# Functional Programming Patterns and Their Role in Instruction

Eugene Wallingford

Department of Computer Science, University of Northern Iowa, Cedar Falls, Iowa 50614
wallingf@cs.uni.edu

**Abstract**

Functional programming is a powerful style in which to write programs, but students and faculty alike often have a hard time learning to exploit its power. A pattern language of functional programs can provide programmers with concrete guidance for writing functional programs and also offer a deeper appreciation of functional style. Used effectively, such pattern languages may help functional programming educators reach a broader audience.

## 1 Introduction

Functional programming is a powerful style for writing programs, yet many students never fully appreciate it. After programming in an imperative style, where state and state changes are central, the idioms of the functional style can feel uncomfortable. Further, many functional programming ideas involve abstractions beyond what other styles allow, and often students do not fully understand the reasons for using them.

Complicating matters, many university faculty do not have strong backgrounds in functional programming. These faculty may have studied a single functional language only briefly as undergraduates. Without working in the style over time on large programs, they never develop a sense of what functional programming is like. Such faculty would like to convey the beauty of functional programming to their students but often find themselves teaching the surface features of a language.

Surface features, though, are not enough for students to learn to read, let alone write, programs. Equipped only with the vocabulary of a programming language and basic concepts of a style, novices have difficulty recognizing higher-level structures in the programs they read. They also tend to work at a low level of abstraction when writing code. What an expert programmer easily sees can seem incomprehensible to them. An instructor's goal is to help novices develop their own expertise.

### 1.1 Patterns as an Approach

Over the last decade, many software professionals have begun to explore the use of patterns as a way to record expert knowledge for building software. In its simplest form, a *pattern* is a three-part rule that expresses a relation among

- a programming context,
- the set of competing concerns, or *forces*, that occurs repeatedly in that context,
- and a stereotypical software configuration that resolves these forces favorably.

The context denotes the state of a system: the components already present and global constraints on the result. The set of forces constitutes a problem to be solved, and the stereotypical configuration solves the problem. In practice, a pattern usually also explains why this solution suffices and describes how to implement the solution in code.

The value of a pattern lies largely in its ability to explain. Writing a pattern requires the author to state explicitly the context in which a technique applies and the design issues involved in implementing a solution. These elements of a pattern provide significant benefits to the reader, who can both learn the technique and explore when and how to use it. The author of a pattern also benefits, as the pattern form encourages a level of explicitness not always present in other written forms.

A *pattern language* is a collection of patterns that together create a complete system. In a pattern language, the context of each pattern refers to other patterns in the language. As such, the language guides the user through the process of building a solution to a more complex problem. It specifies a partial ordering on the use of the patterns, which helps novice programmers as they begin to implement more complex programs. But, because each pattern also enumerates its context and forces, the user can learn when and how to make exceptions. Likewise, a pattern language can also help its user to understand existing programs by describing the higher-level structures they typically contain.

Software patterns began as an industry phenomenon, an attempt to document bits of working knowledge that go beyond what developers learned in their academic study. They have become a standard educational device both in industry and in object-oriented (OO) design and programming instruction at universities. Given their utility in industry and in OO instruction, patterns offer a promising approach to help students and faculty learn to write functional programs. Such patterns will document the common techniques and program structures used by functional programmers, and pattern languages will document the process of using functional programming patterns in the construction of larger programs—ideally, complete programs that solve problems of real interest.

### 1.2 About This Paper

This paper recounts some of my work documenting patterns of functional programs and using pattern languages in my teaching. The next section describes one such pattern language, Roundabout, which documents patterns of recursive programs. Section 3 presents several ways I use Roundabout in my course and offers qualitative evaluation of the results. Section 4 discusses related work, and Section 5 suggests future work.

The patterns described here reflect a Scheme programming style. The details of some of these patterns are language-specific, but the idea of functional programming patterns and their use in teaching extends beyond any particular language.

## 2 Functional Programming Patterns

I have written two small pattern languages for use in my courses. Here, I consider Roundabout, an incomplete pattern language for recursive programs over inductively-defined types [9]. It comprises seven patterns often present in such recursive programs. (See Figure 1.)

Structural Recursion is the entry point to the language, specifying that the shape of a procedure should match the shape of the inductive type definition that it processes. Patterns in the second layer expand on the first and document primary techniques used to structure recursive programs, in particular the creation of particular kinds of helper procedures. Finally, patterns in the third layer describe how to improve the programs generated by the first five in the face of other forces such as run-time efficiency or name-space clutter.

