Multiparadigm Patterns of Thought and Design

Charles D. Knutson
Timothy A. Budd
Curtis R. Cook
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331
{knutson, budd, cook}@research.cs.orst.edu

1. Introduction

Since its inception, the field of software design patterns has emphasized an object-oriented perspective. Yet object-oriented programming is just one of many programming paradigms. When we expand our view of programming to become more inclusive of other paradigms, we are presented with a much wider range of possible design choices. In this paper, we illustrate this, by showing that even such a simple process as iteration can be approached from a number of different perspectives. Alexander [1] sought patterns that succeeded in stating true invariants, summarizing "a *property* common to *all possible ways* of solving the stated problem." If design patterns are to be considered truly universal, the pattern community should be broadened to consider other programming paradigms.

1.1 Paradigmatic Drift

The linguist Edward Sapir [7] wrote that natural language "moves down time in a current of its own making. It has a drift." The same can be said of programming languages. Imperative languages like C get new clothes and a makeover and emerge as object-oriented languages like C++ or Objective-C. The computing world becomes more visual, so elements of visual programming are incorporated, and we get Visual C++[1]. These new languages gain acceptance in part because they incorporate the currently popular ways of programming and problem solving, but also because they bring along the old baggage with which we feel comfortable. We believe that some of the next important evolutionary changes in the way we think about programming will come from viewing problems through the perspective of other programming paradigms. As new elements are merged with existing languages, programming languages will continue to be increasingly multiparadigmatic in their nature.

Alexander [2] states that "every society which is alive and whole, will have its own unique and distinct pattern language." If there is an object-oriented pattern language, then imperative, logical, functional, and visual pattern languages should also exist. And if elements of these languages can be used effectively in combination, there should be multiparadigm pattern languages. These multiparadigm patterns should at least impact the way we think about

---

[1] Paradigmatic purists would argue that C++ is not truly object-oriented, and Visual C++ is not truly visual programming. But when language elements from disparate paradigms come together and share their strengths, some of the purity is often lost in the marriage.

problems, and at best be facilitated by multiparadigm programming languages that can be used to solve them.

2. Multiparadigm Programming and Problem Solving

Four paradigms have dominated programming since its inception: imperative, object-oriented, functional, and logical. Each of these paradigms encapsulates a particular view of programming and problem solving, and colors the world of the programmer who sees through the eyes of the paradigm. As Sapir [7] pointed out, "Languages differ not only in how they build their sentences but also in how they break down nature to secure the elements to put in those sentences." Not only are the syntactic things we are able to do impacted by the languages in which we program, but the actual task of breaking down a problem into a solution is strongly influenced by the choice of paradigm (or even the particular language within a given programming paradigm). This implies that there may be elegant solutions to problems hidden from our view because the paradigm in which we think does not accommodate that view. By examining these multiparadigm patterns, we believe that unique design aspects will be more readily visible, and that these patterns will lend themselves to more elegant design, even in situations in which the language in which the solution is coded is not (strictly speaking) a multiparadigm language.

To lay some foundation, we will discuss these four programming paradigms, and summarize their views of programming and problem solving.

2.1 Imperative

The imperative programming paradigm is the one that most clearly emulates the Von Neumann architecture, with sequenced execution of instructions, and changes to values stored in memory locations. It is the Betty Crocker approach to programming, with "recipe cards" that take the computer through the necessary steps of combining the right things in the right way in the right bowls. Not surprisingly, it is the paradigm that has dominated programming through most of the history of computing. Imperative design typically involves some form of stepwise refinement, by which a problem is broken down into the necessary steps required to solve it. Data structures are designed, and procedures and functions operate on them. The most popular imperative languages have been COBOL, FORTRAN, BASIC, Pascal, and C.

