends IntList {`
`  ...`
`  int sum() { return ... rest.sum() ...; }`
`}`
	`(define-struct NonEmpty (first rest)`

	`(define (sum ilist)`
	`  (cond [(Empty? ilist) ...]`
	`  [(NonEmpty? ilist)`
	`  ... (sum (NonEmpty-rest ilist)) ...])) `

The decomposition of the definition of `sum` forced by the *composite* and *interpreter* patterns *exactly* corresponds to the template for writing a function `sum` to process lists in Scheme. The only aspect of writing the definition of `sum` in Java (or Scheme) that is not purely mechanical is filling in the dots in the two concrete clauses of the definition:.

Object-Oriented Design in Java	Data-directed Design in Scheme

`abstract class IntList {`
`  ...`
`  abstract int sum();`
`}`
	`;; IntList ::= Empty +`
	`  NonEmpty(integer, IntList)`
	`;; sum: IntList -> integer`

`class Empty extends IntList {`
`  ...`
`  int sum() { return 0; }`
`}`
	`(define-struct Empty ())`

`class NonEmpty extends IntList {`
`  ...`
`  int sum() { return first + rest.sum(); }`
`}`
	`(define-struct NonEmpty (first rest)`

	`(define (sum ilist)`
	`  (cond [(Empty? ilist) 0]`
	`  [(NonEmpty? ilist)`
	`  (+ (NonEmpty-first ilist)`
	`  (sum (NonEmpty-rest ilist))]))`

We motivate the introduction of the *singleton* pattern by the fact that some classes such as `Empty` above only contain one object. To support higher-order "functional" programming in Java, we introduce the *command* pattern and anonymous inner classes to represent functions as data objects. To enable the definition of new operations that process a composite class hierarchy *without modifying the classes in the hierarchy*, we introduce the *visitor* pattern. A *factory method* in the visitor interface for constructing new visitors gracefully supports the extension of visitor classes if additional concrete variant classes are ever added to the composite hierarchy.

The second phase of the course introduces the concept of mutable state and the accompanying design patterns including *state*, *strategy*, *iterator*, *observer*, and *model-view-controller*. These patterns control and encapsulate the use of mutable state. We motivate the introduction of state by introducing memo-functions that avoid re-computing solutions to problems that have been solved previously. We do not introduce the concept of loops (which are *procedural* control structures) until we discuss the *iterator* pattern.

The course has been codified in an online monograph entitled "Elements of Object-Oriented Program Design" at www.teachjava.org. We plan to expand these notes into a textbook. Their content is almost completely disjoint from extant Java textbooks.

The second course is supported by a pedagogic programming environment called DrJava that includes a general "read-eval-print" loop. DrJava supports exactly the same programming interface as DrScheme[3]: a Definitions window for writing program text and an Interactions window for evaluating program expressions based on the definitions in the Definitions window. The Interactions window is a "read-eval-print" loop that is reset every time the contents of Definitions window are recompiled.

The "read-eval-print" loop in DrJava completely eliminates the need for the static `main` method used to run Java programs from a command line interface. In fact, it enables us to postpone introducing static methods until students are very comfortable writing instance methods and thoroughly understand the use of `this`.

The Definitions windows provides an "intelligent" editor that understands strings, comments, and the nesting levels in Java syntax. It highlights strings, comments, keywords, and matching nesting brackets and maintains an accurate highlighting of program text on every keystroke.

DrJava is an ordinary Java jar file and is available for downloading at www.teachjava.org. The Java compiler used in DrJava is the GJ (Generic Java) compiler written by Martin Odersky. As a result, DrJava supports the use of generics in the Definitions window.

Pedagogic Rationale

The core concepts of OOP can be intelligibly explained to beginners if we restrict the domain of computation to *immutable* data. The immutability of objects enables

i. simple algebraic reasoning about program behavior,
ii. the definition of most program operations by structural recursion,
iii. and the formulation of a simple operational semantics for Java (or other OO language) based on textual rewriting, akin to the semantics for Scheme presented in HTDP and implemented in the DrScheme stepper.

The algebraic structure of functional computation is leveraged in programs that can be conveniently written using structural recursion. Programs are constructed (and proved correct) using simple inductive reasoning. To express such simple computations in an object-oriented language, inductive data definitions must be represented as composite class hierarchies. Operations defined by structural recursion are expressed as method definitions based on the interpreter pattern over a composite hierarchy or as visitor objects that encapsulate the code for a new operation without disturbing the class definitions in the composite hierarchy.

