Envoy

A Pattern Language for Managing State in a Functional Program

Eugene Wallingford
wallingf@cs.uni.edu

Introduction

Envoy is a pattern language for managing state in functional programs. It simulates an object-oriented programming (OOP) style using the features of a statically-scoped functional programming language such as Scheme.[1]

Functional programming and object-oriented programming provide alternative ways to address the difficulties of writing and maintaining large programs. Both styles help to solve the problems that result from unrestricted access to global data: functional programming by prohibiting global state, and OOP by encapsulating global state into more manageably-sized units.

In the functional style, a program never changes the value of a variable. Indeed, there is no sense of "variable", since symbols are really names for values, not names for slots that hold values. But this style becomes uncomfortable when dealing with problems that are more naturally seen as operating on data than as computing values.

Rather than forcing the solution to such problems to fit the style, functional programmers often step outside the style to program with state. This sometimes takes the form of using an OOP language to write part or all of a program. However, integrating code written in some interpreted languages with code written in other languages can often be difficult. Furthermore, the programmer may simply want to create a new abstract data type with encapsulated operations, and implementing the type outside the language imposes costs greater than the benefits it gains.

Envoy describes a set of structures that a functional programmer might use to write programs that represent and maintain state. These structures use constructs provided by the functional language to simulate an OOP style.

---

[1] All examples in this paper are given in Scheme. For a brief summary of the subset of Scheme used in these examples, see the appendix "A Scheme Primer".

The Patterns

Envoy consists of eight patterns:

1. Function as Object
2. Closure
3. Constructor Function
4. Method Selector
5. Message-Passing Interface
6. Generic Function
7. Delegation
8. Private Method

Pattern 1 is the entry point to the language, the pattern that denotes the use of an OOP style. Pattern 2 shows how to create a function with state. Patterns 3 and 4 show how to define an object that responds to messages. Patterns 5 and 6 describe ways to interact with an object. Patterns 7 and 8 show how reuse objects and method code.

Envoy also refers to a number of patterns documented elsewhere. Thumbnail sketches of these patterns appear in the External Patterns section at the end of the paper.

Finally, the patterns that constitute Envoy relate in various ways to other techniques and ideas outside the realm of functional programming. The Related Ideas section discusses of some of the more interesting relationships.

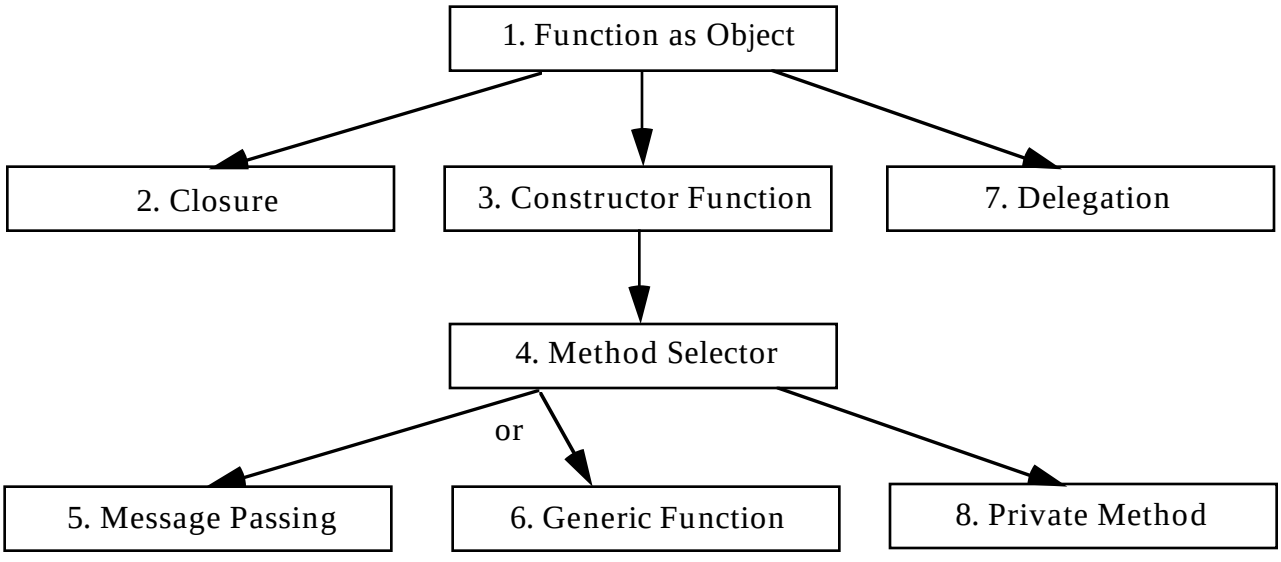

Figure 1. A map of the patterns in Envoy.

1. Function as Object

You are writing Simply Understood Code (Gabriel) in a statically-scoped functional programming language.

How do you represent and maintain state that can change over time?

The idea of functional programming is to write programs as stateless collections of functions. Each function refers only to those data values passed to it as arguments and always returns the same value for any particular set of inputs. By writing code in this way, a functional programmer realizes two benefits. First, the mathematical theory of functions makes it relatively straightforward to prove the properties of a functional program. Second, since all data are local to a single function, modifying and maintaining individual functions are simplified.