2.2 Object-oriented

The object-oriented programming paradigm views systems as groups of interacting objects. Objects communicate by sending and receiving messages, which are requests for action bundled with whatever arguments may be necessary to complete the task. Each object has its own memory, which may consist of other objects. This view of a system as being composed of objects forces a different kind of problem decomposition than the imperative approach. Rather than refining the problem into smaller steps, we think of refining the system into smaller subsystems or objects. In addition, the definitions of objects are stored in classes, which have behaviors that can be shared through inheritance and composition. Object-oriented design is often centered around the design of class hierarchies (where object classes are declared) and instance hierarchies (where objects are instantiated). This approach largely subsumes the imperative paradigm, since the methods contained in classes typically operate in an imperative fashion on data encapsulated within the class. The most popular object-oriented languages have been Smalltalk, C++, and (most recently) Java.

2.3 Functional

The functional programming paradigm is built on the notion of functional composition. This is a view of computation in which the ultimate solution is seen as the composition of all the functions that have operated upon some value. For example, in an imperative view, if `x` is a variable with the value `5`, and it needs two things done to it, we would probably view it in the following way: `x := 5; x := f(x); x := g(x);` At this point `x` holds our solution. But in a functional approach, we might represent the solution as: `g(f(x<-5))`, where `x` becomes a handle for the value `5`, and the `f()` and `g()` are applied to it in turn. Functional programming eschews the side-effects inherent in imperative programming. Values, once created, are not variables, but entities that cannot be changed (there is no concept of changing values at memory locations). Reuse is facilitated through the creation of functions which can, in turn, be applied to other functions. In this way, entire systems can be built up through functional composition. These functions are "first-class" data values, meaning that functions can be assigned to identifiers, passed as arguments, or returned as the result of executing other functions. The most popular functional languages have been LISP, APL, and ML.

2.4 Logical

The logical programming paradigm is based upon propositional calculus which was originally used for formalized theorem proving. In this system, there are three key parts: a set of axioms (or facts), a set of rules of inference, and a question or query. This approach is declarative, or nonprocedural. That is, the programmer provides a series of assertions of facts, a collection of rules of inference, and a query. The programmer need not specify how the query is to be answered using the information provided. Instead, an underlying searching mechanism works behind the scene to determine whether the facts can be deduced from the given information. The search mechanism typically uses a depth-first search strategy with backtracking to search the problem space for the answer to a query. The first and most popular logical language is PROLOG.

2.5 Multiparadigm

In a loose sense, any combination of programming paradigms can be viewed as "multiparadigm." For example, logic programming with "cuts" to control infinite searching can be viewed as multiparadigm, combining declarative logic with control flow constraints. Similarly, functional programming with monads can be viewed as multiparadigm, combining functional programming with input and output, the nature of which requires side effects.

The most common implementations of multiparadigm programming involve the linking together of separately compilable object modules, which interact through some common procedure call mechanism. This provides some of the benefit of multiparadigm programming, but does not exploit the advantages of hybrid solutions, where the paradigmatic elements are tightly coupled. Languages like Leda [3] attempt to bring these elements together in a single language, providing a synthesis of programming concepts. Interesting design patterns can be found in the hybrid solutions that are made possible by multiparadigm languages.

3. Patterns of Thought and Design

If it is true, as Sapir [7] said, that "language and our thought-grooves are inextricably interrelated, are, in a sense, one and the same," then to limit design to one of these four perspectives is to limit our ability to solve some problems effectively. Or put even more succinctly by Whorf [8], "The forms of a person's thoughts are controlled by inexorable laws of pattern of which he is unconscious. These patterns are the unperceived intricate systematizations of his own language....And every language is a vast pattern-system, different from others, in which are culturally ordained the forms and categories by which the personality not only communicates, but also analyzes nature, notices or neglects types of relationship and phenomena, channels his reasoning, and builds the house of his consciousness."

We should be aware that the language and paradigm in which we have been trained to think leaves an indelible mark on the patterns we see, and the solutions we design. It is important to be able to see some of the patterns that are suggested by other paradigms, to understand that they exist, and to analyze the ways in which they can impact our design.