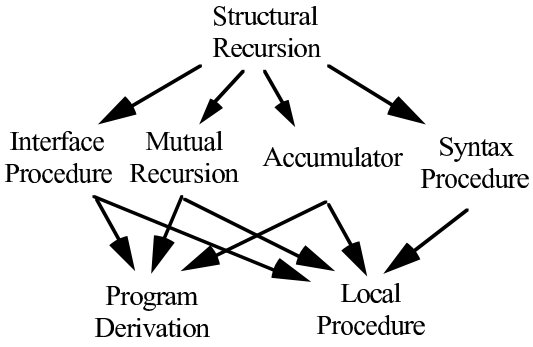


**Fig. 1.** The patterns of Roundabout

The patterns of Roundabout collaborate to create complete solutions to problems involving inductively-defined types. One way they do this is through the contexts of the patterns, which record the relationships between patterns explicitly. Each link in Figure 1 indicates that the first pattern appears in the context of the second. This means that, after applying the first pattern, you may want to apply the second pattern to address some forces in the solution unresolved by the first.

### 2.1 The Anatomy of a Pattern

Each pattern in Roundabout consists of six parts. This section considers each, using the Accumulator pattern as an example.

**Context**: a statement of the patterns and global factors that must be present before this pattern is relevant. This is a precondition on the pattern's applicability.

• Accumulator presumes that the program uses Structural Recursion.

**Problem**: a concise statement of the specific issue addressed by this pattern. The problem solved by a pattern embodies its intent.

• Accumulator addresses a problem that sometimes occurs when multiple procedures collaborate: the natural flow of a recursive program can lead to unnecessarily inefficient code. Use of Mutual Recursion can cause this problem, so Accumulator's context also mentions Mutual Recursion specifically.

**Forces**: a discussion of the competing constraints that face a programmer trying to solve the problem. This discussion motivates the pattern's use by showing how other solutions do not adequately solve the problem. Often, this section introduces a sample problem to make the discussion of the pattern more concrete.

• The Accumulator pattern uses the example of a procedure that flattens nested lists. A straightforward use of Mutual Recursion results in a solution that is $O(n^2)$ in the size of the data. The naturalness and readability of the two-procedure solution conflicts with the inefficiency of the resulting code.

**Solution**: a description of the pattern's structure, and how to implement it.

• Accumulator resolves the competing forces by introducing an accumulator variable to hold the result of the computation as it is built. Each procedure receives the accumulator as an argument, so each can add to the solution and control the flow of the computation independently. Implementing an accumulator requires that the top-level procedure become an Interface Procedure that passes the initial value of the accumulator to a helper procedure that does the computation.

**Problem Resolved**: a demonstration of how the pattern's structure solves the problem and addresses the forces. This section may explicitly discuss the balancing of the forces, or it may only show the pattern implemented for the sample problem.

• Use of Accumulator solves the flattening problem in a way that uses mutual recursion and computes its answer in O(n) time.

**Resulting Context**: a statement of the patterns that may be used to address any constraints that affect the resulting program structure. These constraints may be forces left unresolved by this pattern or new forces introduced by implementing the structure.

• An Accumulator often relies on mutually-recursive procedures, so the program may be inefficient. Use of the Program Derivation pattern may retain most of the benefits of the design while achieving better performance. Extra procedures may clutter the name space of the program, which can be resolved in part by use of the Local Procedure pattern.

As a pattern writer, I must pay careful attention to why I use a given structure in a particular program, why I choose to implement it at that point in the process, and why it is better than some other in that situation. These answers help my students learn the vocabulary of the style and to think critically about the writing of programs.

### 2.2 Writing a Program with Roundabout

Roundabout leads to the creation of a complete program. Sometimes, the resulting solution will differ from any program a student would have built on his or her own.

Consider the task of writing the procedure to `replace` all occurrences of one symbol in an s-list with another. Because the s-list is defined inductively, the *Structural Recursion* pattern applies. This pattern says to build a procedure with one arm for each type of s-list. The case for handling a non-empty s-list involves two choices of its own: Is the first item in the list a symbol? And, if so, is it the symbol to be replaced? This requires a nested `if` expression of its own, and each arm will repeat code found in the others.

But the s-list data type consists of two mutually inductive data definitions, one for s-lists and one for symbol expressions. The resulting context of the Structural Recursion

pattern tells us that such data types should be processed further using the more specialized *Mutual Recursion* pattern. This pattern applies Structural Recursion to both data definitions, with the resulting procedures making calls to one another in the appropriate locations. The resulting procedures are truer to the data definition than a one-procedure solution might be, and they do not repeat code in the way a naive application of Structural Recursion would have.