But you may find that your program is telling you to use state information to simplify. Perhaps you are passing too much common data to a number of different functions. Or perhaps your program's task can be viewed better as modeling action on data than as computing a value.

One such situation is when you are constructing an abstract data type, or ADT. Suppose that you were implementing a bank account ADT. At its simplest, this ADT is a single data value, a balance, and a set of associated operations. You might implement this ADT something like this in a functional style:

```
(define balance 0)

(define withdraw
   (lambda (balance amount)
      (if (<= amount balance)
          (- balance amount)
          (error "Insufficient funds" balance))
   ))

(define deposit
   (lambda (balance amount)
      (+ balance amount)
   ))

(define accrue-interest
   (lambda (balance interest-rate)
      (+ balance (* balance interest-rate))
   ))
```

You could create new accounts simply by binding values to names. Operating on accounts involves passing the account to the appropriate procedure and binding the new value as appropriate.

This solution is simple and will work, but it increases the chance of programmer error in several different ways:

- First, you must remember to bind the new value of the account balance after every call to an account operation.

- Second, and more serious, you do not need to use the `withdraw` function to reduce the value of an account balance! You can use the expression `(- account amount)` directly. This introduces the possibility that a balance can become negative, which violates a postcondition of the withdraw operation.

- Third, if you want to create specialized bank accounts, such as an account that logs all transactions to a file, then you must do something to guarantee that the "regular" bank account operations are not applied to the new kind of accounts.

The source of these difficulties is the separation of the data value from the operations that act on it. An ADT is defined not just by its data and not just by its operations, but by its data and its operations together. A pure functional style makes it difficult to represent this combination.

Therefore, create a function that acts like an object.

Such a function carries the data it needs along with the expression that operates on the data. More importantly, an object encapsulates its data, ensuring that only the allowed operations are applied to them.

The primary drawback to this kind of a function is that it behaves differently from the rest of the functions in your program. Evaluating the function can change its encapsulated state, which means that the function's value can be different for two function calls with the same arguments. This hampers the code's understandability.

This approach also makes it difficult for you to prove formally the properties of your function, such as correctness. When you write functional programs that do not involve side effects, you can use well-developed theory to prove such properties. But you cannot use the same theory to analyze a function that carries and modifies its own data, and no comparable theory exists for programs having side effects. In practice, though, programmers rarely have the time or resources to do large-scale proofs of correctness or other properties anyway. This pattern surrenders a rarely used benefit to achieve other goals, including reduced risk of programmer error.

Use a Closure (2) to create a function that encapsulates data defined outside itself. Make a Constructor Function (3) that returns such a closure as its answer. [The closure itself should be a Method Selector (4) that selects a method to run based on an argument.] If you need to create an object that extends an existing object, use Delegation (7).

2. Closure

You are writing a function with a free variable. Perhaps you are creating a Function as Object (1).

How do you bundle a function with a data value defined outside the procedure's body?

Suppose that you were building a model of a bank account. You would like to be able to withdraw funds from the account. The simplest way to bundle a data value with a function is to define the value within the function's body:

```
(define withdraw
   (lambda (amount)
      (let ((balance 100))
         (if (>= balance amount)
             (begin
                (set! balance (- balance amount))
                balance)
             (error "Insufficient funds" balance))
      )))
```

But this solution fails, because each evaluation of the function creates a new and temporary instance of `balance` and initializes its value to 100. For example:

```
(withdraw 20)    =>  80
(withdraw 60)    =>  40
(withdraw 30)    =>  70
```

The problem, of course is that local variables do not persist beyond the current evaluation. So you might try defining the balance outside the function:

```
(define balance 100)

(define withdraw
   (lambda (amount)
      (if (>= balance amount)
          (begin
             (set! balance (- balance amount))
             balance)
          (error "Insufficient funds" balance))
   ))
```

In this function, `balance` is a *free variable*. This means that `withdraw` refers to `balance`, but the name `balance` is not defined within the function. When the function is evaluated, though, `balance` is bound to the value defined for the name at the top level of the interpreter. Free variables can make a piece of code harder to understand, since the programmer must look outside the code for some of the information needed to understand it.

This approach seems to work fine, and `withdraw` now behaves as you had hoped. The problem with this definition is that the balance is external to the definition of `withdraw`, which means that *other* procedures can also modify its value:

```
: (withdraw 20)          ;; I need a little cash.
80

: balance
80

: (set! balance 30)      ;; Hey!  What's going on here??
30

: (withdraw 60)          ;; I need a little more cash, but...
*** ERROR -- Insufficient funds 30
```

So, this approach achieves persistence of the data at the unacceptable cost of surrendering encapsulation. You are no longer able to guarantee that the account balance will be changed only by your `withdraw` function. What you seem to need is a blend of these solutions: data that is *relatively* global to the function, but not global to everyone.

Therefore, create the function in an environment where its free variables are bound to local variables.

