Advanced Programming Language Features for Executable Design Patterns "Better Patterns Through Reflection"

Gregory T. Sullivan
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
gregsai.mit.edu http://www.ai.mit.edu/~gregs

February 13, 2002

DRAFT

Abstract

The Design Patterns book [GHJV94] presents 23 time-tested patterns that consistently appear in well-designed software systems. Each pattern is presented with a description of the design problem the pattern addresses, as well as sample implementation code and design considerations. This paper explores how the patterns from the "Gang of Four", or "GOF" book, as it is often called, appear when similar problems are addressed using a dynamic, higher-order, object-oriented programming language. Some of the patterns disappear – that is, they are supported directly by language features, some patterns are simpler or have a different focus, and some are essentially unchanged.

1 Introduction

In [Nor96], Peter Norvig describes design patterns as:

- Descriptions of what experienced designers know (that isn't written down in the Language Manual)

- Hints/reminders for choosing classes and methods
- Higher-order abstractions for program organization
- To discuss, weigh and record design tradeoffs
- To avoid limitations of implementation language.

Norvig goes on to classify the levels of implementation that a pattern may have:

- **Invisible:** So much a part of language that you don't notice (e.g. when `class` replaced all uses of `struct` in C++, no more "Encapsulated Class" pattern)
- **Informal:** Design pattern in prose; refer to by name, but must be implemented from scratch for each use
- **Formal:** Implement pattern itself within the language. Instantiate/call it for each use. Usually implemented with macros.

This paper focuses on language features that can move design patterns from the Informal realm into the Invisible or Formal realms.

Norvig's summary of implementing the GOF patterns in Dylan or Lisp:

> 16 of 23 patterns are either invisible or simpler, due to:

- **First-class types** (6): Abstract Factory, Flyweight, Factory Method, State, Proxy, Chain of Responsibility.
- **First-class functions** (4): Command, Strategy, Template Method, Visitor.
- **Macros** (2): Interpreter, Iterator.
- **Method combination** (2): Mediator, Observer.
- **Multimethods** (1): Builder.
- **Modules** (1): Facade.

Peter Norvig's slides give no more information about how he arrived at these numbers. Part of what this paper does is to delve more deeply into the implications of implementing the 23 Design Patterns using a dynamic object-oriented programming language.

1.1 Reflection

Design Patterns are concerned primarily with the coordination of multiple "parts" of a program – typically instantiated as classes and (virtual) functions. Note how this overlaps with the "crosscutting concerns" of aspect-oriented programming.

Reflection is used to explicitly manipulate program components – sometimes defined in terms of the state of a language interpreter. It is at that level – the level of an interpreter – that coordination between program elements (classes, methods) can be directly implemented.

1.1.1 Reflection via Expressed Values ("Firstclassedness")

Reflection refers to the ability of a program to reason about its own structure and behavior. To enable reflection, the elements that compose a program must themselves be manipulable – that is, they must be "expressed values" in the host programming language. In other words, program elements, such as methods and classes, must be "first class" – able to be bound to variables and passed as arguments. If having program elements as first class values is one requirement for reflection, then all higher order programming languages are reflective in this sense.

The sorts of elements that make up typical programs include:

1. **Virtual Functions**, also known as *generic functions*.
2. **Methods** the elements of virtual functions. Methods contain the actual code to be run, given arguments of the appropriate types.
3. **Lines of code** Static – doesn't seem like a good idea to expose source code via reflection.
4. **Types** Used for specifying the correctness of code, and also for *dynamic dispatch* – the selection of applicable methods based on the types of arguments in a function call. Types are often related by a *subtyping* relation, which typically must satisfy the "substitutivity" requirement – if $t_1 \leq t_2$ (t_1 is a subtype of t_2), then any object that is of type t_1 must be usable in a context in which a t_2 is usable.
5. **Units of Encapsulation** such as **modules** or **classes**. In many languages, the set of classes either corresponds to, or is a subset of, the set of types. The *inheritance* relation between classes establishes the subtyping relation between corresponding types and also specifies that subclasses contain all the state elements (aka "fields", "slots", or "attributes") of any ancestor classes. Furthermore, in some languages, a class is a unit of encapsulation, defining what operations are visible to other program entities.

It is not at all clear what the operations on these values should be – such decisions can have huge implementation and performance repercussions; not to mention issues of safety, correctness, extensibility, and understandability. For example, can we expose and/or manipulate the code associated with a method object? Can we change the supertypes of a type? Different languages answer these questions differently; most mainstream languages answer uniformly with "no".

1.1.2 Reflection via Language Internals

Historically, reflection is sometimes presented in terms of a program being able to access the state of the interpreter that is currently executing the program [SMI84]. Of course the interpreter is being interpreted by an interpreter, and so on, thus leading to the "infinite tower of interpreters". This is a fun and interesting viewpoint, but not of direct practical importance. More practical is the idea of metaobject protocols (MOPs), which give an API (Applications Programming Interface) to the workings and state of any interpreter for a given programming language. Typical dynamic state exposed by a MOP includes:

1. **Bindings / Environment:** See Python.
2. **Dispatch:** Either overriding default method dispatch methods, or insinuating user code into the dispatch protocol (method combination).
3. **Continuations, Stack.**

2 GLOS (Greg's Little Object System)

The example code in this paper is given in Scheme plus a library of macros and functions that provide some object-oriented programming facilities. The features provided by GLOS (Greg's Little Object System) are intended to be representative of those provided by more full-featured dynamic object-oriented languages such as Dylan and CLOS. This section gives a quick overview of the features provided by GLOS beyond what is in the base Scheme programming language.

2.1 GLOS Types

The primary use of types in Scheme+GLOS is to specialize the allowable types of arguments to *methods*, which are described in the next section. Another place that types are used is in the "typed let" construct, introduced with the keyword `tlet`. Each binding in a `tlet` is either a pair `(var exp)` as in Scheme's `let` or a triple `(var type-exp exp)` where `type-exp` must evaluate to a

GLOS type. If a type constraint is given, the value to which `exp` evaluates is dynamically checked to satisfy that type, at binding time.

Predicates as Types: In general, boolean-returning, single-argument functions can be used as type specializers. For example, to indicate that the variable `n` must be bound to an even integer, the variable `name` must be bound to a string, and the variable `x` can be bound to any value, we would write:

```
(tlet ((n even? (foo y))
       (name string? (bar y))
       (x (baz y)))
  ... )
```

We test type membership using `isa?`:

```
(isa? 42 integer?)    ;; => true
(isa? 42 string?)     ;; => false
```

Type Conjunction: "and types" are constructed with the `and?` constructor:

```
(define <number> number?)
(define <complex> (and? <number> complex?))
(define <real> (and? <complex> real?))
(define <rational> (and? <real> rational?))
(define <integer> (and? <rational> integer?))
```

Type Disjunction: Likewise, we can use the `or?` constructor to create type disjunctions:

```
(define <callable> (or? <generic> <method> <procedure>))
```

Subtyping: There is a subtyping relation between types. In the case of type conjunction and disjunction, subtyping corresponds to logical implication. We use the predicate `subtype?` to query that relation:

```
(subtype? <integer> <number>)     ;; => true
(subtype? <generic> <callable>)   ;; => true
```

Every type is a subtype of <*top*>, and the type <*bottom*> is a subtype of every other type.

Singleton Types: A type that matches exactly one value is constructed with the `==` constructor:

```
(define l1 '(1 2 3))
(define <is-l1> (== l1))
(isa? '(1 2 3) <is-l1>)   ;; => false
(isa? l1 <is-l1>)   ;; => true
```

Record Types: A record type describes allocated objects with 0 or more named fields. The definition of a record type includes a list of *supertypes*, which serves the following two purposes:

1. Fields are inherited from supertypes, thus saving the programmer from having to respecify the inherited fields.

2. The *accessor* generic functions, that get and set inherited field contents, are updated so that they can handle instances of the new subtype as well as instances of the inherited types.

The `defrectype` macro defines new GLOS record types, and has the form

```
(defrectype typename (supertype ...)
    (fieldspec ...)
    accessor-spec ...)
```

where a `fieldspec` looks like either `(fieldname type)` or `(fieldname type initial-value-expression)` and an `accessor-spec` looks like either `(fieldname getter-name setter-name)` or `(fieldname getter-name)`. Note that the `initial-value-expression` is embedded in a thunk which is evaluated for every new instance.