4. Multiparadigm Views of the Iterator Pattern

In Alexander's original work [1], he identifies three different kinds of patterns within his volume. Patterns identified with two asterisks are those that the authors believe to have stated a true invariant, "that the solution we have stated summarizes a *property* common to *all possible ways* of solving the stated problem," and "that it is not possible to solve the stated problem properly, without shaping the environment in one way or another according to the pattern that we have given--and that, in these cases, the pattern describes a deep and inescapable property of a well-formed environment." Patterns with one asterisk are those believed to "have made some progress towards identifying such an invariant." Those with no asterisks "have not succeeded in defining a true invariant," meaning that their pattern is *one* way of doing it, but does not capture the essence of the problem.

When multiple paradigms provide multiple views of even simple problems like iteration, the task of identifying the "true invariants" seems much more daunting than when viewed from a single perspective. As an example, we have chosen to examine iteration from the perspectives of paradigms other than object-oriented. Gamma et al. [5] include an excellent object-oriented Iterator Pattern, which has as its stated purpose to "provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." Additionally, they state as motivations that "you might want to traverse the list in different ways, depending on what you want to accomplish," and that "you might also need to have more than one traversal pending on the same list." We agree with these motivations for iteration, and will refer to them as we discuss iteration from different programming perspectives. All coding examples are in the multiparadigm language Leda.

4.1 Direct Manipulation

The most obvious (if unenlightened) approach to the task of iteration is for the user of the list abstraction to cycle directly over the elements contained in the individual links. While we of the enlightened object-oriented community are religiously well-grounded in Parnas' principles of information hiding [6], this view of iteration has dominated programming for decades, and traps programmers within a perspective from which they can extricate themselves only at great mental and emotional cost.

The following code segment shows an imperative iteration over a list in Leda. Leda's overriding operative paradigm is object-oriented, so variables `aList` and `p` are of class type `List` and `Link`, respectively. These two classes are parametrically polymorphic, so we can indicate the type of data held in the lists dynamically by indicating `[integer]` after the class name.

```
var
   aList : List[integer];
   p : Link[integer];
begin
   .
   .
   .
   {print out the values maintained by a list}
   p := aList.data;
   while defined(p) do
   begin
      print(p.datum);
      print(" ");
      p := p.link;
   end;
end;
```

This solution, while workable and efficient, is unsatisfactory in several regards. The major objection is that this solution forces the user of the `List` data abstraction to have detailed knowledge of the way in which lists are represented. In particular, the user of a list must not only know that elements are maintained in links but must declare the seemingly extraneous variable `p`. Even worse, not only must the variable `p` be declared, but it must be declared as an instance of the class `Link`, a class that is otherwise hidden from the user of the list abstraction.

The imperative pattern of iteration can allow for traversal in different ways, and could be extended to allow for more than one traversal pending on a single list. However, it cannot deliver the main thing that Gamma's Iterator pattern requires, namely, "a way to access the elements of an aggregate object sequentially without exposing its underlying representation." However, we should recognize that this requirement itself betrays an awareness of Parnas' principles and of object-oriented design--further evidence that our paradigmatic exposure influences our design, and that we see patterns from within our paradigm.

4.2 Passing the Action as a Function

As an alternative to the direct manipulation of link values, the functional version of iteration relies on the ability to create functions as expressions. In particular, functions can be passed as arguments to procedures and return as results of execution. The use of functions in this fashion is an important aspect of functional programming.

The basic idea in the following example is to define in class `List` a method, which we will call `onEach`, that takes as argument a one-argument function. When the `onEach` method is invoked, the function supplied as argument is then executed repeatedly, once for each element contained in the list. Each time the function is executed, a different value of the list is passed as argument.

```
class List [T : equality];
var
   data : Link[T];
   .
   .
   .
   function onEach(fun : function(T));
   var
      p : Link[T];
   begin
      p := data;
      while defined(p) do
      begin
         fun(p.datum);
         p := p.next;
      end;
   end;
   .
   .
   .
end;
```

Since the `onEach` method is part of the `List` class, it is permitted to access directly the link fields in the list without violating Parnas' principles. One way this operation could be used to print out the values maintained by a list is for the user to define a special printing function:

```
function printValue(i : integer);
begin
   print(i);
   print(" ");
end;
```