This framework is accessible to beginning programmers yet it makes essential use of the core concepts of object-oriented programming: encapsulation, inheritance, and polymorphism.

Future Directions

We are contemplating the development of a new curriculum that is entirely Java-based. Despite the merits of our existing curriculum model, it is hard sell at most universities because functional languages like Scheme are dismissed as outside the mainstream of computing. Fortunately, we believe that the principles of program design embodied in our curriculum can be taught directly in Java--provided that the syntactic complexity of Java is tamed.

For this reason, we plan to add support for a hierarchy of progressively more complex language levels in DrJava where the two simplest levels are confined to functional computation (immutable data). The beginning language level will automatically generate the constructors, getters, the `toString` method, and the `equals` method for a class--just as the `define-struct` construct in Scheme automatically generates these operations. With such support, Java is comparable in syntactic complexity to Scheme properly annotated with data definitions and operation types. The table below juxtaposes the sum programs in beginning level Java and properly annotated Scheme. We also plan to develop an operational semantics for functional Java based on textual rewriting and implement that semantics in a stepper for DrJava.

Object-Oriented Design in Java	Data-directed Design in Scheme

`abstract class IntList {`
`  abstract int sum();`
`}`
	`;; IntList ::= Empty +`
	`;;  NonEmpty(integer, IntList)`
	`;;  sum: IntList -> integer`

`class Empty extends IntList {`
`  int sum() { return 0; }`
`}`
	`(define-struct Empty ())`

`class NonEmpty extends IntList {`
`  int first;`
`  IntList rest;`
`  int sum() { return first + rest.sum(); }`
`}`
	`(define-struct NonEmpty (first rest))`

	`(define (sum ilist)`
	`  (cond [(Empty? ilist) 0]`
	`  [(NonEmpty? ilist)`
	`  (+ (NonEmpty-first ilist)`
	`  (sum (NonEmpty-rest ilist)))]))`

References

1. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass. 1995.
2. M. Felleisen, R. Findler, M. Flatt, S. Krisnamurthi. *How to Design Programs.* MIT Press, 2001.
3. R. Findler, C. Flanagan, M. Flatt, Shriram K., and M. Felleisen. DrScheme: a pedagogic programming environment for Scheme. In *Proc. 1997 Symposium on Programming Languages: Implementations and Logics*.
4. D. Nguyen and S. Wong, "Design Patterns for Sorting," SIGCSE Bulletin 33:1, March 2001, 263-267.
5. D. Nguyen and S. Wong, "Design Patterns for Lazy Evaluation," SIGCSE Bulletin 32:1, March 2000, 21-25.
6. D. Nguyen and S. Wong, "Design Patterns for Decoupling Data Structures and Algorithms," SIGCSE Bulletin 31:1, March 1999, 87-91.
7. D. Nguyen and S. Wong, "Design Patterns for Sorting," SIGCSE Bulletin 30:1, March 1998, 336-340.
8. [TeachJava] Rice Computer Science Department. The TeachJava Project. http://www.teachjava.org.
9. [TeachScheme] Rice Computer Science Department. The TeachScheme Project. http://www.teach-scheme.org

Biographies:

Robert "Corky" Cartwright: Corky is a Professor of Computer Science at Rice University. He received an A.B. degree in Applied Mathematics from Harvard College in 1971 and a Ph.D. in Computer Science from Stanford University in 1977. He worked as an Assistant Professor of Computer Science at Cornell University from 1976 to 1980 before joining the Rice University faculty in 1980. He has published numerous research papers in the area of programming languages and jointly developed Rice's introductory programming curriculum with Matthias Felleisen. Last year, he served as a member of the College AP Computer Science Ad Hoc Committee that recommended converting the AP syllabus from object-based programming in C++ to object-oriented programming in Java. He is an ACM Fellow and chair of the ACM Education Board Committee on Pre-college Education.

Dung "Zung" Nguyen: Zung received his Ph.D. in Mathematics from the UC Berkeley in 1981. He trained in Computer Science at the Institute For Retraining In Computer Science (IFRICS) at Kent State in 1985 and 1986 and subsequently took Computer Sciences courses at Stanford University, UC Berkeley, and UCLA on computer graphics, compiler construction, program interoperability, and network programming. He has participated in NSF sponsored workshops on object-oriented programming (Illinois State University), artificial intelligence (Temple University), and computer networks (University of Dayton). Before joining Rice, he was a visiting professor at Pepperdine University where he and Stan Warford created a new Computer Science/Mathematics curriculum with a strong emphasis on OOP (www.pepperdine.edu/seaver/natsci/COMP%20SCI/compsci.htm).