For the account ADT, define the `withdraw` function within the body of an expression that binds the variable `balance`, like this:

```
(define withdraw
   (let ((balance 100))          ;; balance is defined here,

      (lambda (amount)
         (if (>= balance amount) ;; so this reference is bound
             (begin
                (set! balance (- balance amount))
                balance)
             (error "Insufficient funds" balance)))

      ))
```

Since Scheme is statically-scoped, the interpreter saves a copy of the binding for `balance` that exists at the time the function body itself is defined. This package of function and variable bindings is called a *closure*. This term derives from the idea that this function is now “closed” to the state of the outside world—it can be used in any environment, independent of the bindings that exist in that environment, because all of the identifiers it uses are bound either to arguments, local variables, or the closure’s external bindings.

Each evaluation of `withdraw` refers to the same `balance` object, whatever its current value. For example:

```
(withdraw 20)    =>  80
(withdraw 35)    =>  55
(withdraw 80)    =>  *** ERROR -- Insufficient funds 55
```

Scheme will create a closure with the binding for a free variable regardless of how the binding is created. This example uses a `let` expression to create a local binding, but the same sort of closure is created if the binding results from a `lambda` expression's formal parameter or even from a binding at the top-level.

A Closure (2) achieves both persistence and encapsulation by creating a data object that is global to the function definition but inaccessible to the rest of the program. These benefits come at two costs: the use of a free variable and use of the `set!` operation.

- As noted above, free variables can make a piece of code harder to understand by requiring the programming to read beyond the piece of code. A Closure (2) minimizes this cost by defining the object to which the free variable refers immediately outside the `lambda` expression.

- Using `set!` enables you to write code that changes a variable's state. But `set!` treats an identifier as the name of a slot that holds a value, which runs undermines a fundamental tenet of functional programming: the computation and naming of values. Unrestricted use of such *mutators* can result in code that is hard to read and thus to modify. A Closure (2) minimizes this cost as much as possible by using `set!` only to modify the value of the free variable that it encapsulates.

Use a Constructor Function (3) if you need to create multiple copies of the closure or if you need the ability to initialize the closure's state in client code.

3. Constructor Function

You are creating a Function as Object (1) using a Closure (2).

How do you create instances of the object?

Your definition of `withdraw` creates a Closure (2), a procedure bound to a data value that lies outside the procedure. Your function provides both encapsulation and persistence of the data value. But in some situations, you may need more flexibility:

- Your definition allows only one account to exist at any point in time. Every time you evaluate the `withdraw` function, you create a new closure, with its balance set to 100.

- Your definition "hard-wires" the initial bank account balance as 100. You may want to allow clients to create an account with a specific initial balance. If you offer the ability to create multiple instances of the object, then clients may want to open different accounts with different initial balances.

In some situations, you would like to use the same function definition to create multiple instances of the object without duplicating code. You would also like to be able to customize objects with different starting states.

Therefore, make a function that returns your Function as Object (1). Give the function an Intention Revealing Name (Beck), such as `make-`*`object`*.

Re-write your definition of `withdraw` as:

```
(define make-withdraw
   (lambda (balance)
      (lambda (amount)
         (if (>= balance amount) ;; balance is still bound,
             (begin      ;; but to a new object on each call!
                (set! balance (- balance amount))
                balance)
             (error "Insufficient funds" balance)))
   ))
```

`make-withdraw` is a procedure that takes a single argument, the initial account balance. Evaluating `(make-withdraw 60)` returns a withdrawal procedure that works on a **new** bank account object having an initial balance of 60. Each call to `make-withdraw` produces a new procedure that is, in effect, an instance of a bank account. We could use `make-withdraw` in this way:

```
(define account-for-eugene (make-withdraw 100))
(account-for-eugene 20)     =>  80
(define account-for-tom (make-withdraw 1000))
(account-for-tom 20)        =>  980
```

```
(account-for-eugene 20)     =>  60
(account-for-eugene 120)    =>  *** ERROR ...
(account-for-tom 120)       =>  860
```

If you need to create an object whose instances always start with the same default state, wrap the function value being returned in a local variable definition, and make the constructor function take no arguments. For instance, if you want all bank accounts to start with $100 balances, write:

```
(define make-withdraw
   (lambda ()
      (let ((balance 100))
         (lambda (amount)
            (if (>= balance amount)
                (begin
                   (set! balance (- balance amount))
                   balance)
                (error "Insufficient funds" balance)))
      )))
```

You can, of course, mix these two approaches if your object has some values that are initialized by the user and some that are initialized to default values.

A Constructor Function (3) is a bit more complex than a standard Closure (2) because it nests one `lambda` expression inside another.

If your object needs to provide more than one operation, then make your constructor function return a Method Selector (4).

4. Method Selector

You are creating a Function as Object (1) using a Closure (2). A Constructor Function (3) creates new instances of the object.

How do you provide shared access to the closure's state?