The following definitions declare record types <*point*> and <*colorpoint*>:

```
(defrectype <point> ()
   ((x <integer>)
    (y <integer>))
   (x get-x set-x!)
   (y get-y set-y!))
(defrectype <colorpoint> (<point>)
   ((color <string> "black"))
   (color get-color set-color!))
```

The <*point*> type has fields named `x` and `y`, both of which are constrained to hold integer values. A total of five names are bound in the top-level environment as a result of the (**defrectype** <*point*> ...) definition: <*point*>, get-x, set-x!, get-y, and set-y!. <*point*> is bound to the new record type. The name `get-x`, for example, is bound to the generic function used to get the value of the `x` field from instances of <*point*>. Note that any preexisting definitions of <*point*>, get-x, etc. are overridden by these new definitions. There is no way in Scheme to check for this situation.

The <*colorpoint*> type inherits fields `x` and `y` from <*point*>, and adds a new field, `color`. The `color` field has an initial value, `"black"`. If we examine the generic functions `get-x, get-y, etc.` we will find that they have two methods each in them – one for instances of <*point*> and one for instances of <*colorpoint*>.

Record Instantiation: To create a new instance of a record type, we can call the constructor thunk for the type:

```
(define p1 ((glos-record-type-constructor <point>)))
```

which returns an instance with fields initialized either to their initial value, if given, or to `*undefined*`, an instance of <*bottom*>, if no initial value was specified.

For the purposes of this paper, we have a simple instantiation protocol, accessed by calling the function `new` with the record type and any initialization arguments.

```
(define cp1 (new <colorpoint> 'x 2 'y 3)) ;; x=2, y=3, color = "black" (default)
```

The `new` function first calls the `make` generic function with the record type and any arguments sent to `new`, and then calls the `initialize` generic function with the new instance and also with the arguments sent to `new`. The default method on `make` ignores all arguments but its first (the record type) and calls the record type's constructor function (thunk), as described above. The default method on `initialize` knows how to handle a list of alternating field names and values, and calls the utility function `set-by-name*` with this list.

The `new` function is an instance of the Factory Method and Template Method patterns.

Record types can be recursive:

```
(defrectype <mypair> ()
   ((head)
    (tail (false-or <mypair>)))
   (head head set-head!)
   (tail tail set-tail!))
(gfmethod (initialize (obj <mypair>) x (y (false-or <mypair>)))
    (set-by-name* obj 'x x 'y y))
(define p1 (new <mypair> 'e1 (new <mypair> 'e2 (new <mypair> 'e3 false))))
(head (tail p1))   ;; => 'e2
```

The method on `initialize`, above, allows creation of <*mypair*> objects without having to send field names `'head` and `'tail` to `new`.

2.2 Methods and Generic Functions

Methods: A method is a Scheme procedure with an argument signature and a return type. An argument signature is a type that must be satisfied by the list of argument values from a call of this method.

Typically the argument signature type is a **signature type**. A signature type is a list of required argument types and a *rest type*. If a method `m1` has an argument signature type of `((t1 t2 t3) rest-type)`, then every call of `m1` must supply at least three arguments, and the first three arguments must satisfy types `t1`, `t2`, `t3` respectively. If more than three values are supplied as arguments to a call to `m1`, each argument after the third one must satisfy `rest-type`. If `rest-type` is `#f`, exactly three arguments are required for a call to `m1`. The `method` macro, and its name-binding cousin `defmethod`, allow specification of argument specializers and rest types as follows:

```
(define move (method ((p <point>) (x <integer>) (y <integer>))
                     (set-x! p x) (set-y! p y)))
(defmethod (sum (init-val <integer>) :rest (other-vals <integer>))
   (apply + init-val other-vals))
(sum 1 2 3)  ;; => 6
```

Generic Functions: Also known as **virtual functions**, a generic function is a collection of methods and a *composer* function. A composer function takes the generic function being called, a list of applicable methods, and the list of argument values, and it returns a *call context*. When a generic function is called, it performs four actions:

1. A list of applicable methods is selected by calling `isa?` on each method signature and the argument value list.
2. A call context is created by calling the generic function's composer function, and
3. The dynamic variable `*call-context*` is bound to the new context, and
4. The *executor* function of the context is invoked.

A **call context** describes an active generic function invocation, including the generic function involved, the chain of methods to be executed, a list of applicable "next" methods, the current method being executed, the argument values, and the "executor" function. The executor function is a thunk that calls each of the chosen methods in turn.

The *composer* functions determine what method or methods of the applicable methods will be called, and, if more than one, in what order. For example, the `primary-composer` function selects a single most applicable method, produces a one-element method chain, and produces an executor thunk that will call that most applicable method.

Here is *factorial*, implemented using a generic function:

```
(defgeneric fact
  (method ((n (== 1)))
          1)
  (method ((n <int>))
          (* n (fact (- n 1)))))
```

Method Combination: In CLOS, *method combination* refers to the ability to add methods *before*, *after*, and *around* the primary methods. GLOS supports similar functionality, by using different composer functions. For example:

```
(defgeneric move
  (method ((p <point>) (x <integer>) (y <integer>))
          (set-x! p x)
          (set-y! p y)))
(add-before-method
 move (method ((p <colorpoint>) (x <integer>) (y <integer>))
              (format true "before move colorpoint (~a, ~a)~%" x y)))
(add-after-method
 move (method ((p <point>) (x <integer>) (y <integer>))
              (format true "after move point (~a, ~a)~%" x y)))
(move p1 2 3)    ;; "after move point"
(move cp1 2 3)   ;; "before move colorpoint" "after move point"
```

If we explore the `move` generic function after adding the before and after methods to it, we would find that the generic consists of three methods - one with the specializer <*before*>, one with <*primary*>, and one with <*after*>. These specializers are always applicable (arguments always satisfy them), and they are ordered <*before*> < <*primary*> < <*after*>. The composer function for modified generics simply calls each applicable method, in order. The <*before*> method has as its callable part not a Scheme procedure but another generic function (!). This generic function has a `before-composer` composer function, which calls each applicable before method, from least specific to most specific. The <*primary*> method has as its callable a generic function with the usual `primary-composer` function, and the <*after*> method has as its callable a generic function with an `after-composer` function that invokes each applicable after method from most specific to least specific.

There is another method combination, *around*, which supersedes before and after methods. Adding an around method to a generic adds a third level to the generic function. After adding around methods to a generic, the least specific method implements the behavior of the generic before any around methods were added. The `around-composer` is identical to the `primary-composer` in that it chooses a single most applicable method and calls it. In order to continue with the rest of the generic function's methods, the program must invoke `call-next-method`.

`call-next-method` accesses the call context established when the generic function was invoked and recomposes new context values starting with the rest of the chain and the list of next methods.

```
(defgeneric g1
  (method ((i <integer>))
    (format #t "g1$<\!\!\itbox{integer}\!\!>$(~a)~%" i) i)
  (method ((n <number>))
    (format #t "g1$<\!\!\itbox{number}\!\!>$(~a)~%" n) n)
  (method ((s string?))
    (format #t "g1string?(~a)~%" s) s))
(add-around-method g1
  (method ((x <int>))
      (format #t "around $<\!\!\itbox{int}\!\!>$~%")
      (call-next-method)))
(add-around-method g1
  (method ((x <string>))
      (format #t "around $<\!\!\itbox{string}\!\!>$~%")
      'foo))
(g1 2)     ;; => "around <int> ... g1<integer>" ... 2
(g1 "hi")  ;; => "around <string> 'foo
```

3 The GOF Design Patterns

As a study of how the basic capabilities of reflection and dynamism impact the need for, use of, and implementation of design patterns, we will survey the "GOF" patterns from [GHJV94]. Each pattern will be introduced with the Intent as given in [GHJV94].

For most of the design patterns, we present much of the code from the Sample Code section of the pattern's chapter in the GOF book alongside code in Scheme+GLOS that provides similar functionality. One caveat: Scheme+GLOS has no namespace management system, and some of the brevity of the Scheme+GLOS code with respect to C++ code is related to C++ having separate sections in class definitions for private, protected, and public class members. If Scheme+GLOS had better namespace management, the Scheme code would be slightly longer than it is in this paper, in order to take advantage of the additional functionality.

