Design Patterns in OCaml

Antonio Vicente
int18@mit.edu
Earl Wagner
ejw@mit.edu

Abstract

The GOF Design Patterns book is an important piece of any professional programmer's library. These patterns are generally considered to be an indication of good design and development practices. By giving an implementation of these patterns in OCaml we expected to better understand the importance of OCaml's advanced language features and provide other developers with an implementation of these familiar concepts in order to reduce the effort required to learn this language. As in the case of Smalltalk and Scheme+GLOS, OCaml's higher order features allows for simple elegant implementation of some of the patterns while others were much harder due to the OCaml's restrictive type system.

Contents

1 Background and Motivation

Throughout this course we have seen many examples of methodologies and tools that can be used to reduce the burden of working in a software project. These tools focus force the programmer to take a different view toward the software development process and in exchange provide the programmer with ways to write code that is easier to check for correctness.

OCaml helps developers by forcing them to write software components that are well typed and more modular. It has a number of interesting features like first-class functions, pattern matching, lazy evaluation and static type checking that, combined with a powerful module system, simplify the implementation of many common programming tasks and help promote healthy coding practices. These features come at the expense of expressiveness and complexity which is often enough to prevent developers from learning and using OCaml in their development projects.

By trying to implement Gang of Four's (GOF) Design Patterns in OCaml, we expected to reduce the difficulties that new developers go though while trying to perform some common tasks in this language. The Design Patterns book is a good start because of the large number of professional programmers that know about these patterns and use them to make informed decision about how to design and implement a software system. In the GOF's book, the authors discuss how Smalltalk's higher order features simplify several of their patterns. This work was extended by several others including Greg Sullivan who analyzed how the dynamic features of his GLOS extension to Scheme affect the expression of many of these patterns.

Also, we noticed that very few programmers take full advantage of the OCaml's module system. Browsing though the code for Unison, we noticed that they make minimal use of functors and parameterized classes. We saw similar tendencies when looking at OCaml code written for ICFP programming competitions. We hoped that our work motivates other developers to learn how to use these advanced features and take advantage of them in their day to day programming tasks, or at least help us understand why these features are not as widely used as they should.

2 Results and Evaluation

The original intention was to implement the design patterns though heavy use of functors and modules in order to provide a framework that makes design patterns as transparent as possible. We soon realized that ML's semantics were too limited and made the implementation of many patterns extremely hard and verbose. An example of these difficulties appear in our initial attempt to implement the Abstract Factory pattern where we are forced to export accessors as part of the module's signature in order to keep types abstract.

After seeing these preliminary results we modified our approach to include OCaml's OO extensions in our study. These extensions allow for much more natural implementation of Design Patterns, but they are not enough to give simple elegant solutions for all patterns. For example, according to OCaml's documentation, the observer pattern often appears in the literature as a hard inheritance problem since it involves two classes that must be defined in terms of each other. OCaml's static checker cannot infer the type of the *add_observer* method unless you call that method somewhere in your code. Similar issues appear when trying to implement the Mediator and Visitor pattern using classes.

The following features either allow or simplify the implementation of Design Patterns in OCaml:

- **First-Class Functions:** Adapter, Composite, Command, Iterator, Strategy, Template Method
- **Pattern Matching:** Factory Method, **Interpreter**, State, Visitor
- **Functors:** Abstract Factory, Adapter
- **Module System:** Abstract Factory, Facade, Singleton, Flyweight, Proxy
- **Inheritance:** Factory Method, Chain of Responsibility, Template Method
- **Functional Object Update:** Prototype
- **Lazy Evaluation:** Flyweight
- **Parametrized Classes:** Command, Mediator, Observer, Visitor

Builder, Bridge, Decorator, Chain of Responsibility and Template Method remained mostly unaffected by OCaml's language features.

There are other operations like object unmarshalling, which is part of the Memento pattern, and object copy, which is part of the Prototype pattern, that are not fully implementable in OCaml since the

language does not support up-casting or recursive type/module definitions. We can get around some of these limitations by using our use of classes to those cases where they are strictly necessary and try to be consistent with respect to what interface is used to access instances.