To model a real bank account, you will need the ability to deposit money as well as to withdraw it. Suppose that you add a Closure (2) for the `deposit` function:

```
(define make-withdraw
   (lambda (balance)
      (lambda (amount)
         (if (>= balance amount)
             (begin
                (set! balance (- balance amount))
                balance)
             (error "Insufficient funds" balance)))
   ))

(define make-deposit
   (lambda (balance)
      (lambda (amount)
         (set! balance (+ balance amount))
         balance)
   ))
```

This solution fails, though, because the two functions are closed on different `balance` variables. Invoking `deposit` has no effect on the balance that `withdraw` operates on, and vice versa. You need for the functions to share a single `balance` variable. You could try defining `balance` outside of the two functions, but then your functions no longer encapsulate the data they modify.

Even if you find a way to resolve this problem, you face another difficulty. Defining `make-withdraw` and `make-deposit` separately obscures the fact that these functions are closely coupled by their shared variable. Worse still, you can define functions that share state anywhere in your program file, or even in different files. Separation makes it harder for a human reader to understand the relationships among the functions and increases the risk of programmer error when modifying them.

Therefore, make the function returned by your Constructor Function (3) take one argument, a symbol that names a method. Make the value of the returned function be a selection statement with one arm for each method. Use the symbol argument to make the choice of which method to return. Add an arm that handles an invalid selector in a suitable fashion.

For your bank account, implement the following procedure:

```
(define make-account
   (lambda (balance)

      (lambda (transaction) ;; The procedure uses its argument
                            ;; to select an operation on
         (case transaction  ;; balance--returning a procedure!
            ('withdraw
               (lambda (amount)
                  (if (>= balance amount)
                      (begin
                         (set! balance (- balance amount))
                         balance)
                      (error "Insufficient funds" balance))))
            ('deposit
               (lambda (amount)
                  (set! balance (+ balance amount))
                  balance))
            ('balance
               (lambda ()
                  balance))
            (else
               (error "Unknown request -- ACCOUNT"
                      transaction))))
   ))
```

You can now invoke deposit and withdrawal operations on the same account:

```
(define account-for-eugene (make-account 100))
((account-for-eugene 'withdraw) 10)             =>  90
((account-for-eugene 'withdraw) 10)             =>  80
((account-for-eugene 'deposit) 100)             =>  180
(define account-for-tom (make-account 1000))
((account-for-tom 'withdraw) 10)                =>  990
((account-for-eugene 'withdraw) 10)             =>  170
```

Furthermore, the account balance is still encapsulated from outside access. Each call to `make-account` creates a procedure that has its own local binding for the variable `balance`.

The resulting procedure is a “selector”. It returns either the withdrawal procedure or the deposit procedure, depending upon the argument it is sent. These procedures are created in an environment containing a binding for the variable `balance`, so they, too, are bound to `balance` in the closure. Thus, they access the same `balance` variable. The deposit operation for Eugene’s account refers to the balance in its closure, while the deposit operation for Tom’s account refers to its own balance. This is a more general use of the closure, in which mutable data are local to a *set* of procedures.

These benefits come at the cost of an extra function call. Each time you withdraw money from an account, you must first retrieve the withdraw procedure from the object's closure. In a large program with many accounts, these extra calls can have a significant effect on the program's run-time performance.

Using a Closure (2) in this way also makes explicit the coupling between the two functions, by defining the functions within the same closure. This technique has the potential to improve the understandability of the code and to reduce the risk of programmer error when modifying one of the functions. However, the client code that uses a Method Selector (4) is difficult to read. Create a Message-Passing Interface (5) or a set of Generic Functions (6) to help users interact with your objects.

Two functions within a selector may use common code to compute their results. Consider using a Private Method (8) to eliminate such duplication.

5. Message-Passing Interface

You have created a Method Selector (4) for a Function as Object (1). You prefer to use your object in code that has an object-oriented feel.

How do you invoke the methods of an object?

Consider your bank account listed above. To use the value returned by `make-account`, you have to remember to invoke it as a function:

```
(define account-for-eugene (make-account 100))
((account-for-eugene 'withdraw) 10)            =>  90
((account-for-eugene 'deposit) 45)             =>  135
((account-for-eugene 'balance))                =>  135
```

The nesting of parentheses in the function position of your expressions is unlike most of your code. The functions returned by the closure require different number of arguments. Both of these features make writing code that uses the closure prone to error and hard to read.

You may want to use your objects in code that closely follows the OOP message-passing metaphor. By creating objects with message protocols, you are able to plug new objects into existing code in place of other objects with the same protocol.

Therefore, provide a simple message-passing interface for using the closure.

Making new syntax is one of Scheme's strengths, so you should not be surprised to learn that we can simulate a message-passing syntax with relatively little effort. Create a procedure `(send object message arg1 ... argn)`, where:

- `object` is a Closure (2) that returns procedure values,
- `message` is a symbol that corresponds to a message, and
- `arg1` through `argn` the *n* arguments that `object` expects to receive when sent message.

```
(define send
   (lambda argument-list
      (let ((object  (car argument-list))
            (message (car (cdr argument-list)))
            (args    (cdr (cdr argument-list))))
         (apply (get-method object message) args))
   ))

(define get-method
   (lambda (object selector)
      (object selector)
   ))
```

`send` uses `get-method` to retrieve the function corresponding to `message` in `object`. It then uses `apply` to call this function with `arg1` through `argn` as arguments, returning the value of the function call as its value. For example:

```
(define account-for-eugene (make-account 100))
(send account-for-eugene 'withdraw 50)          => 50
(send account-for-eugene 'deposit 100)          => 150
(send account-for-eugene 'balance)              => 150
```

Note that `send` must be accept any number of arguments two or greater. It requires two arguments, an object and a message. Any other arguments will be the values passed to the selected method.