3.1 Abstract Factory

> **Intent:** *Provide an interface for creating families of related or dependent objects without specifying their concrete classes.*

The sample code for Abstract Factory in the GOF revolves around creating mazes. The `MazeFactory` class is in charge of creating families of consistent maze components and thus has methods named `makeMaze`, `makeWall`, etc. Subclasses of `MazeFactory` override the default methods.

As discussed in the GOF book, a language such as Smalltalk, with first class classes, can implement an Abstract Factory as a dictionary with keys as maze elements (`#room`, `#door`) and with classes as table data. Embedding lookup by names in the execution makes optimization more difficult. Using dynamic multiple dispatch, first class classes, and singleton types, we can use the builtin multiple dispatch of GLOS. In the following, Smalltalk code from the GOF is on the left, Scheme code is on the right.

```
createMazeFactory
    ^ (MazeFactory new
        addPart: Wall named: #wall;
        addPart: Room named: #room;
        addPart: Door named: #door;
        yourself)

make: partName
    ^ (partCatalog at: partName) new

createMaze: aFactory
    | room1 room2 aDoor |
    room1 := (aFactory make: #room) number: 1.
    room2 := (aFactory make: #room) number: 2.
    aDoor := (aFactory make: #door)
             from: room1 to: room2.
    ...
```

```
(defgeneric make-maze-element
  (method ((f <maze-factory>)
           (eltType (== <wall>))) => <wall>
      (new <wall>))
  (method ((f <maze-factory>)
           (eltType (== <room>)) :rest args)
          => <room>
      (apply new <room> args))
  (method ((f <maze-factory>)
           (eltType (== <door>)) :rest args)
          => <door>
      (apply new <door> args)))
(define the-factory (new <maze-factory>))
(define room1
    (make-maze-element the-factory <room>
                       'number 1))
(define room2
    (make-maze-element the-factory <room>
                       'number 2))
(define door1
    (make-maze-element the-factory <door>
                       'from room1 'to room2))
...
```

Using the system's method dispatch implementation rather than a user data structure such as a dictionary allows calls such as `(make-maze-element the-factory` *<room>* `2)` to be more easily optimized.

Summary: Reflection allows for all the variations of the pattern, with tradeoffs as discussed in the GOF.

3.2 Builder

> **Intent:** *Separate the construction of a complex object from its representation so that the same construction process can create different representations.*

The code in Scheme+GLOS is pretty similar to that given in the GOF, in C++. The `CreateMaze` protocol calls `buildMaze`, `buildRoom`, `buildDoor`, etc. Subclasses of `MazeBuilder` override `buildMaze`, `buildRoom`, etc.

```
class MazeBuilder {
public:
  virtual void BuildMaze() { }
  virtual void BuildRoom(int room) { }
  virtual void BuildDoor(int roomFrom,
                         int roomTo)
       { }
  virtual Maze* GetMaze() { return 0; }
protected:
  MazeBuilder();
};
Maze* MazeGame::CreateMaze
      (MazeBuilder\& builder) {
    builder.BuildMaze();
    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);
    return builder.GetMaze();
}
```

```
(defrectype <maze-builder> () ())
(defgeneric create-maze
  (method ((game <maze-game>)
           (builder <maze-builder>))
    (build-maze builder game)
    (build-room builder game 1)
    (build-room builder game 2)
    (build-door builder game 1 2)
    (get-maze builder)))
```

Suppose we want to play the same abstraction trick as in Abstract Factory – have one `buildElement` virtual function that takes a parameter that indicates what type of element you want to add. In a language with single dispatch, you have to use double dispatching to get to the right code, or, even worse, the code from the GOF (GOF, pg. 97) in their motivating example of a text conversion engine:

```
switch t.Type {
  CHAR:
    builder->convertCharacter(t.Char);
  FONT:
    builder->convertFontChange(t.Font);
  PARA:
    builder->convertParagraph();
```

With multiple dispatch, however, you can write code such as:

```
(add-method* convert
    (method ((builder <tex-converter>) (token <char>))
        ... convert TeX character ... )
    (method ((builder <tex-converter>) (token <font>))
        ... convert TeX font change ... )
    (method ((builder <text-widget-converter>) (token <char>))
        ... convert text widget character ... )
    (method ((builder <text-widget-converter>) (token <font>))
        ... convert text widget font change ... ))
```

The presentation of the Builder pattern is "class-centric" – there is a `Director` class (Converter or Game) with an instance variable `builder` that points to an instance of a `Builder` class. The Director's `construct` function calls the relevant methods of its builder object.

In a generic function setting, we do not need to have behavior "owned" by a class. We can write a top-level `construct` generic function that instantiates a `Builder` object and calls the relevant methods on that Builder. In other words, the Builder pattern involves a `Director` class that has behavior but no real state. In a generic function setting, we can do without such classes and define the behavior directly in terms of generic functions.

C++ allows functions to exist outside of classes, but those functions are not dynamically dispatched.

Summary: The idea of a construction protocol that returns the finished product when asked is universal. Multiple dispatch and first-class classes can be used to abstract the part construction protocol as in Abstract Factory. Finally, the Builder pattern is somewhat "class-centric", and in a setting where dynamically dispatched behavior can be specified independently of classes, we may be able to do without the Director class of the Builder pattern.

3.3 Factory Method

> **Intent:** *Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.*

Any language that lets the programmer override the default creation method provides direct support for the Factory Method pattern. Dylan [Sha97], for example, provides a two-level protocol for instance creation, and it is not unusual to override the default `make` method for a class `C` to return an instance of some subclass of `C`. Controlling instance creation is another instance of reflection – this time more related to metaobject protocols.

Typically, C++ code as from the GOF - on the left below, can be replaced by Scheme code as on the right below.

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();
    virtual Wall* MakeWall() const
        { return new BombedWall; }
    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

```
(gfmethod (make (c (== <wall>)))
    (make <bombed-wall>))
(gfmethod (make (c (== <room>)) (n <int>))
    (make <bombed-room>))
```

Note that the programmer does not need to remember to call a distinguished instantiation function (e.g. `makeWall`) but can instead use the standard instantiation interface (e.g. `new`).

Multiple dispatch again comes in useful - this time for implementing what the GOF call "parameterized Factory methods". On the left is code from the GOF, with Scheme code on the right:

```
Product* Creator::Create (ProductId id) {
   if (id == MINE) return new MyProduct;
   if (id == YOURS) return new YourProduct;
   // repeat for remaining products...
}
```

```
(add-method* make
   (method ((c (== <product>)) (id (== 'mine)))
           (make <my-product>))
   (method ((c (== <product>)) (id (== 'yours)))
           (make <your-product>)))
```

Summary: An extensible instance creation protocol directly supports the Factory Method pattern. Multiple dispatch is useful to further abstract the instance creation process.

3.4 Prototype

> **Intent:** *Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.*

As the GOF (pg. 121) point out,

> *Prototype is particularly useful with static languages like C++, where classes are not objects, and little or no type information is available at run-time. It's less important in languages like Smalltalk or Objective C that provide what amounts to a prototype (i.e., a class object) for creating instances of each class. This pattern is built into prototype-based languages like Self, in which all object creation happens by cloning a prototype.*

The use of Prototype to get around the lack of first-class classes is obviously addressed by basic reflection.

The Prototype pattern also requires that every object be "cloneable", which is an addition to what is provided by first-class classes. Smalltalk provides a standard `copy` method for this purpose. With sufficient reflection – that is, being able to query a class for its fields, and also being able to set fields abstractly (i.e. without hardwiring setter / field names into code) one can write a general purpose shallow `clone` method. For example, in GLOS we can write:

```
(defmethod (clone (obj <object>))
  (let* ((obj-type (instance-type obj))
         (new-obj ((glos-record-type-constructor obj-type))))
    (format true "created, now setting.~%")
    (for-each (lambda (field)
                ((field-spec-setter field) new-obj
                 ((field-spec-getter field) obj)))
              (glos-record-type-fields obj-type))
    new-obj))
```

Summary: The main purpose of Prototype is provided in-language by having first-class classes – a basic feature of reflection. Support for instance copying can be provided by the system, as in Smalltalk's `copy` method, or, with reflection, cloning can be a user-level library function. For more complex cloning (deep rather than shallow), either the language has to have a more sophisticated protocol for cloning, or one has to revert to implementing the cloning part of the Prototype design pattern.