3 Lessons Learned and Conclusions

OCaml sacrifices some expressiveness in exchange of type safety. There are things that just can't be expressed in the language and many others that are hard to get right the first time. These are not serious issues unless you try to write programs that require recursive class types or involve complex class hierarchies. Unfortunately, these are exactly the kinds of issues that design patterns try to address directly and therefore it is not surprising that we encountered so many problems.

The syntax and semantics of the language are rather complicated and were hard to reason about. We had a hard time trying to predict the problems that we would run into while trying to implement the patterns before trying to perform the actual implementation and the deciding how good our implementation is. It is not surprising that so many OCaml developers use pattern matching for almost everything and only use functors and parametrized classes when it is strictly necessary as in the case of the observer pattern.

4 Creational Patterns

4.1 Abstract Factory

In ML, modules and classes are not first class objects. It is possible to get around this limitation by using functors, but that can force us to have a large number of functors at the top level or modules that need to be functorized just because some module they use is. Abstract factories can be used to group together related modules and thus eliminate or significantly reduce the need many functorized top-level modules. Functors can be used to reduce bindings inside the factory but much less effectively than using reflection mechanisms.

Our implementation takes advantage of functors to define a family of abstract factories that satisfy the MAZEFACTORY module type and create two instances that provide different kinds of doors. When we are dealing with structures, the MAZEFACTORY module needs to expose basic functionality of doors, rooms and mazes though its interface in order to allow the user to call methods on the abstract types returned by the factory.

If we had used OCaml's OO extensions, this pattern would look much more similar to the C++ implementation. We can now select to export an interface that is satisfied by the objects that the factory returns in order to provide ways to interact with those objects.

absmazefactory.ml

```
open Mazefactory;;
open Enchanteddoor;;
open Normaldoor;;
open Enchantedroom;;
open Maze;;

open Doori;;
open Room;;
open Mazei;;

module Abstractmazefactory =
  functor (DoorT:DOORI) ->
    functor (RoomT:ROOM) ->
      functor (MazeT:MAZEI) ->
struct
  module CRoom = RoomT (DoorT)
  module CMaze = MazeT (RoomT) (DoorT)

  (* allows calling methods by means of the factory *)
  include DoorT
  include CRoom
  include CMaze

  type doortype = DoorT.dt
  type roomtype = CRoom.rt
  type mazetype = CMaze.mt

  let newDoor () = DoorT.buildDoor ()
  let newRoom () = CRoom.newRoom ()
  let newMaze () = CMaze.buildMaze ()
end;;


module AMazefactory : MAZEFACTORY = Abstractmazefactory (Enchanteddoor) (Enchantedroom) (Maze)
module BMazefactory : MAZEFACTORY = Abstractmazefactory (Normaldoor) (Enchantedroom) (Maze)
```

4.2 Builder

The builder pattern pretty much the same as in C++. You can pass the arguments to the builder either by using functors, using pattern matching or regular objects. The builder pattern as a construction protocol seems to be a universal pattern as Greg Sullivan points out, which seems to be among the list of universal patterns.

maze.ml

```
open Doori;;
open Room;;
open Mazei;;

module Maze : MAZEI =
  functor (RoomT:ROOM) ->
    functor (DoorT:DOORI) ->
struct
  module CRoom = RoomT (DoorT)
  type roomt = CRoom.rt
  type roomrel = roomt list
  type doorrel = DoorT.dt * CRoom.rt * CRoom.rt

  type mt = { rooms:roomrel ; doors: doorrel list}
  type mazebuilder = { mutable maze:mt }
  let newMaze rr dr = { rooms = rr ; doors = dr }
  let addRoom r mb = mb.maze <- newMaze (r::mb.maze.rooms) mb.maze.doors
  let newMazeBuilder () = { maze = newMaze [] [] }
  let connectRooms (r1:roomt) (r2:roomt) (mb:mazebuilder) =
    let newdoor = DoorT.buildDoor in
      mb.maze <- newMaze mb.maze.rooms ((newdoor (), r2, r1)::(newdoor (), r1, r2)::mb.maze.doors)
  let buildMaze () = newMaze [] []
end;;
(*  val newMaze : roomrel -> doorrel -> t*)
```