This pattern allows you to use more standard functional notation when interacting with your objects, but it does not behave like other Scheme functions. If you need to use your objects in a truly functional way, consider using a Generic Function (6) instead.

This solution also imposes other costs, such as the overhead of extra function calls. If these costs become unacceptable, then you might implement this pattern using a syntactic extension to the language, such as a macro. Use of a language extension may simplify the user's task as well, but at the cost disorienting future programmers who are unfamiliar with the extension.

6. Generic Function

You have created a Method Selector (4) for a Function as Object (1). You want to take full advantage of the tools available in your functional language.

How do you invoke the methods of an object?

You must invoke a Method Selector (4) as a function in order to access one of its methods:

```
(define account-for-eugene (make-account 100))
((account-for-eugene 'withdraw) 10)             =>  90
((account-for-eugene 'deposit) 45)              =>  135
((account-for-eugene 'balance))                 =>  135
```

One way to avoid this uncomfortable syntax is to use a Message-Passing Interface (5), which closely simulates the OOP message-passing metaphor. If you wish to create a program that has an OOP feel, then this works fine.

But `send` works differently than most Scheme functions. Suppose that you had a list of objects called `bank-accounts` and that you wished to compute a list of their balances. In Scheme, you would ordinarily write an expression something like:

```
(map balance bank-accounts)
```

But `balance` is not a function; it is only a symbol that allows an object to select and evaluate the appropriate function. So, instead you would have to write:

```
(map (lambda (account)
       (send account 'balance))
     bank-accounts)
```

This requires users of your objects to remember that they behave differently from other functions. Such a lack of consistency increases the chance that a user will misunderstand the code or make errors while using it. You may want to maintain the benefits of a Method Selector without giving up the ability to use functional programming tools or techniques in a natural way.

Therefore, provide a simple interface to the Method Selector (4) that more closely follows the functional style. Write a generic Interface Procedure (Wallingford) for each method defined for the object. For your bank account objects, write:

```
(define withdraw
   (lambda argument-list
      (let ((object             (car argument-list))
            (withdraw-arguments (cdr argument-list)))
        (apply (object 'withdraw) withdraw-arguments)
      )))
```

```
(define deposit
   (lambda arguments
      (let ((object            (car arguments))
            (deposit-arguments (cdr arguments)))
         (apply (object 'deposit) deposit-arguments)
      )))

(define balance
   (lambda (object)
      (object 'balance)
   ))
```

This allows you to write code of the form

```
(withdraw an-account an-amount)
```

instead of

```
(send an-account 'withdraw an-amount)
```

It also allows users to use standard Scheme tools such as `apply` and `map` with your objects. For example, you can now compute a list of their balances from a list of account objects using:

```
(map balance bank-accounts)
```

You can implement this pattern on top of a Message-Passing Interface (5), if you wish to use both styles of interaction in the same program. But mixing these styles may confuse programmers unless they are told why you are using a particular style in a particular part of the program.

This pattern makes it possible for you to use objects in a way that feels very similar to ordinary functional programming. One potential drawback, though, is that these objects have mutable state, even though the code that uses them looks functional.

For example, suppose that you have a list of accounts called `bank-accounts` and that you wish to deposit 10 dollars into each. You might write:

```
(map (lambda (account) (deposit account 10)) bank-accounts)
```

Since the deposit operation changes the state of an account object, you have just modified the state of every account in the list `bank-accounts`. This is not how ordinary Scheme functions behave.

Like Message-Passing Interface (5), this pattern imposes other run-time costs that may be reduced by using a syntactic language extension, at the expense of requiring readers to learn new constructs.

7. Delegation

You are creating a Function as Object (1).

How do you create a new object that extends the behavior of an existing object?

Sometimes, you want to create a Function as Object (1) that can be thought of as an extension of an existing object. For example, an interest-bearing bank account is a bank account that knows how to accrue interest. That is, an interest-bearing bank account can respond to the same message that a "regular" account, plus perhaps some other, like `accrue-interest`.

You could simulate this relationship using a very un-technical technique: copy-and-paste. You could define a new closure, `interest-bearing-account`, with the desired interest accrual behavior, by starting with the code from `account`:

```
(define make-interest-bearing-account
   (lambda (balance interest-rate)
      (lambda (transaction)
         (case transaction
            ('withdraw
               (lambda (amount)
                  (if (>= balance amount)
                      (begin
                         (set! balance (- balance amount))
                         balance)
                      (error "Insufficient funds" balance))))
            ('deposit
               (lambda (amount)
                  (set! balance (+ balance amount))
                  balance))
            ('balance
               (lambda ()
                  balance))
            ('accrue-interest
               (lambda ()
                  (set! balance
                        (* balance (+ 1.0 interest-rate)))
                  balance))
            (else
               (error "Unknown request -- ACCOUNT"
                      transaction))))
   ))
```

The result is a working interest-bearing account object that behaves just like a regular account, except that it also can accrue interest.