3.5 Singleton

> **Intent:** *Ensure a class only has one instance, and provide a global point of access to it.*

Similar to the Factory Method pattern, if the language provides an extensible instance creation protocol, it is straightforward to implement the Singleton pattern so that coding conventions are not required. Code from the GOF, on the left, can be replaced by code in Scheme, on the right:

```
class MazeFactory {
public:
    static MazeFactory* Instance();
    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
...
MazeFactory* MazeFactory::_instance = 0;
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

```
(let ((the-instance *undefined*))
  (gfmethod (make (c (== <maze-factory>)))
      (if (undefined? the-instance)
          (set! the-instance
              ((glos-record-type-constructor
                  <maze-factory>))))
      the-instance))
```

Summary: An extensible instance creation protocol enables implementation of the Singleton pattern without having to introduce a new virtual method and user defined instantiation protocol. However, requiring the programmer to call a designated method such as `instance()` to get the singleton has its benefits – it makes it clear to the programmer that the usual instantiation protocol is not being used.

3.6 Adapter

> **Intent:** *Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.*

The Adapter pattern, simply put, translates messages from one protocol ("Target") to another ("Adaptee"). More generally, it allows the programmer to use functionality from one or more classes (Adaptees) to fulfill the interface requirements of a new class (Adapter) that is a subclass of Target.

The GOF point out that one may use multiple inheritance as a mechanism for automatically incorporating the adaptee into the adapter instance, but this is an implementation/efficiency hack.

The GOF also point out that with first-class functions, as in Smalltalk, it is possible to create "parameterized adapters", where the adaptation code is sent as functions to be run by the adapter.

This is also reminiscent of nested anonymous classes in Java.

The presentation of the Adapter pattern is class-centric, "how do we get this new class to support behavior which happens to be largely supported by this other class?"

The generic function model takes a different view: "how do we extend this functionality to work on elements of this new class?" The example from the GOF is adding a `TextShape` subclass to `Shape`, where `TextShape` implements the required `BoundingBox` behavior by adapting (delegating to) a corresponding `TextView` object.

In the following, the `TextShape` class has an instance variable `TextView* _text` initialized by the constructor.

```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;
    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);
    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height,
                     left + width);
}
bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

```
(gfmethod (bounding-box (t <text-shape>)) => <pair>
    (let* ((the-view (_text t))
           (bottom (origin-bottom the-view))
           (left (origin-left the-view)))
      (pair (new <point> bottom left)
            (new <point>
                 (+ bottom (extent-height the-view))
                 (+ left (extent-width the-view)))))))
```

Summary: In the generic function model, the Adapter pattern has a somewhat different flavor than in the class-centric model. This is because adding methods to a generic function can be viewed as extending functionality (such as `boundingBox`) to cover new domains (such as `TextShape`).

As discussed in the GOF, first-class functions also make the adapter pattern more abstract.

Note that sophisticated module systems may mitigate the need for the Adapter pattern, allowing at the very least for renaming of interface elements upon import.

3.7 Bridge

> **Intent:** *Decouple an abstraction from its implementation so that the two can vary independently.*

The basic pattern of programming in terms of a family of Abstraction classes, and operations on them, and having a clean interface to one or more Implementor, device-specific classes, seems universal.

3.8 Composite

> **Intent:** *Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.*

The Composite pattern is basically the extension of recursive sum and product types to the object-oriented regime. Since both C++ and Scheme + GLOS support recursive class (resp. record) types, the implementations are similar (although the Scheme version is somewhat less verbose):

```
class Equipment {
public:
    virtual ~Equipment();
    const char* Name() { return _name; }
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

```
(defrectype <equipment> ()
  ((name <symbol>))
  (name equipment-name set-equipment-name!))
(defgeneric power)
(defgeneric net-price)
...
(defrectype <floppy> (<equipment>) ())
```

```
class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator* CreateIterator();
protected:
    CompositeEquipment(const char*);
private:
    List _equipment;
};
Currency CompositeEquipment::NetPrice () {
    Iterator* i = CreateIterator();
    Currency total = 0;
    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

```
(defrectype <composite-equipment> (<equipment>)
  ((parts <list>))
  (parts composite-parts set-composite-parts!))
(gfmethod (net-price (e <composite-equipment>))
    (fold (lambda (item total)
            (+ total (net-price item)))
          0 (composite-parts e)))
(defrectype <chassis> (<composite-equipment>) ())
```

Note the use of the higher-order `fold` function in the Scheme code.

Summary: Because recursive datatypes are supported in both C++ and Scheme+GLOS, implementations of the Composite pattern are similar in both languages.

3.9 Decorator

Intent: *Attach additional responsibilities to an object dynamically. Decorators provide*

a flexible alternative to subclassing for extending functionality.

The idea of method combination in Common Lisp provides direct support for idiomatic behavior decoration. There are similar ideas in AspectJ, with before and after "advice". The AspectJ advices are not first class, and cannot be added or removed dynamically (at runtime). Method combination allows one to directly express the idea that you want to do one bit of behavior before (or after, or instead of) doing the primary behavior, and combinations can be combined.

```
class VisualComponent {
public:
    VisualComponent();
    virtual void Draw();
};
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);
    virtual void Draw();
private:
    VisualComponent* _component;
};
void Decorator::Draw () {
    _component->Draw();
}
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*,
                    int borderWidth);
    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
};
void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
Window* window = new Window;
TextView* textView = new TextView;
window->SetContents(textView);
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

```
(defrectype <visual-component> () ())
(defgeneric draw
  (method ((comp <visual-component>))
          (format true "drawing visual-component~%"))
  (method ((w <window>))
          (draw (window-contents w)))
(defmethod (decorate (component <visual-component>)
                     (decoration <visual-component>))
  (add-after-method draw
                    (method ((c (== component)))
                            (draw decoration))))
(defrectype <border-decorator> (<visual-component>)
  ((width <int>))
  (width border-width set-border-width!))
(gfmethod (draw (comp <border-decorator>))
          (draw-border (border-width comp)))
(define tv1 (new <text-view>))
(define w1 (new <window> 'contents tv1))
(decorate tv1 (new <scroll-decorator>))
(decorate tv1 (new <border-decorator> 'width 4))
```

In the Scheme code above, we do not need a new `Decorator` class, because we use the built-in method combination capability. The `decorate` method makes the Decorator pattern dynamically instantiable.

Summary: Method combination provides direct support for a common case of decoration. Prototype-based languages also provide direct support for decoration. Specializing methods on singletons allows behavior to be extended (specialized) on a per-object basis.

3.10 Facade

> **Intent:** *Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.*

By and large, this is simply good coding practice, use of abstraction, and namespace management.

Since namespace management is not considered in this paper, we won't discuss the Facade pattern any further.

3.11 Flyweight

> **Intent:** *Use sharing to support large numbers of fine-grained objects efficiently.*

Hell, the Lisp community practically invented this pattern, except it was given obscure names like "hash consing" and "interning", which is why Lisp never caught on, but that's another story...

More seriously, having access to the object creation protocol is important for implementing Flyweight, in much the same way as for the Factory Method and Singleton patterns. As for the Factory Method and Singleton patterns, the programmer can use the standard object creation process (in our case, `new`) to get instances, and the fact that these objects are shared is invisible. The sample GOF code for the factory component of the Flyweight pattern is on the left below, with correpsonding Scheme code on the right:

```
const int NCHARCODES = 128;
class GlyphFactory {
public:
    GlyphFactory();
    virtual ~GlyphFactory();

    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
    // ...
private:
    Character* _character[NCHARCODES];
};
GlyphFactory::GlyphFactory () {
    for (int i = 0; i < NCHARCODES; ++i) {
        _character[i] = 0;
    }
}
Character* GlyphFactory::CreateCharacter
            (char c) {
    if (!_character[c]) {
        _character[c] = new Character(c);
    }
    return _character[c];
}
```