The name of this function can then be passed as argument to the method `onEach`:

```
var
   aList : List[integer];
   .
   .
   .
   {print out values held in the list}
   aList.onEach(printValue)
end;
```

The printValue function would then be invoked once for each value in the list named `aList`. An even better solution eliminates the necessity of explicitly creating the function `printValue`. This approach uses the ability in Leda to create unnamed functions directly as expressions, for example, in an argument list. The following statement shows how the `onEach` method might be used in this fashion to print the values maintained by a list:

```
{print out the values held in the list}
aList.onEach(function(val : integer);
      begin
         print(val);
         print(" ");
      end);
```

The function passed as argument to `onEach` is created directly in the argument list. Notice this function does not have a name, and no other extraneous variables have been declared. In all other respects the syntax used to create the function is exactly the same as the syntax used in normal functions.

This concept of creating a method like `onEach`, that takes a function as an argument and applies it to all elements of a list is an example of what we call the Functional Backpack Pattern. This pattern becomes obvious only after the functional elements of Leda are combined with its object-oriented aspects, as in the prior examples. Such a pattern has an almost limitless applicability in performing functions iteratively on list of objects. It also adequately meets the criteria set by Gamma's Iterator Pattern.

4.3 Using Relations and Backtracking

Since Leda supports the logic programming paradigm, it naturally includes an automatic backtracking facility. An alternative approach to the problem of iteration can be constructed by making use of this mechanism. Although not exactly logic programming per se, this solution is in the spirit of the logic programming facilities in Leda.

Backtracking in Leda is delimited by a `for` or an `if` statement. Within the expression portion of such a statement, the programmer can invoke a relation. A relation is simply an expression of type `relation`, most generally constructed out of the relational assignment operator, Boolean expressions, logical operator, or functions that return relations. The Boolean `or` operator naturally defines a "solution" to a relational expression as a sequence of alternative choices, trying each alternative solution in turn. In the case of lists, a "solution," as implemented by the relation named `items`, is simply a binding of a value in the list to the argument variables. Using the `for` statement in this fashion, the following code fragment will print the values held in a list, separated by spaces:

```
{print out the values held by a list}
for aList.items(i) do begin
   print(i);
   print(" ");
   end;
```

This approach to iteration, straightforward and simple, is the approach most commonly used when programming in Leda. As with several of the other sample functions we have included, the implementation of the `items` relation is accomplished by a combination of methods in both `List` and `Link` classes, the bulk of the work being performed by the latter class. The argument to both methods is passed by reference, so changes made within the method will be reflected as changes made to the actual argument. The definition in class `List` is as follows:

```
class List [T : equality];
var
   data : Link[T];
   .
   .
   .
   function items(byRef val : T)->relation;
   begin
      {enumerate the values held in list}
      return defined(data) & data.items(val);
   end;
   .
   .
   .
end;
```

The backtracking takes place within the `Link` class. At each step through this recursive procedure are two alternatives. The first alternative invokes the `unify` relation, which binds the value of the first argument to the quantity given in the second. The second alternative performs a recursive call, which subsequently binds the remaining value to the variable.

```
class Link [T :equality];
var
   datum : T;
   next : Link[T];
   .
   .
```

```
    function items(byRef val : T)->relation;
    begin
       return unify[T](val, datum)
           | defined(next) & next.items(val);
    end;
    .
    .
end;
```

The `or` statement, when used with relational values, is the key component of the backtracking mechanism. Each alternative possibility in an `or` expression is systematically executed in conjunction with the statement portion of the `for` or `if` statement. In this manner every element in the list will eventually be generated.

This pattern of viewing iteration as the searching through a potential solution space becomes obvious when logical elements of Leda are incorporated into an object-oriented framework. A wide variety of traversal problems can henceforth be viewed as search problems, and relations can be used to traverse structures in any way needed to return the elements in the desired way.