But what happens if you need to change or extend how "plain" bank accounts work? Suppose that you need to add an account number to the object, along with corresponding access methods. You will have to modify the procedure `make-`

`account`—but you will also have to modify the procedure `make-interest-bearing-account`!

One of the central ideas of object-oriented programming is that you should "say it once, and only once". This idea calls for you to re-use existing code rather than re-implement it. You would like to find a way to mimic such re-use in your bank account program.

In an object-oriented programming language, you might use inheritance to implement this relationship, so that an `interest-bearing-account` closure could be defined as an `account` extended with interest-accrual capability. But your functional language doesn't support inheritance directly, and implementing such a facility using syntactic extension may be unattractive. It requires the creation of a new special form, which makes the code harder to read and understand.

If you send an `accrue-interest` message to an interest-bearing account, it should execute a method defined in its closure. But if you send it message common to all accounts, such as `deposit`, it can execute the method defined in the account object.

Therefore, use delegation. Make a Function as Object (1) that has an instance variable an instance of the object you want to extend. Implement behaviors specific to the new object as methods in a Method Selector (4). Pass all other messages onto the instance variable.

For example, define `interest-bearing-account` to have an `account` object as an instance variable. `make-interest-bearing-account` can initialize it with the account's initial balance. Whenever the `interest-bearing-account` receives an `accrue-interest` message, it handles it directly by interacting with the instance variable. Otherwise, it simply passes the message on to its `account` object.

Here is how the `make-interest-bearing-account` constructor would look:

```
(define make-interest-bearing-account
   (lambda (balance interest-rate)
      (let ((my-account (make-account balance)))
         (lambda (transaction)
            (case transaction
               ('accrue-interest
                  (lambda ()
                     ((my-account 'deposit)
                        (* ((my-account 'balance))
                           interest-rate)) ))
               (else
                  (my-account transaction))
               )))
))
```

Here is a sample run:

```
(define foo (make-interest-bearing-account 100 0.05))
((foo 'balance))                                    =>  100
((foo 'deposit) 100)                                =>  200
((foo 'balance))                                    =>  200
((foo 'accrue-interest))                            =>  210.
((foo 'balance))                                    =>  210.
```

Notice one difference between your new implementation of the `accrue-interest` method and the one in the first cut above. Since the interest-bearing account's balance is now managed by an account object, it is encapsulated—that is, hidden from all users, including the interest-bearing account! So, when we need to access that object in order to accrue interest, we must use the public interface of the account.

This pattern offers a nice way to simulate inheritance, since it can be implemented in many languages that do not support OOP or inheritance directly.

As in the case of inheritance, this pattern imposes a run-time penalty. Inherited methods require an extra function call to delegate the message to the encapsulated account. If the encapsulated object is also implemented using a Method Selector (4), then this solution requires additional calls to access and evaluate the method.

This use of delegation does not solve the problem caused by redefining the account object. If you redefine `make-account`, then you will need to recreate all of your interest-bearing accounts with calls to the new account constructor. If that sort of long-term is maintenance is needed for your objects, then you need to step out of this pattern language and use some other technique to make idea of a *class* explicit. Programmers could then define one class as an extension of another.

8. Private Method

You have created a Method Selector (4).

How do you factor common behavior out of the methods in the Method Selector?

Suppose that you need to log all transactions that modify an account's balance. You might add a second data value, `transaction-log`, to the Closure (2) and have all procedures that modify the balance update the new data value:

```
(define make-account
   (lambda (balance)
     (let ((transaction-log '()))
        (lambda (transaction)
           (case transaction
              ('withdraw
                 (lambda (amount)
                    (if (>= balance amount)
                        (begin
                           (set! balance (- balance amount))
                           (set! transaction-log
                                 (cons (list 'withdraw amount)
                                       transaction-log))
                           balance)
                        (error "Insufficient funds" balance))))
              ('deposit
                 (lambda (amount)
                    (set! balance (+ balance amount))
                    (set! transaction-log
                          (cons (list 'deposit amount)
                                transaction-log))
                    balance))
     ...)))
```

But the new code in the withdraw and deposit methods is nearly identical, which creates a potential maintenance problem. You could try to eliminate this duplication by making each of these methods a Composed Procedure (Beck) that calls a common function:

```
(define log-transaction
   (lambda (type amount)
      (set! transaction-log
            (cons (list type amount) transaction-log))
   ))
```

This straightforward solution fails for two reasons:

- Since you define the common behavior in a function at the top level, other clients can use it. But this function exists only to support methods defined within the Method Selector (4).

- Even worse, since you define the function outside the Closure (2), the new function cannot access the data values shared within the Method Selector (4)!

You would like define the common behavior in a way that preserves data sharing and encapsulation without duplicating code.

Therefore, define the common code in a Local Procedure (Wallingford). Invoke this procedure in place of the duplicated code within the Method Selector (4).