The use of mutually recursive procedures interjects new forces into the problem. For inputs with deeply nested lists, many of the mutually recursive calls will result in immediate calls back to the calling procedure. If this sort of inefficiency is unacceptable, then Roundabout suggests the *Program Derivation* pattern as a potential technique for resolving the forces. Program derivation is a technique for using functional programming's substitution model to generate code. Applied to the two `replace` procedures produced by Mutual Recursion, this technique gives a one-procedure solution (Figure 2).

```
(define replace
   (lambda (new old slist)
      (if (null? slist)
          '()
          (cons (if (symbol? (car slist))
                    (if (eq? (car slist) old)
                        new
                        (car slist))
                    (replace new old (car slist)))
                (replace new old (cdr slist))))))
```

**Fig 2.** The final version of the `replace` procedure

This result can be quite surprising to novices. The final solution consists of a single procedure, which is often their goal, yet it is unlike any procedure that they would have written from scratch themselves. They see how to use Scheme's tools to create a solution with little or no repeated code, much as an expert programmer might. Students are gratified that they can approach such expert results by using an almost mechanical technique, which further motivates their use of the pattern language.

The pattern language form makes explicit the relationships among the structures that compose a whole. These explicit connections guide students in the process of developing a program while leaving room for them to make trade-offs in the application of the patterns. Through experience, the programmer will internalize the pattern language and begin to develop instincts for the programming style.

## 3 Teaching with Patterns

I wrote the pattern language to help me teach the functional programming segment of my Programming Languages and Paradigms course. It now serves as the basis for a three-week unit on recursive programming. Roundabout affects my teaching in several ways.

*Lecture organization.* The pattern form presents ideas in a way suitable for students. A pattern begins with a problem, explores the forces that drive the solution, and then describes a solution that balances the forces. The discussion of the forces can explore other candidate solutions and why they are not good enough. Presentation of the

solution can explain why the solution works and what new forces it introduces. This form helps students discern when to use a technique and when to look for a different one.

Another approach shows students a "before and after" picture: give them a program that solves a problem using the techniques they know at that point, and then give a refactored solution that employs a new pattern. This technique works better for larger problems, as that yields a starker contrast between solutions. This approach motivates students by helping them see the kind of difference a pattern can make.

The pattern language form can help the instructor design several sessions that deal with the patterns working together. The pattern contexts give a natural partial ordering of the presentation of topics, and the resulting contexts provide for transitions between patterns.

*Assignment creation*. Using Roundabout affects how I select homework and exam problems by encouraging the use of problems that (1) explore each pattern, (2) cause the student to choose among techniques based on the forces in the problem, and (3) cause the student to apply several patterns to create a single solution. While these seem to be natural goals for good assignments, and textbook authors often cover all three, I find that the pattern language helps me to design assignments with more complete coverage.

*Design evaluation*. A pattern language can serve as a useful tool in the task of evaluating student designs, for the purposes of giving assistance, feedback, and grades [10]. The patterns in the language constitute a vocabulary for describing designs, and the language gives rules for generating good designs. And, most important from the perspective of evaluation, the pattern contexts give an explicit standard with which to evaluate designs. By specifying more completely when each pattern applies and what forces it addresses, the pattern language guides students in applying the techniques they learn, and it gives instructors a concrete reference for critiquing student work.

### 3.1 Student Perceptions

I can offer some qualitative evaluation of the effects that teaching with this pattern language has had on my students' performance and perceptions. I have taught our programming languages course five times using Roundabout. During the last session of each semester, students evaluate the course in small groups, and each small group then reports its results to the class as a whole. These public evaluations typically list 'recursive programming' as one of the three most valuable items learned during the semester. This seems significant because by this time students have studied and used recursion in at least two prior courses. Their new confidence could result simply from growing experience, but students report that Roundabout has helped them learn how to use recursion more comfortably.

Students also complete an anonymous course evaluation, on which they may comment on any part of the course. Most students do not comment but, when they do, approximately one-fourth mention Roundabout, always favorably.

Student scores on recursive programming exercises have improved by an average of 7.5% since I began to use this pattern language in class. This may only be a result of the fact that I now do a better job teaching recursive programming than I did before, but that is

one of the central reasons to use the pattern form: it encouraged and sometimes forced me to understand and explain why and when to use the techniques that I was teaching.

This improvement occurs whether I discuss the patterns explicitly in class or use them only as a way to structure the content of the course. The one semester that scores did not improve was last spring, when I stated directly that use of the patterns was "optional": that students were responsible only for their solutions and not for their technique. I found that many students chose not to use them, instead relying on their past experience with recursion. On average, student scores on recursive exercises fell back near previous levels. Students must be encouraged to try the techniques in the first place.

However, this encouragement must focus on the content of the patterns and not the patterns *qua* patterns. One possible drawback of using a pattern language to teach is that instructors or students focus on pattern names and "learning the patterns", losing sight of the fact that patterns simply document good style and technique for solving a class of problems. Learning how to build good solutions is the goal, not memorizing patterns.