4.3 Factory Method

In OCaml, factory methods can be implemented in two different ways. We can either make a factory that only returns objects that are subclasses of each other or we can wrap the results inside an enumeration type that we can later access though pattern matching. The second approach requires heavy use of wrappers in order to be able to provide some of the functionality of the original objects to objects that are wrapped in the enumeration class. On the other hand, the pattern matching approach is the only to provide mechanisms to cast objects up the class hierarchy which might be essential in some cases. This approach is problematic because each time you add a new type to the factory you need to go back and update the code around all places where the factory is used. It all depends on what flexibility we need and what kinds of objects our factory method will need to be able to generate.

factorymethod.ml

```
exception Unknowndoor;;

module Normaldoor =
struct
  type t = { id:int }
  let buildDoor () =  { id = 0 }
  let doorIsOpen n = true
end;;

module Enchanteddoor =
struct
  type t = { id:int; isOpen:bool; spell:string }
  let buildDoor () =  { id = 0; isOpen = false; spell = "currying" }
```

```
  let doorIsOpen n = n.isOpen
end;;

type doortype = Normal of Normaldoor.t | Enchanted of Enchanteddoor.t
let doorIsOpen d = match d with
    Normal dr -> Normaldoor.doorIsOpen dr
  | Enchanted dr -> Enchanteddoor.doorIsOpen dr

(*val doorfactory : string -> DOOR*)
let doorfactory n = match n with
    "normal" -> Normal (Normaldoor.buildDoor ())
  | "enchanted" -> Enchanted (Enchanteddoor.buildDoor ())
  |  _ -> raise Unknowndoor
```

factorymethod2.ml

```
class virtual door =
object (self : 't)
  method virtual doorOpen : unit -> unit
  method virtual doorClose : unit -> unit
  method virtual isOpen : unit -> bool
end;;

class normaldoor () =
object (self)
  inherit door
  val mutable opened = false
  method doorOpen () = opened <- true
  method doorClose () = opened <- false
  method isOpen () = opened
end

class enchanteddoor () =
object (self)
  inherit door
  val mutable opened = false
  val mutable cursed = true
  method doorOpen () = if not cursed then opened <- true
  method doorClose () = if not cursed then opened <- false
  method isOpen () = opened
end

exception Unknowndoor

let doorfactory n = match n with
    "normal" -> ((new normaldoor ()):door)
  | "enchanted" -> ((new enchanteddoor ()):door)
  | _ -> raise Unknowndoor
```

4.4 Prototype

There is no notion of reflection or a standard copy interface. It is up to the user to define and implement such an interface if he wants to be able to copy objects. OCaml provides a way to perform functional object updates which can simplify the implementation of the prototype pattern.

When defining or implementing a copy interface we will have to take into account the fact that OCaml does not support casting objects up the class hierarchies. This problem can be solved by using the same mechanisms used to implement the factory method. This suggests that OCaml would require at least two "standard" copy interfaces and it is not clear that even then that would be enough.

4.5 Singleton

This pattern appears as a central concept in ML's module system. Modules act as global singletons that can provide all kinds of functionality. Functor instantiation is another good example how singletons appear in ML.

5 Structural Patterns

5.1 Adapter

The GOF discuss class and object versions of Adapter. In the class version of the pattern, a subclass of the Adaptee class is created and extra code is added to satisfy the Target interface. In the object version, a class satisfies the Target by creating an instance of the Adaptee class and delegating to it.