In the bank account example, change your definition to the following:

```
(define make-account
   (lambda (balance)
      (let ((transaction-log '())
            (log-transaction
               (lambda (type amount)
                  (set! transaction-log
                        (cons (list type amount)
                              transaction-log))))  )
        (lambda (transaction)
           (case transaction
              ('withdraw
                 (lambda (amount)
                    (if (>= balance amount)
                        (begin
                           (set! balance (- balance amount))
                           (log-transaction 'withdraw amount)
                           balance)
                        (error "Insufficient funds" balance))))
              ('deposit
                 (lambda (amount)
                    (set! balance (+ balance amount))
                    (log-transaction 'withdraw amount)
                    balance))
     ...)))
```

This pattern simulates a private method, a common construct in object-oriented programming.

Using a Private Method (8) imposes the cost of an extra function call whenever one of its client methods is evaluated. Perhaps worse is the loss of readability that results from nesting the the Method Selector (4) within a `let` expression. In most situations, though, preserving encapsulation and eliminating duplication are worth complicating the definition of the Function as Object (1).

External Patterns

The following patterns are not a part of Envoy but are referred to by one or more of its patterns. Links to on-line versions of the patterns are given where available.

- Composed Procedure (Called “Composed Method” by Beck)
    - Problem: How should you divide a program into components?
    - Solution: “Divide your program into [procedures] that do one identifiable task. Keep all of the operations in a [procedure] at the same level of abstraction. This will naturally result in programs with many small [procedures], each a few lines long.”
- Intention Revealing Name (Called “Intention Revealing Selector” by Beck)
    - Problem: How do you name a procedure?
    - Solution: Name procedures for what they accomplish.
- Interface Procedure (Wallingford)
    - Problem: How do you hide the user of a procedure from the details of the procedure’s interface?
    - Solution: Create a new procedure that serves as the public interface to the client code.
- Local Procedure (Wallingford)
    - Problem: How do you eliminate duplicated code without creating a global procedure?
    - Solution: Make a procedure that is local to its clients.
- Simply Understood Code (Gabriel)
    - Problem: People need to be comfortable reading a piece of code before they feel confident that they understand it and can modify it.
    - Solution: “Arrange the important parts of the code so it fits on one page. Make that code understandable to a person reading it from top to bottom. Do not require the code to be repeatedly scanned in order to understand how data is used and how control moves about.”

.

Related Ideas

Envoy resides at the intersection of functional programming, especially in Lisp-based languages, and object-oriented programming, so you should not be surprised that its patterns relate to a number of ideas outside its direct purpose.

Closures

The concept of a closure plays a fundamental role in the implementation of block-structured languages. In languages that allow procedures to be passed as arguments, a closure is typically used to represent procedure parameters. This allows the compiler to support static scope, since the closure associates the procedure with the bindings of its free variables in effect at the time that the procedure was created.

Closures also play an important role in functional programming, but this time as a standard programming tool. In addition to simulating OOP, functional programmers uses closure for at least two other common purposes: to create function generators and to curry procedures (for example, to create mappable `lambda`s).

Even in object-oriented languages such as Java and Smalltalk, closures serve as the basis for numerous idioms. Java supports nested functions and classes, both of which enable closures on references to variables defined in the enclosing context. Smalltalk makes extensive use of blocks, which also permit free variable references.

In these languages, you do not need to use closures to simulate OOP, but you can use them for other purposes. Often, closures are used to provide access to private state without breaking encapsulation. One application of this idea involves the use of closures to construct collection iterators. (See “Use Closures Not Enumerations”.) Sandu and Deugo document a number of contexts for using Java inner classes and Smalltalk blocks to create closures, including as lightweight implementations of several well-known design patterns.

Simulation of OOP

Envoy comprises patterns for simulating several basic features of object-oriented programming. You could certainly simulate other features of OOP by extending the structures generated by these patterns.

At least two authors have documented delegation as a pattern in object-oriented programming itself. Beck’s delegation patterns describe how to share common behavior in Smalltalk without using inheritance. They capture the forces that drive a programmer to use delegation concisely and elegantly. Deugo presents delegation at the level of programmers just learning to do OOP, with more detailed discussion of forces and more detailed examples.

Norvig devotes one chapter of *Paradigms of Artificial Intelligence Programming* to the simulation of OOP in Common Lisp. Aimed at more advanced functional programmers, this chapter covers the basic ideas quickly and then shows how to use macros to simulate classes and inheritance.

Acknowledgements

I thank all those who helped me to improve Envoy at the 1999 Pattern Languages of Programming conference, including shepherd Dwight Deugo.

The ideas in this paper owe much to the work of two specific individuals: Phillip J. Windley, of Brigham Young University, whose on-line lecture notes helped me learn how to teach functional programming in the context of a programming languages course; and Peter Norvig, whose writing about AI programming has helped me reach a deeper understanding of functional programming techniques.

Finally, I thank Kent Beck for his encouragement to continue writing patterns at this level.

References

1. Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, New York, 1997.

2. Dwight Deugo, "Foundation Patterns", *Proceedings of the 1998 Pattern Languages of Programming Conference*, August 1998.