```
(defrectype <character> ()
  ((char <char>))
  (char get-char set-char!))
(gfmethod (initialize (obj <character>) (char <char>))
          (set-char! obj char))
(let ((char-table (make-integer-table)))
  (gfmethod
   (make (class (== <character>)) (char <char>))
   (let ((char-int (char->integer char)))
     (cond ((table-ref char-table char-int)
            => identity)
           (else
            (let ((new-char (call-next-method)))
              (table-set! char-table char-int new-char)
              new-char))))))
```

Summary: As in some of the creational patterns, having access to the instantiation protocol enables more direct and dynamic implementation of the Flyweight pattern.

3.12 Proxy

Intent: *Provide a surrogate or placeholder for another object to control access to it.*

The GOF list the following kinds of Proxy uses.

[TO DO: replace quoted references with real ones:]

1. A **remote proxy** provides a local representative for an object in a different address space. NEXTSTEP [Add94] uses the class NXProxy for this purpose. Coplien [Cop92] calls this kind of proxy an "Ambassador."

2. A **virtual proxy** creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.

3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system [CIRM93] provide protected access to operating system objects.

4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include

   - counting the number of references to the real object so that it can be freed automatically when there are no more references (also called smart pointers [Ede92]).
   - loading a persistent object into memory when it's first referenced.
   - checking that the real object is locked before it's accessed to ensure that no other object can change it.

We can insinuate a proxy into the instantiation protocol, as the following code illustrates:

```
class Graphic {
public:
     virtual void Draw(const Point& at) = 0;
     virtual void Load(istream& from) = 0;
     // ...
};
class Image : public Graphic {
public:
  Image(const char* file); // loads from file
  virtual void Draw(const Point& at);
  virtual void Load(istream& from);
private:
    // ...
};
class ImageProxy : public Graphic {
public:
     ImageProxy(const char* imageFile);
     virtual void Draw(const Point& at);
     virtual void Load(istream& from);
protected:
     Image* GetImage();
private:
     Image* _image;
     char* _fileName;
};
ImageProxy::ImageProxy (const char* fileName) {
     _fileName = strdup(fileName);
     _image = 0;
}
Image* ImageProxy::GetImage() {
     if (_image == 0) {
          _image = new Image(_fileName);
     }
     return _image;
}
void ImageProxy::Draw (const Point& at) {
     GetImage()->Draw(at);
}
void ImageProxy::Load (istream& from) {
     from >> _extent >> _fileName;
}
class TextDocument {
public:
     TextDocument();
     void Insert(Graphic*);
     // ...
};
TextDocument* text = new TextDocument;
text->Insert(new ImageProxy("anImageFileName"));
}
```

```
(defrectype <graphic> () ())
(defrectype <image> (<graphic>)
   ((bits))
   (bits image-bits set-image-bits!))
(gfmethod (initialize (obj <image>) (fn <string>))
          (set-image-bits! obj (bits-from-file fn)))
(defgeneric draw
   (method ((i <image>))
      (let ((bits (image-bits i)))
        (format true "Drawing image bits ~a~%" bits))))
(defrectype <image-proxy> (<image>)
   ((filename <string>)
    (empty? <boolean> true))
   (filename image-proxy-filename
             set-image-proxy-filename!)
   (empty? image-proxy-empty? set-image-proxy-empty?!))
;; hijack instantiation of <image> superclass
(gfmethod (make (class (== <image>)) :rest args)
          (make <image-proxy>))
(gfmethod (initialize (obj <image-proxy>) (fn <string>))
          (set-image-proxy-filename! obj fn))
(add-before-method
 image-bits
 (method ((obj <image-proxy>))
    (if (image-proxy-empty? obj)
       (begin
          (set-image-bits! obj
            (bits-from-file (image-proxy-filename obj)))
          (set-image-proxy-empty?! obj false)))))
(defrectype <text-document> ()
   ((elts <list> '()))
   (elts text-document-elements
         set-text-document-elements!))
(define td1 (new <text-document>))
(insert td1 (new <image> "image.gif"))
```

The hijacking of the instantiation of <*image*>s above is similar to that of the Scheme version of the Factory Method pattern, and the C++ code would benefit from a similar use of the Factory Method pattern. We chose to subclass <*image*> with the proxy type to avoid having to shadow every method of <*image*>.

With decent namespace management, a more straightforward approach is to define, in the scope where images are to be proxied, the <*image*> class as the <*image-proxy*> class is defined above. Then import the external <*image*> class, private to the scope of the proxied <*image*> class, and directly call the imported, but hidden constructor and other methods.

Note that a protection proxy can be implemented using method combination, without the need for a separate proxy object. Using *around* methods, we can check constraints before executing the primary method.

The GOF remark:

> These operations are usually very similar to each other, as the Sample Code demonstrates. Typically all operations verify that the request is legal, that the original object exists, etc., before forwarding the request to the subject. It's tedious to write this code again and again. So it's common to use a preprocessor to generate it automatically.

Thus, macros are also a useful avenue for implementing the proxy pattern. For example, one could extend the `defrectype` macro to allow a `lazy` keyword on the field specification that would cause the field to be memoized.

Lazy imperative languages implement virtual proxies directly.

Summary: The features of method combination, macros, singleton types, and an extensible instantiation protocol can all be used to good effect in implementing the Proxy pattern.

3.13 Chain of Responsibility

> **Intent:** *Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.*

If the chain of responsibility is structural, based on a pointed-to-by relation, then either reflection capabilities provide pointed-to-by information or the application has to maintain it in much the same way as in the GOF book.

The following are C++ and Scheme implementations of the simple Chain of Responsibility example in the GOF.

```
typedef int Topic;
const Topic NO_HELP_TOPIC = -1;
class HelpHandler {
public:
    HelpHandler(HelpHandler* = 0,
            Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};
HelpHandler::HelpHandler (
    HelpHandler* h, Topic t
) : _successor(h), _topic(t) { }
bool HelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}
void HelpHandler::HandleHelp () {
    if (_successor != 0) {
        _successor->HandleHelp();
    }
}
class Widget : public HelpHandler {
protected:
    Widget(Widget* parent,
           Topic t = NO_HELP_TOPIC);
private:
    Widget* _parent;
};
Widget::Widget (Widget* w, Topic t)
             : HelpHandler(w, t) {
    _parent = w;
}
class Button : public Widget {
public:
    Button(Widget* d, Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();
    // Widget operations that Button overrides...
};
Button::Button (Widget* h, Topic t)
              : Widget(h, t) { }
void Button::HandleHelp () {
    if (HasHelp()) {
        // offer help on the button
    } else {
        HelpHandler::HandleHelp();
    }
}
```

```
(defrectype <topic> () ((text))
  (text topic-text set-topic-text!))
(gfmethod (initialize (obj <topic>) text)
          (set-topic-text! obj text))
(defgeneric display-topic
  (method ((t <topic>))
          (format true "Topic: ~a~%" (topic-text t))))
(defrectype <handler> ()
  ((topic (false-or <topic>) false)
   (next (false-or <handler>) false))
  (topic handler-topic set-handler-topic!)
  (next handler-next set-handler-next!))
(defgeneric handle-help
  (method ((handler <handler>))
          (cond ((handler-topic handler)
                 => display-topic)
                ((handler-next handler)
                 => handle-help)
                (else
                 (format true "No topic found.~%")))))
(defrectype <widget> (<handler>)
  ((parent (false-or <widget>) false))
  (parent widget-parent set-widget-parent!))
;; handle new widget with only a parent handler,
;; not a parent widget:
(gfmethod (initialize (newobj <widget>)
                      (parent-handler (false-or <handler>))
                      (topic (false-or <topic>)))
      (set-handler-next! newobj parent-handler)
      (set-handler-topic! newobj topic))
(gfmethod (initialize (newobj <widget>)
                      (parent-widget (false-or <widget>))
                      (topic (false-or <topic>)))
      (set-widget-parent! newobj parent-widget)
      (set-handler-next! newobj parent-widget)
      (set-handler-topic! newobj topic))
(defrectype <button> (<widget>) () )
```

Continued...