As in the case for Smalltalk, it is possible to implement the Adapter pattern using anonymous functions in Caml. The GOF point out that the disadvantage of using an object Adapter is it is more difficult to change the Adaptee object's behavior. This can be solved by making the Adapter class a parameterized module with the Adaptee's object specified at compile time. This is less flexible than specifying the type of the Adaptee object at runtime, using an Abstract Factory, however it permits static analysis.

adapter.ml

```
(*class version*)

class virtual windowwidget =
object
  method virtual bbox : unit ->  (int * int) * (int * int)
end

class textshape s =
object
  inherit windowwidget
  method bbox () = (0,0), (8 * String.length s, 10)
end

(*instance version*)

class windowwidgetadaptor bboxm resizem =
object
  method bbox = (bboxm : unit ->  (int * int) * (int * int))
  method resize = (resizem : int -> int -> unit)
end

exception Opnotsupported

let s = "test"
let wwa = new windowwidgetadaptor (fun () -> (0,0), (8 * String.length s, 10))
            (fun x y -> raise Opnotsupported)
```

5.2 Bridge

As Greg Sullivan notes, the Bridge pattern, in which an implementation is defined separately from its interface, appears to be a universal idea for abstraction.

5.3 Composite

In the composite pattern, the interface to a tree data structure is simplified by providing the same interface for leaf and non-leaf nodes in the tree.

One question in implementing the Composite pattern is which methods for the Leaf class to incorporate into the Component interface. They give the example of calling "draw" on a canvas, which then calls "draw" on all of its children. This can be solved using closures with functions like "map" and "filter", which could apply the function to all of the Leaf objects. In this case, the canvas would receive "map draw" and call this on all of its children. The Leaf objects would then draw themselves.

5.4 Decorator

The Decorator pattern is used to dynamically add and remove responsibilities to and from objects. We implement this as it is implemented in C++:

decorator.ml

```
class virtual visual_component =
  object (self)
          method virtual draw: unit -> unit
          method virtual resize: unit -> unit
  end;;

class basic_visual_component =
  object (self)
          inherit visual_component
          method draw () = Printf.printf "draw\n"
          method resize () = Printf.printf "resize\n"
  end;;

class decorator init_visual_comp =
  object (self)
        inherit visual_component
        val my_comp = (init_visual_comp: visual_component)
        method draw () = my_comp#draw ()
        method resize () = my_comp#resize ()
  end;;

class border_decorator (init_visual_comp, init_border_width) =
  object (self)
        val my_border_width = init_border_width
        inherit decorator init_visual_comp as super
        method draw_border (border_width: int) = Printf.printf "draw_border\n"
        method draw () = super#draw (); self#draw_border my_border_width
        method resize () = super#resize ()
  end;;

(*
let bvc = new basic_visual_component;;
# let bvc = new basic_visual_component;;
val bvc : basic_visual_component = <obj>
# let bd = new border_decorator (bvc, 1);;
val bd : border_decorator = <obj>
# bd#draw;;
- : unit -> unit = <fun>
# bd#draw ();;
draw
draw_border
- : unit = ()
*)
```

5.5 Facade

The Facade pattern is an example of how to break down a program to increase modularity. ML's module system provides support this pattern directly by allowing the user can explicitly say exactly what they want to export to specific clients.

5.6 Flyweight

These are basically the idea of making everything an object to abstract away representation of even the simplest objects. this can be done by instantiating objects on module loading and then making a factory method to access these new objects. This pattern is worth using for the same purposes as in C++.

5.7 Proxy

A Proxy is used to control access to another object. This is useful in the case that the object is located over the network, as in the case of a network proxy, when the object is expensive to create and is instantiated on demand, as in the case of a virtual proxy, when there should be different access rights to the object, or when additional behaviors are required of a reference to the object, as in the case of a smart reference.