3. Richard Gabriel, "Simply Understood Code", quoted by Jim Coplien, "The Column Without a Name: Software Development as Science, Art, and Engineering," *C++ Report*, July/August 1995, pp. 14-19. Also appears at `http://c2.com/cgi/wiki?SimplyUnderstoodCode`

4. Dave Harris et al., "Use Closures Not Enumerations", `http://c2.com/cgi-bin/wiki?UseClosuresNotEnumerations`

5. Peter Norvig, *Paradigms of Artificial Intelligence Programming*, Morgan Kaufmann Publishers, San Mateo, California, 1996.

6. Dorin Sandu and Dwight Deugo, "The Lambda Pattern", to appear in *Proceedings of the 1999 Pattern Languages of Programming Conference*, August 1999.

7. Eugene Wallingford, "Roundabout, A Pattern Language for Recursive Programming", *Proceedings of the 1997 Pattern Languages of Programming Conference*, September 1997.

Appendix: A Scheme Primer

Scheme provides the standard types of primitive expressions. *Literal expressions* include numbers, booleans, characters, strings, and symbols. Scheme differs from most languages in its extensive use of symbols as data values. Here are some sample literal expressions in Scheme:

```
25      1.2      7/3                ;; numbers
a       +        argument-list      ;; symbols
```

Variable expressions are symbols that stand in the place of particular data values. In Scheme, procedures are values just like numbers and characters. You can name a procedure value using a symbol, just like any other variable expression. Scheme provides a large number of primitive procedures, including `+`, `>=`, and `apply`.

You combine primitive expressions to form compound expressions by applying a procedure to one or more arguments. Compound expressions use prefix notation:

```
(<procedure> <argument1> <argument2> ...)
```

For example, `(+ 1 2 3)` is a compound expression whose value is 6. `+` is the primitive addition procedure, and 1, 2, and 3 are literal number expressions. Procedure applications are evaluated by first evaluating the procedure and all of its arguments and then applying the procedure to the arguments.

All combinations are fully parenthesized, which eliminates the need for complex precedence rules like those found in most other languages. This benefit comes at the cost of nested parentheses, which can be difficult for the human reader to follow. Scheme programmers minimize this cost by carefully formatting their code and by creating procedures to eliminate excessive nesting.

Scheme's main means of abstraction is the definition. A definition looks just like a procedure application, with the `<procedure>` replaced by the keyword `define`:

```
(define  <name> <expression>)
```

The `<expression>` is evaluated, and the `<name>` is bound to that value. A definition is evaluated for its side effect—the naming of a value—and not for its value.

`define` is called a *special form*, because `define` expressions are not evaluated using the standard rule for procedure applications. Scheme provides a small set of special forms that define common control structures and abstraction mechanisms. The following special forms play an important role in Envoy's examples:

- `let` allows you to create local variables. The first argument to `let` is a list of variable bindings, in the form of `(variable value)` pairs, and the second is the

body of the expression. A `let` expression returns the value of its body, which is evaluated in a context binding the variables to their local values. For example, the following binds `x` to 1 and `y` to 2, and then returns the value of `(+ x y)`.

```
(let ((x 1) (y 2)) (+ x y))
```

- A `lambda` expression creates a procedure. Its first argument is a list of formal parameters, and its second argument is an expression. For example,

  ```
  (lambda (x) (* x x))
  ```

  creates a one-argument procedure that returns the square of its argument. The procedure is "nameless", though you can give it a name, using `define` or `let`, or by passing the procedure as an argument to another procedure. To name this procedure `square`, you would say `(define square (lambda (x) (* x x)))`.

  Sometimes, you need to create a procedure that takes any number of arguments. You can do this by giving `lambda` a single parameter instead of a parameter list:

  ```
  (lambda arguments ...)
  ```

  This `lambda` creates a procedure that binds the name `arguments` to a list containing all of the arguments passed to the procedure. The body of the `lambda` can then process the list of arguments however it sees fit.

- `if` and `case` denote standard control structures. They are special forms because only one of the alternative expressions is evaluated, depending on the value of the control expression.

- `set!` and `begin` allow the programmer to step outside of the functional style and program imperatively. `set!` assigns a new value to an existing variable. `begin` indicates a sequence of expressions to be evaluated in order. The value of a `begin` expression is the value of the last expression in its sequence.

Most of Scheme's primitive procedures have Intention Revealing Names (Beck). Unfortunately, not all do. Consider Scheme's procedures for manipulating lists. The list is Scheme's primary data structure, and list-processing facilities are among the oldest features of the language. Most of the Scheme's list-processing primitives retain their historical names. For example:

```
(car  '(a b c d))   => 'a          ;; the first element
(cdr  '(a b c d))   => '(b c d)    ;; the rest of the list
(cadr '(a b c d))   => 'b          ;; the second element
(cddr '(a b c d))   => '(c d)      ;; the "rest of the rest"

(cons 'a '(b c d))  => '(a b c d)  ;; add an item to a list
(list 'a 'b 'c 'd)  => '(a b c d)  ;; make a list from its parts
```