4.4 Data Generators

Our last variation on the Iterator Pattern involves data generators. A generator is a function that yields a sequence of values, one new value each time it is invoked. Probably the most common form of generator function is a random number generator, which yields a new pseudo-random integer value on each call. But generators can be constructed for other tasks. In particular, we can tackle the task of iteration by creating a generator that will yield, on each successive invocation, a different value from the underlying data structure.

If the functional solution as implemented by the method named `onEach` is thought of as "bundling up the action to be performed and handing it to the list," then the use of generators is almost exactly the reverse. Namely, the list itself is bundled into a structure that will yield each element in turn, and it is this structure that the programmer receives to manipulate.

While a random number generator can be considered to be producing a sequence that is infinite in length, the generators for data structures produce a finite sequence. Thus some protocol must be observed for indicating the end of the sequence. The technique we use is to produce the value `NIL` when the values are exhausted.

Imagine we add a method `generate` to the `List` class in order to create a list generator. Then the task of printing all the values held in a list might be implemented as follows:

```
var
    aList : List[string];
    gen : function()->string;
    val : string;
    .
```

```
.
.
gen := aList.generator();
val := gen();
while defined (val) do
begin
   print(val);
   print(" ");
   val := gen();
end;
```

Note that the generator itself is stored in a variable that is declared as type `function`. The generator function requires no arguments, but it does produce on each call a value of the appropriate type.

To implement the generator operation, we define a function that returns another function as a result. Within the generator function we can define a pointer to the current element. This value is initially set to reference the first element in the list and is continually modified as each element of the returned function takes place. Within the generator function itself another local variable is used to hold the value being returned. If the pointer `current` references a legal value, then this variable is set, otherwise once all values have been enumerated it remains undefined.

```
class List [T : equality];
var
   data : Link[T];
   .
   .
   .
   function generator()->function()->T;
   var
      current : Link[T];
   begin
      current := data;
      return function()->T;
         var
            val : T;
         begin
            if defined(current) then begin
               val := current.datum;
               current := current.next;
            end;
            return val;
         end;
   end;
   .
   .
   .
end;
```

5. Conclusion

Iteration may seem like a relatively simple and straightforward pattern, when viewed from a strictly object-oriented perspective. But when the potential of functional and logical programming elements are included, our vision expands. Not only do we see different ways to perform iteration, but we see different implications within the various designs. For example, the ability to send a function along as a parameter to an iterator to either print the elements or perform some other function opens up other interesting design patterns. Similarly, the use of generators as a method for iteration opens up other avenues of design.

In the introduction to the first collection of PLoP papers [4], Johnson and Cunningham state, "We expect that as the PLoP community grows and matures that PLoP will itself splinter along traditional lines of interest." While it's clear that PLoP will expand along the lines of traditional computer science problem domains, it seems equally valuable that some expansion also take place along the lines of programming paradigms, to optimize the ability of patterns to suggest new, effective solutions to problems.

We believe that the object-oriented paradigm currently holds the greatest potential as a mind set within which to design and implement high-level solutions. But we also believe that programming languages will continue to be more inclusive of elements from diverse paradigms. The study of multiparadigm patterns of design not only suggests ways in which multiparadigm languages can and should be used, but it suggests ways in which evolutionary languages (such as C++) can and should adopt elements from other paradigms. Independent of programming issues, multiparadigm patterns of design expand our horizons in ways not normally suggested by a single paradigm, whatever that paradigm might be.

References

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. New York: Oxford University Press, 1977.

[2] Christopher Alexander. *The Timeless Way of Building*. New York: Oxford University Press, 1979.

[3] Timothy A. Budd. *Multiparadigm Programming in Leda*. Reading, MA: Addison-Wesley, 1995.

[4] James O. Coplien, and Douglas C. Schmidt, Editors. *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[6] David L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, 15(12):1059-1062, 1972.

[7] Edward Sapir. *Language: An Introduction to the Study of Speech*. New York: Harcourt Brace Jovanovich, 1921.

[8] Benjamin Lee Whorf. *Language Thought and Reality*. Cambridge, MA: MIT Press, 1956.