```
class Dialog : public Widget {
public:
    Dialog(HelpHandler* h,
           Topic t = NO_HELP_TOPIC);
    virtual void HandleHelp();
    // Widget operations that Dialog overrides...
};
Dialog::Dialog (HelpHandler* h, Topic t)
               : Widget(0) {
    SetHandler(h, t);
}
void Dialog::HandleHelp () {
    if (HasHelp()) {
        // offer help on the dialog
    } else {
        HelpHandler::HandleHelp();
    }
}
class Application : public HelpHandler {
public:
    Application(Topic t) : HelpHandler(0, t) { }
    virtual void HandleHelp();
    // application-specific operations...
};
void Application::HandleHelp () {
    // show a list of help topics
}
const Topic PRINT_TOPIC = 1;
const Topic PAPER_ORIENTATION_TOPIC = 2;
const Topic APPLICATION_TOPIC = 3;
Application* application =
        new Application(APPLICATION_TOPIC);
Dialog* dialog =
        new Dialog(application, PRINT_TOPIC);
Button* button =
        new Button(dialog,
                   PAPER_ORIENTATION_TOPIC);
button->HandleHelp();
```

```
(defrectype <dialog> (<widget>) () )
(defrectype <application> (<handler>) ())
(gfmethod (initialize
                (obj <application>) (topic <topic>))
    (set-handler-topic! obj topic))
(gfmethod (handle-help (topic <topic>)
            (handler <application>))
     (format true "Displaying application topics.~%"))
(define printing-topic (new <topic> "help for printing"))
(define paper-orientation-topic
    (new <topic> "help for paper orientation"))
(define application-topic
    (new <topic> "help for application"))
(define application
     (new <application> application-topic))
(define dialog
     (new <dialog> application printing-topic))
(define button
     (new <button> dialog paper-orientation-topic))
(handle-help button)
```

For such a simple setup, we do not get much benefit from advanced language features. With a more complex chain of responsibility, though, we can use some of these features. For example, the

GOF discuss the possibility of different handlers in the chain being able to handle different types of requests. An outline of the required C++ code is given:

```
void Handler::HandleRequest (Request* theRequest) {
      switch (theRequest->GetKind()) {
      case Help:
          // cast argument to appropriate type
          HandleHelp((HelpRequest*) theRequest);
          break;
      case Print:
          HandlePrint((PrintRequest*) theRequest);
          // ...
          break;
      default:
          // ...
          break;
      }
  }
class ExtendedHandler : public Handler {
    public:
        virtual void HandleRequest(Request* theRequest);
        // ...
    };
    void ExtendedHandler::HandleRequest (Request* theRequest) {
        switch (theRequest->GetKind()) {
        case Preview:
            // handle the Preview request
            break;
        default:
            // let Handler handle other requests
            Handler::HandleRequest(theRequest);
        }
    }
```

This can be handled much more elegantly by setting up a chain of multiply-dispatched generic functions. In the following, a <*handler*> maintains both a pointer to its successor (`next`) and a generic function `local` that contains methods specialized by request type. The default method in the local generic function, which is invoked if there are not other applicable methods, delegates the request to its successor handler, if there is one.

```
(defrectype <handler> ()
   ((next (false-or <handler>))
    (local generic?))
   (next handler-next set-handler-next!)
   (local handler-local set-handler-local!))
(defgeneric handle-request
   (method ((request <request>) (handler <handler>))
           ((handler-local handler) request)))
(gfmethod (initialize (this-handler <handler>) :rest args)
           ;; install local handler generic
           (set-handler-local!
            this-handler
            (make-generic
             ;; default method defers to next handler in chain
             (method ((request <request>))
                     (if (handler-next this-handler)
                         (handle-request request
                                         (handler-next this-handler))
                         (format true "No handler found.~%")))))
           ;; get any local handler methods from initialization args
           (handle-key 'handler args
                     (lambda (handler-method)
                       (add-method (handler-local this-handler)
                                   handler-method))))
...
(define d1 (new <dialog> app1
                  'handler (method ((req <help-request>))
                                   (format true "dialog help~%"))
                  'handler (method ((req <preview-request>))
                                   (format true "dialog preview~%"))))
(define b1 (new <button> d1
                  'handler (method ((req <help-request>))
                                   (format true "button help~%"))))
```

Summary: In the example above, we make use of first class dynamically dispatched functions. Note that each of the generic functions is, in that example, only singly dispatched.

3.14 Command

> **Intent:** *Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.*

Simple Commands are directly supported in languages with closures. Below, some of the sample

code from the GOF is trivially implemented in Scheme.

```
class Command {
public:
    virtual ~Command();
    virtual void Execute() = 0;
protected:
    Command();
};
class OpenCommand : public Command {
public:
    OpenCommand(Application*);
    virtual void Execute();
protected:
    virtual const char* AskUser();
private:
    Application* _application;
    char* _response;
};
OpenCommand::OpenCommand (Application* a) {
    _application = a;
}
void OpenCommand::Execute () {
    const char* name = AskUser();
    if (name != 0) {
        Document* document = new Document(name);
        _application->Add(document);
        document->Open();
    }
}
// added by gregs:
someMenu->addMenuItem(
   "Open", new OpenCommand(theApp));
...
someMenuItem->Command()->Excecute();
```

```
(define (make-open-command app)
  (lambda ()
    (let ((name (ask-user)))
      (if name
          (let ((doc (new <document> name)))
            (add app doc)
            (open doc))))))
...
(add-menuitem some-menu
              (make-open-command the-app))
...
((menuitem-command some-menuitem))
```

If a command needs to support `unExecute`, it may have to be an actual object (user record type) rather than a native closure.

Summary: Closures directly support the `Execute` functionality of the Command pattern, without any undo functionality.

3.15 Interpreter

> **Intent:** *Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.*

To add support for the "Known Uses" subsection in the Interpreter chapter, we have the following quote by Phil Greenspun, promulgated by Paul Graham:

> Greenspun's Tenth Rule of Programming: *Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.*

As the discussion in the GOF indicates, Interpreter overlaps heavily with the Composite pattern and the more general Visitor pattern. Comments from those two design patterns cover the Interpreter pattern.

3.16 Iterator

> **Intent:** *Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.*

The GOF presentation is focused on "external" iterators, implemented in C++, and certainly every modern OO programming language comes with at least one standard iterator protocol for abstractly traversing collections.

Less attention is given by the GOF to "internal" iterators. For example, the GOF write:

> It's easy to compare two collections for equality with an external iterator, for example, but it's practically impossible with internal iterators.

However, in Scheme, we can simply do

```
(every eq? l1 l2)
```

First class functions make internal iterators much easier to use, so we present the GOF sample code for an internal iterator alongside comparable code in Scheme. The C++ code uses an external iterator class `ListIterator` whose definition is not included below.

```
template <class Item>
class FilteringListTraverser {
public:
    FilteringListTraverser(List<Item>* aList);
    bool Traverse();
protected:
    virtual bool ProcessItem(const Item&) = 0;
    virtual bool TestItem(const Item&) = 0;
private:
    ListIterator<Item> _iterator;
};
template <class Item>
void FilteringListTraverser<Item>::Traverse () {
    bool result = false;
    for (
        _iterator.First();
        !_iterator.IsDone();
        _iterator.Next()
    ) {
        if (TestItem(_iterator.CurrentItem())) {
            result = ProcessItem(_iterator.CurrentItem());
            if (result == false) {
                break;
            }
        }
    }
    return result;
}
// gregs added following
class HighlyPaidEmployees :
    public FilteringListTraverser<Employee*> {
public:
    HighlyPaidEmployees(List<Employee*>* aList, int n) :
        FilteringListTraverser<Employee*>(aList),
        _min(n) { }
protected:
    bool ProcessItem(Employee* const&);
    bool TestItem(const Employee&);
private:
    int _total;
    int _count;
};
bool HighlyPaidEmployees::ProcessItem
            (Employee* const& e) {
    _count++;
    e->Print();
}
bool HighlyPaidEmployees::TestItem
            (Employee* const& e) {
    return e->Salary() > _min
}

List<Employee*>* employees;
// ...
HighlyPaidEmployees pa(employees, 100000);
pa.Traverse();
```

```
(define employees (list ...))
(filter (lambda (e)
          (> (employee-salary e) 100000))
        employees)
```

** TO DO: ** cover iteration protocols of Dylan, as discussed in Norvig's talk. **

Summary: With first class functions, much more powerful "internal" iteration idioms are possible, making for more concise code. The work on polymorphic `fold` and `map` in imperative functional languages shows how powerful and general this approach can be.

3.17 Mediator

> **Intent:** *Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.*

The main complexity in the sample implementation of the Mediator pattern is the `WidgetChanged` method of the `FontDialogDirector` class. It turns out that the primary functions of the `Director` class can be handled by a generic function: (1) an object to be notified of events, and (2) a procedure for controlling interactions between objects.