There appears to be no easy way to implement a virtual proxy, nor smart reference in ML, and the implementation network proxy depends upon the details of the network interface. However, a protection proxy could be implemented using different signatures for the same structure. An example of this is given in the "Extended Example: Managing Bank Accounts" section of the OCaml book. book. [2]

proxy.ml

```
# module Manager =
  FManager (Account)
           (Date)
           (FLog(Date))
           (FStatement (Date) (FLog(Date))) ;;
module Manager :
  sig
    type t =
      FManager(Account)(Date)(FLog(Date))(FStatement(Date)(FLog(Date))).t =
      { accnt: Account.t;
        log: FLog(Date).t }
    val create : float -> float -> t
    val deposit : FLog(Date).tinfo -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statement :
      (FLog(Date).t -> (Date.t * float) list) -> t -> string list
    val statementB : t -> string list
    val statementC : t -> string list
  end

# module type MANAGER_BANK =
   sig
    type t
    val create : float -> float -> t
    val deposit : float -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statementB : t ->  string list
   end ;;

# module MBank = (Manager:MANAGER_BANK with type t=Manager.t) ;;
module MBank :
```

```
  sig
    type t = Manager.t
    val create : float -> float -> t
    val deposit : float -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statementB : t -> string list
  end

# module type MANAGER_CUSTOMER =
   sig
    type t
    val deposit : float -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statementC : t ->  string list
   end ;;

# module MCustomer = (Manager:MANAGER_CUSTOMER with type t=Manager.t) ;;
module MCustomer :
  sig
    type t = Manager.t
    val deposit : float -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statementC : t -> string list
  end
```

In the example, a bank account is implemented as a Manager module created from the functor module FManager. Unlike the customer, the bank has the ability to directly create the account and both have different requirements for what they want from their statements. These different interfaces provide different functionality for the bank and the customer when interacting with the same account object. Although a language like Java allows a single class to implement different interfaces, in ML new interfaces can be created for classes without having to change and recompile the class definition.

6 Behavior Patterns

6.1 Chain of Responsibility

The implementation of this pattern in OCaml looks very similar to the C++ implementation. New handlers can be added by overriding the handleRequest method and delegating any unhandled messages to the next handler in the chain.

chain.ml

```
exception UnhandledRequest

class ['request] chain =
object(self : 'a)
  (*what i really wanted is something i can store either an object or
    null.  lists work great*)

  val mutable nextHandler = ([]:'a list)
  method setNextHandler h = nextHandler <- [h]

  (*override to provide handler functionality*)
  method handleRequest (r:'request) = (raise UnhandledRequest:unit)
  method delegateRequest (r:'request) = match nextHandler with
      [] -> raise UnhandledRequest
```

```
    | x::xs -> (x#handleRequest r:unit)
end

type request = Help of string | Ok of int

class helphandler =
object (self : 'a)
  inherit [request]chain
  method handleRequest h = print_string "handler 1"; match h with
      (Help "test") -> print_string "request handled"; ()
    | _ -> self#delegateRequest h
end

class helphandler2 =
object (self : 'a)
  inherit [request]chain
  method handleRequest r = print_string "handler 2"; match r with
      (Help "test2") -> print_string "request 2 handled"
    | _ -> self#delegateRequest r
end

class client () =
object(self)
  val handleChain = let h1 = new helphandler in
    let h2 = new helphandler2 in
      h2#setNextHandler h1; h2
  method dostuff () = handleChain#handleRequest (Help "test")
end

let main () =
  let c = new client () in
    c#dostuff();
    print_newline();
    exit(0);;

main ();;
```

6.2 Command

The Command pattern represents a call on an object as an object itself. This can be helpful when information about the call is known, but the object that will receive the call is unknown. In essence, this is currying, which ML supports directly. It may be that the call is required to be undo-able. In this case, a more sophisticated solution will be necessary.

Here is the structure for the simple case:

command.ml

```
class ['a] receiver =
  object (self)
        method perform_action (action: ('a -> 'b)) = action self
  end;;

class ['a] simple_command (receiver_init, action_init) =
  object (self)
        val my_receiver = (receiver_init : 'a #receiver)
        val my_action = action_init
        method execute () = my_receiver#perform_action my_action
  end;;
```

6.3 Interpreter

look at desk calculator example. pattern matching makes interpreters easy to write. ocalm people seem to jis over this

appears in web page as calculator example. pattern matching makes writting parsers and interpreters easy.