## 4 Related Work

Several groups are working to improve how we teach functional programming. At least two are in a similar spirit as my work with patterns. One approach, exemplified in the introductory text *How to Design Programs* [4], focuses on the design process that programmers use to *create* programs. Another approach focuses on how programmers *evolve* their programs through refactoring [7]. Pattern languages complement these approaches by defining both a vocabulary and a process. Individual patterns can serve as components in a program being built using another design process. The pattern language can be used to augment the design process, either by giving specific examples of design steps or by filling in gaps in the process. Further, it can also be used in deciding when and how to refactor, because each pattern specifies when it applies and how it resolves the forces at play. Refactoring often involves replacing one pattern with another from the same pattern language that generated the program.

Relatively few papers in the patterns literature document patterns of functional programming. In addition to Roundabout, I have written Envoy, a pattern language for programs that maintain state [8]. Ferguson and Deugo discuss standard ways to build complex control structures using `call-with-current-continuation` in Scheme [5]. Kuehne [6] shows how common functional programming patterns can generate desirable OO designs.

A small group of computer science educators have documented *elementary patterns* for teaching object-oriented and procedural programming. These patterns capture low-level knowledge that underlies more advanced patterns, the sort of knowledge that every expert programmer has and uses subconsciously. This work includes patterns of selection [2], loops [1], and substitutability and delegation [3].

## 5 Conclusion

This paper describes my work documenting patterns of functional programs and using pattern languages to teach. In this approach, patterns serve as a vocabulary of design

elements; the pattern language documents relationships among patterns and a process for generating elegant programs that meet functional goals. The pattern form encourages the author to explain carefully when each design element works best and why. Such explanations help students better understand how to use the techniques they learn. These benefits also accrue to instructors, perhaps especially to those who write patterns.

To develop this approach further, programmers and educators must write more and better pattern languages of functional programs. The existing literature fills only a few small niches. The patterns yet to describe vary widely, from the uses of techniques such as higher-order procedures and currying to more advanced topics such as monads. Another fruitful avenue is to study the targets of typical refactorings of functional programs.

More work also needs to be done to explore ways to use patterns effectively in teaching. The value of such work can hardly be overestimated. Though faculty generally design a curriculum with their students in mind, often it most benefits other faculty who will teach it. Teachers need to learn both the content of the approach as well as how to present it effectively. The largest first-order effect of patterns on OO instruction has been on instructors, and this would likely hold for functional programming instruction, too.

The value in writing pattern languages of functional programs extends beyond its potential effects on university teaching. Pattern languages share expert knowledge with a larger audience. Patterns have played a central role in the transfer of OO technology within industry. Pattern languages of functional programs may help to reveal the beauty and power of functional programming to a wider audience of programmers.

## References


[1] Owen Astrachan and Eugene Wallingford, *Loop Patterns*, Proc. Fifth Pattern Languages of Programs Conference, Allerton Park, Illinois. 1998.

[2] Joe Bergin, *Selection Patterns*, Proc. Fourth European Pattern Languages of Programs Conference, Bad Irsee, Germany, 1999.

[3] Dwight Deugo, *Foundation Patterns*, Proc. Fifth Pattern Languages of Programs Conference, Allerton Park, Illinois. 1998.

[4] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi, *How to Design Programs: An Introduction to Computing and Programming*, MIT Press, Cambridge, Massachusetts, 2001.

[5] Darrell Ferguson and Dwight Deugo, *Call with Current Continuation Patterns*, Proc. Eighth Pattern Languages of Programs Conference, Allerton Park, Illinois, 2001.

[6] Thomas Kuehne, *A Functional Pattern System for Object-Oriented Design*, Verlag Dr. Kovac, Hamburg, Germany, 1999.

[7] Simon Thompson and Claus Reinke, *Refactoring Functional Programs*, Technical Report 16-01, Computing Laboratory, University of Kent at Canterbury, October 2001.

[8] Eugene Wallingford, *Envoy: A Pattern Language for Maintaining State*, Proc. Sixth Pattern Languages of Programs Conference, Allerton Park, Illinois, 1999.

[9] Eugene Wallingford, *Roundabout: A Pattern Language for Recursive Programming*. Proc. Fourth Pattern Languages of Programs Conference, Allerton Park, Illinois, 1997.

[10] Eugene Wallingford, *Using a Pattern Language to Evaluate Design*, OOPSLA Workshop on Evaluating Object-Oriented Design, Vancouver, British Columbia, 1998.


This paper and links to references and supporting material can be found at:
`http://www.cs.uni.edu/~wallingf/patterns/functional/`