```
class DialogDirector {
public:
    virtual void WidgetChanged(Widget*) = 0;
    // ...
    virtual void CreateWidgets() = 0;
};
class Widget {
public:
    Widget(DialogDirector*);
    virtual void Changed();
    virtual void HandleMouse(MouseEvent& event);
    // ...
private:
    DialogDirector* _director;
};
void Widget::Changed () {
    _director->WidgetChanged(this);
}
class ListBox : public Widget {
public:
    ListBox(DialogDirector*);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};
class EntryField : public Widget {
public:
    EntryField(DialogDirector*);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};
class Button : public Widget {
public:
    Button(DialogDirector*);
    virtual void HandleMouse(MouseEvent& event);
    // ...
};
void Button::HandleMouse (MouseEvent& event) {
    // ...
    Changed();
}
class FontDialogDirector : public DialogDirector {
public:
    // ...
    virtual void WidgetChanged(Widget*);
protected:
    virtual void CreateWidgets();
private:
    Button* _ok;
    Button* _cancel;
    ListBox* _fontList;
    EntryField* _fontName;
};
void FontDialogDirector::CreateWidgets () {
    _ok = new Button(this);
    _cancel = new Button(this);
    _fontList = new ListBox(this);
    _fontName = new EntryField(this);
    // fill the listBox with the available font names
    // assemble the widgets in the dialog
}
void FontDialogDirector::WidgetChanged (
    Widget* theChangedWidget
) {
    if (theChangedWidget == _fontList) {
        _fontName->SetText(_fontList->GetSelection());
    } else if (theChangedWidget == _ok) {
        // apply font change and dismiss dialog
        // ...
    } else if (theChangedWidget == _cancel) {
        // dismiss dialog
    }
}
```

```
(defgeneric handle-mouse)
(defrectype <widget> ()
  ((notify-fn generic? (make-generic)))
  (notify-fn widget-notify-fn
             set-widget-notify-fn!))
(gfmethod (handle-mouse (obj <widget>))
          ((widget-notify-fn obj) obj))
(defrectype <list-box> (<widget>)
  ((list))
  (list list-box-list set-list-box-list!))
(gfmethod (initialize (obj <list-box>) lst)
          (set-list-box-list! obj lst))
(defrectype <entry-field> (<widget>)
  ((text))
  (text field-text set-field-text!))
(gfmethod (initialize (obj <entry-field>) text)
          (set-field-text! obj text))
(defrectype <button> (<widget>)
  ((label))
  (label button-label set-button-label!))
(gfmethod (initialize (obj <button>) label)
          (set-button-label! obj label))
(define a-font-dialog
  (let ((ok (new <button> 'ok))
        (cancel (new <button> 'cancel))
        (font-list (new <list-box> '(1 2 3)))
        (font-name (new <entry-field> 'font-name-here))
        (notify-fn (make-generic)))
    (for-each
      (lambda (w)
        (set-widget-notify-fn! w notify-fn))
        (list ok cancel font-list font-name))
    (for-each
      (lambda (m) (add-method notify-fn m))
        (list
          (method ((obj (== font-list)))
            (set-text font-name
                      (get-selection font-list)))
          (method ((obj (== ok)))
            (format #t "ok button pressed~%"))
          (method ((obj (== cancel)))
            (format #t "cancel button pressed~%"))))
    ... display the dialog ... ))
```

Summary: First class dynamically dispatched functions can be used as the core event handling mechanism in the Mediator pattern.

3.18 Memento

> **Intent:** *Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.*

The GOF code has few details related to actually creating mementos, and is mainly concerned with correct use of the C++ protection keywords (`private`, `public`, etc.), so we will not give comparison code for the Memento pattern.

Logging (an important element of transactions) can be implemented by instrumenting record constructors and field mutators using method combination.

Also, as demonstrated in the Prototype pattern, sufficient reflection allows abstract copying of a record's state, which could be used to create mementos.

3.19 Observer

> **Intent:** *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*

In the GOF Sample Code for the Observer pattern, a subject keeps a list of observers. When the subject changes, it calls its `notify` function, which calls the `update` method of each observer.

In the Scheme version of the Observer pattern, the subject contains a generic function, `notify-fn`, to which observers can add methods. When an observer wants an action to be invoked when a subject changes, the observer adds a method to the update-fn of the subject. So, instead of having `attach` and `detach` methods on `Subject`, we use `add-method` and `remove-method`. Note that the `notify-fn` generic has a special composer function, `chain-composer`. `chain-composer` simply calls all applicable methods in the order encountered.

```
class Subject;
class Observer {
public:
     virtual void Update(Subject* theChangedSubject) = 0;
     // ...
};
class Subject {
public:
     virtual void Attach(Observer*);
     virtual void Detach(Observer*);
     virtual void Notify();
     // ...
private:
     List<Observer*> *_observers;
};
void Subject::Attach (Observer* o) {
     _observers->Append(o);
}
void Subject::Detach (Observer* o) {
     _observers->Remove(o);
}
void Subject::Notify () {
     ListIterator<Observer*> i(_observers);
     for (i.First(); !i.IsDone(); i.Next()) {
          i.CurrentItem()->Update(this);
     }
}
class ClockTimer : public Subject {
public:
     virtual int GetHour();
     // ...
     void Tick();
};
void ClockTimer::Tick () {
     // update internal time-keeping state ...
     Notify();
}
class DigitalClock: public Widget, public Observer {
public:
     DigitalClock(ClockTimer*);
     virtual ~DigitalClock();
     virtual void Update(Subject*);
          // overrides Observer operation
     virtual void Draw();
          // overrides Widget operation;
          // defines how to draw the digital clock
private:
     ClockTimer* _subject;
};
DigitalClock::DigitalClock (ClockTimer* s) {
     _subject = s;
     _subject->Attach(this);
}
DigitalClock:: DigitalClock () {
     _subject->Detach(this);
}
void DigitalClock::Update (Subject* theChangedSubject) {
     if (theChangedSubject == _subject) {
          Draw();
     }
}
void DigitalClock::Draw () {
     // get the new values from the subject
     int hour = _subject->GetHour();
     // etc.
     // draw the digital clock
}
class AnalogClock : public Widget, public Observer {
public:
     AnalogClock(ClockTimer*);
     virtual void Update(Subject*);
     virtual void Draw();
     // ...

};
```

```
(defrectype <subject> ()
  ((notify-fn generic?
              (really-make-generic 'update '()
                                   chain-composer false)))
  (notify-fn subject-notify-function
             set-subject-update-function!))
(defmethod (notify (s <subject>))
  ((subject-notify-function s)))
(defrectype <clock-timer> (<subject>)
  ((data))
  (data clock-timer-data set-clock-timer-data!))
;; we imagine that tick gets called by an internal timer
(defmethod (tick (timer <clock-timer>) time-data)
  (set-clock-timer-data! timer time-data)
  (notify timer))          ;; call the update generic
(defrectype <clock-widget> (<widget>)
  ((the-timer <clock-timer>))
  (the-timer clock-timer set-clock-timer!))
;; New clock-widgets register themselves with their timer.
(gfmethod (initialize (clock <clock-widget>)
                      (timer <clock-timer>))
    (set-clock-timer! clock timer)
    (add-method (subject-notify-function timer)
                (method ()
                  (draw clock))))
(defrectype <digital-clock> (<clock-widget>) ())
(gfmethod (draw (clock <digital-clock>))
    (format true "Drawing digital clock, time = ~a~%"
            (clock-timer-data (clock-timer clock))))
(defrectype <analog-clock> (<clock-widget>) ())
(gfmethod (draw (clock <analog-clock>))
    (format true "Drawing analog clock, time = ~a~%"
            (clock-timer-data (clock-timer clock))))
```

One implementation strategy discussed in the GOF allows observers to register with aspects of interest. At notification time, the subject supplies the changed aspect to its observers as a parameter to the Update operation. For example:

```
void Subject::Attach(Observer*, Aspect& interest);
...
void Observer::Update(Subject*, Aspect& interest);
```

The Scheme implementation of Observer encompasses finer-grained notification simply by allowing specializers on argument(s) to the `notify` methods to limit applicability.

3.20 State

> **Intent:** *Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.*

As mentioned in the GOF book, prototype/delegation-based languages like Self can implement the State pattern directly.

The `change-class` capability of CLOS could also be used.

In GLOS, we can maintain a state flag in the connection, as well as all the connection state, and use predicate types to dispatch:

```
class TCPState;
class TCPConnection {
public:
    void Open();
    void Close();
private:
    friend class TCPState;
    void ChangeState(TCPState*);
private:
    TCPState* _state;
};
class TCPState {
public:
    virtual void Open(TCPConnection*);
    virtual void Close(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
TCPConnection::TCPConnection () {
    _state = TCPClosed::Instance();
}
void TCPConnection::ChangeState (TCPState* s) {
    _state = s;
}
void TCPConnection::Open () {
    _state->Open(this);
}
void TCPConnection::Close () {
    _state->Close(this);
}
void TCPState::Open (TCPConnection*) { }
void TCPState::Close (TCPConnection*) { }
void TCPState::ChangeState
        (TCPConnection* t, TCPState* s) {
    t->ChangeState(s);
}
class TCPEstablished : public TCPState {
public:
    static TCPState* Instance();
    virtual void Close(TCPConnection*);
};
class TCPListen : public TCPState {
public:
    static TCPState* Instance();
    virtual void Send(TCPConnection*);
    // ...
};
class TCPClosed : public TCPState {
public:
    static TCPState* Instance();
    virtual void Open(TCPConnection*);
    // ...
};
void TCPClosed::Open (TCPConnection* t) {
    // send SYN, receive SYN, ACK, etc.
    ChangeState(t, TCPEstablished::Instance());
}
void TCPEstablished::Close (TCPConnection* t) {
    // send FIN, receive ACK of FIN
    ChangeState(t, TCPListen::Instance());
}
```

```
(defrectype <tcp-connection> ()
  ((status <symbol> 'closed))
  (status connection-status
          set-connection-status!))
(define <open>
  (and? <tcp-connection>
        (lambda (c)
          (eq? 'open (connection-status c)))))
(define <closed>
  (and? <tcp-connection>
        (lambda (c)
          (eq? 'closed (connection-status c)))))
(defgeneric open
  (method ((c <tcp-connection>))
          (error "Cannot open connection."))
  (method ((c <closed>))
          (set-connection-status! c 'open)))
(defgeneric transmit
  (method ((c <tcp-connection>) data)
          (error "Cannot transmit on connection."))
  (method ((c <open>) data)
          (format true "Transmitting: ~a~%" data)))
(defgeneric close
  (method ((c <tcp-connection>))
          (error "Cannot close connection."))
  (method ((c <open>))
          (set-connection-status! c 'closed)))
```

Summary: Predicate types allow one to modify behavior based on internal (or external for that matter) state of an object.

3.21 Strategy

> **Intent:** *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.*

First class functions clearly supply much of what Strategy needs – the ability to parameterize an object by the function/algorithm it uses to perform a particular task. Closures can be seen as instances of the Flyweight pattern – a given closure can be shared.

The `Compositor` classes in the sample code from the GOF,

```
class Compositor {
public:
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    ) = 0;
protected:
    Compositor();
};
class SimpleCompositor : public Compositor {
public:
    SimpleCompositor();
    virtual int Compose(
        Coord natural[], Coord stretch[], Coord shrink[],
        int componentCount, int lineWidth, int breaks[]
    );
    // ...
};
...
```

can clearly be replaced by closures in a functional language.

When the algorithm to be selected depends on context in a regular way, we can use generic function dynamic dispatch. Generic function dynamic dispatch is all about selecting the right code for the job.

As Peter Norvig notes, you still may want to encode other aspects of a strategy, but you need only

an instance, not a new class for each:

```
(new Strategy cost: 4 speed: 5)
```

3.22 Template Method

> **Intent:** *Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.*

Again from the GOF:

> Template methods are so fundamental that they can be found in almost every abstract class.

There is very little sample code given for the Template Method pattern, and, because of the pattern's universality, the code would not be much different in Scheme.

Protocols, as in metaobject protocols, are template methods.

3.23 Visitor

> **Intent:** *Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.*

The code in the book relies on the double dispatch idiom, which is tedious. Multiple dispatch avoids this problem. The GOF write:

> Effectively, the Visitor pattern lets you add operations to classes without changing them. Visitor achieves this by using a technique called *double-dispatch*. It's a well-known technique. In fact, some programming languages support it directly (CLOS, for example). Languages like C++ and Smalltalk support *single-dispatch*.

```
class Equipment {
public:
    const char* Name() { return _name; }
    virtual Currency NetPrice();
    // ...
    virtual void Accept(EquipmentVisitor&);
protected:
    Equipment(const char*);
private:
    const char* _name;
};
class EquipmentVisitor {
public:
    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
// and so on for other concrete
//   subclasses of Equipment ...
};
void FloppyDisk::Accept
        (EquipmentVisitor& visitor) {
   visitor.VisitFloppyDisk(this);
}
void Chassis::Accept(EquipmentVisitor& visitor) {
    for ( ListIterator i(_parts);
          !i.IsDone(); i.Next() ) {
        i.CurrentItem()->Accept(visitor);
    }
    visitor.VisitChassis(this);
}
class PricingVisitor : public EquipmentVisitor {
public:
    Currency& GetTotalPrice();
    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    // ...
private:
    Currency _total;
};
void PricingVisitor::VisitFloppyDisk
       (FloppyDisk* e) {
    _total += e->NetPrice();
}
void PricingVisitor::VisitChassis (Chassis* e) {
    _total += e->DiscountPrice();
}
```

```
(defrectype <visitor> () ())
(defrectype <equipment> ()
  ((name <symbol>)
   (price <number>))
  (name equipment-name set-equipment-name!)
  (price equipment-price set-equipment-price!))
(defrectype <composite-equipment> (<equipment>)
  ((parts <list>))
  (parts composite-parts set-composite-parts!))
(defgeneric visit
  (method ((e <equipment>) (v <visitor>))
      true)                    ; do nothing
  (method ((e <composite-equipment>) (v <visitor>))
      (for-each (rcurry visit v)
                (composite-parts e))))
(defrectype <floppy> (<equipment>) ())
(defrectype <chassis> (<composite-equipment>) ())
(defrectype <price-visitor> (<visitor>)
  ((total-price <number> 0))
  (total-price total-price set-total-price!))
(add-method*
 visit
 (method ((f <floppy>) (v <price-visitor>))
         (set-total-price! v
           (+ 1 (total-price v) (equipment-price f))))
 (method ((f <chassis>) (v <price-visitor>))
         (call-next-method)     ; visit subparts
         (set-total-price!
          v (+ 2 (total-price v) (equipment-price f)))))
```

While many people dismiss the Visitor pattern as being "nothing more than double dispatch," there are still design issues related to how to traverse composite objects in the graph. Having multiple dispatch available gets rid of some of the tedium of the Visitor pattern, but does not address all the design and implementation issues.

Note that the Demeter project at Northeastern addresses issues of object graph traversal.

4 Implementation Notes

Add signatures to generics. Then can have before "method" be a generic with methods (as opposed to a method with a callable that's a generic).

Make the subtyping relation be enforced always. That is, whenever a method is declared, make sure it respects the subtyping relation – that is, make sure the generic has coverage for all supers of the arg specializers.

References

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Nor96] Peter Norvig. Design patterns in dynamic programming. In *Object World 96*, Boston, MA, May 1996.

[Sha97] Andrew Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, Reading, Mass., 1997.

[SMI84] B. SMITH. Reflection and semantics in lisp. In *Conference Record of POPL '84: The* 11^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 23–35, 1984.

Contents