An example of this pattern is given in the "Desktop Calculator Example" section of the OCaml book. [2]

6.4 Iterator

The GOF distinguish between external iteration, when iteration is controlled by the client interacting with the iterator, from internal iteration in which the client passes a function to the iterator for it to perform on each element. Internal iteration is implemented by passing a curried function to List.map, which then applies the function to each element of the list. External iteration can also be performed using the List module, by applying the function to successive elements in a list.

6.5 Mediator

In this pattern, a Mediator coordinates the actions of a set of Colleagues. When one Colleague changes and needs to make changes to other Colleagues, it notifies the Mediator, which itself interacts with the other Colleagues. This requires that each Colleague has a reference to its Mediator, and that the Mediator itself keeps track of all of the Colleagues. This cannot be implemented using ML modules, since they are not permitted to be recursively dependent upon each other.

This pattern can be implemented using OCaml's classes, in which case it is not much different from the C++ case:

mediator.ml

```
class virtual ['widget] dialog_director =
  object
    method virtual widget_changed : 'widget -> unit
    method virtual show_dialog : unit -> unit
  end;;

class ['dialog_director] widget init_dialog_director =
  object (self)
    val my_dialog_director = (init_dialog_director: 'a #dialog_director)
    method changed () = my_dialog_director#widget_changed self
  end;;
```

6.6 Memento

While designing and implementing a marshalling protocol for ML, we would expect to run into many of the same difficulties and design decisions that we saw while analysing the prototype pattern. The main problem here is with the return type of a function for unmarshalling. You cannot up-cast the unmarshalled value to return it to its original type.

As in the case of the prototype pattern, implementing the Memento pattern in ML poses many difficulties.

6.7 Observer

This pattern requires mutually-recursive dependence among the subject and observer classes; the observer must be aware of the type of the subjects in order to notify them and the subjects must be aware of the type of the observer to notify it. This can be implemented with parameterized classes, but a meaningless method call must be inserted into the code because OCaml's static checker cannot infer the type of the *add_observer* unless you call that it somewhere in your code.

The following example of this pattern appears in the "Advanced Examples with Classes and Modules" section of the OCaml documentation [1]

observer.ml

```
class virtual ['subject, 'event] observer =
  object
    method virtual notify : 'subject ->  'event -> unit
  end;;

class ['observer, 'event] subject =
  object (self)
    val mutable observers = ([]:'observer list)
    method add_observer obs = observers <- (obs :: observers)
    method notify_observers (e : 'event) =
      List.iter (fun x -> x#notify self e) observers
  end;;

type event = Raise | Resize | Move;;

let string_of_event = function
    Raise -> "Raise" | Resize -> "Resize" | Move -> "Move";;

let count = ref 0;;

class ['observer] window_subject =
  let id = count := succ !count; !count in
  object (self)
    inherit ['observer, event] subject
    val mutable position = 0
    method identity = id
    method move x = position <- position + x; self#notify_observers Move
    method draw = Printf.printf "{Position = %d}\n"  position;
  end;;

class ['subject] window_observer =
  object
    inherit ['subject, event] observer
    method notify s e = s#draw
  end;;

let window = new window_subject;;
let window_observer = new window_observer;;

(*this line here is essential for correctness.  this is so broken!*)
window#add_observer window_observer;;
```

6.8 State

As with many of the patterns above, there are two ways to implement this pattern: using pattern matching or objects. Pattern matching requires knowledge of the types of connection a priori and creating a new type that can be used to store extra information. This kind of state objects are lightweight and allow for centralized implementation of behavior suggested by the pattern. The other alternative is to define a class that all the state classes derive from. These classes would have to be able to call each other's constructors and thus need to be defined in the same class block which is extremely ugly and inconvenient.

state.ml

```
type TCPConn = OpenConn of int | ClosedConn of string;;
```

```
let transmit connection data = match connection with
                          (OpenConn c) -> sendPacket connection data
                          | _ -> Error "closed connection";;
```

6.9 Strategy

The Strategy pattern is used to factor out algorithms from the classes that use them. The GOF put the implementation of the algorithm in a separate class, but as Greg Sullivan points out, with first-class functions, algorithms can be expressed as closures and passed into client classes directly.

Here is a context class that uses this approach:

strategy.ml

```
class ['a, 'b] context (strategy_init) =
  object (self)
        val my_strategy = (strategy_init: ('a -> 'b))
        method operation x = my_strategy x
  end;;
```

6.10 Template Method

Template Method is implemented using the OCaml object system. Because it allows inheritance and delayed-binding of method calls, this is similar to the implementation in C++.

templatemethod.ml

```
class virtual abstract_template =
  object (self)
    method virtual hook_operation : unit -> unit
    method operation () = self#hook_operation ()
  end;;

class concrete_template =
  object
        inherit abstract_template as super
    method operation () = super#operation
    method hook_operation () = Printf.printf "operation\n"
  end;;
```

6.11 Visitor

The visitor pattern expresses the need for full information about an object's class that is needed when trying to perform external operations on a group of objects. The inability to test class membership dynamically or up cast values forces us to use pattern matching or double dispatch to implement this pattern.

The pattern implementation provides full information about the node real type so it can perform meaningful computations as the visitor now has full knowledge about the node's type and methods. As long as none of the classes are subclasses of each other, pattern matching protects the code from problems that appear in other OO languages due to carelessness.

The other alternative is to define mutually-dependent classes similar to the ones used to implement the observer pattern. This implementation suffers from the same typing problems that the observer pattern suffers.

visitor-simple.ml

```
class virtual equipment =
object
  method virtual name : unit -> string
  method virtual price : unit -> int
```

```
end

class virtual compositeequipment =
object
  inherit equipment
end

class floppy () =
object (self : 'a)
  inherit equipment
  method name () = "floppy"
  method price1 () = 5
  method price () = 5
end

class chassis () =
object (self)
  inherit equipment
  method name () = "chassis"
  method price2 () = 10
  method price () = 10
end

class virtual equipmentvisitor =
object (self)
  method virtual visitFloppy : floppy -> unit
  method virtual visitChassis : chassis -> unit
end

class pricevisitor () =
object
  inherit equipmentvisitor
  val mutable totalprice = 0
  method visitFloppy f = totalprice <- totalprice + f#price ()
  method visitChassis c = totalprice <- totalprice + c#price ()
end

type equip = Floppy of floppy | Chassis of chassis

let visit e (v:equipmentvisitor) = match e with
    (Floppy e1) -> v#visitFloppy e1
  | (Chassis e2) -> v#visitChassis e2
```

visitor.ml

```
(*the equipment*)
class virtual ['equipmentvisitor]equipment =
object
  method virtual name : unit -> string
  method virtual price : unit -> int
  method virtual accept : 'equipmentvisitor -> unit
end

class virtual ['equipmentvisitor]compositeequipment =
object
  inherit ['equipmentvisitor]equipment
end
```

```
class ['equipmentvisitor]floppy () =
object (self : 'a)
  inherit ['equipmentvisitor]equipment
  method name () = "floppy"
  method price () = 5
  method accept v = v#visitFloppy self
end

class ['equipmentvisitor]chassis () =
object (self)
  inherit ['equipmentvisitor]equipment
  method name () = "chassis"
  method price () = 10
  method accept v = v#visitChassis self
end

class ['equipmentvisitor]computer () =
object (self:'a)
  inherit ['equipmentvisitor]compositeequipment
  val mutable subcomponents = ([] : 'a list)
  method name () = "chassis"
  method price () = 10
  method accept v = List.map (fun c -> c#accept v) subcomponents; ()
end



(*visitor classes*)
class virtual ['floppy, 'chassis] equipmentvisitor =
object (self)
  method virtual visitFloppy : 'floppy -> unit
  method virtual visitChassis : 'chassis -> unit
end

class ['floppy, 'chassis]pricevisitor () =
object
  inherit ['floppy, 'chassis]equipmentvisitor
  val mutable totalprice = 0
  method visitFloppy f = totalprice <- totalprice + f#price ()
  method visitChassis c = totalprice <- totalprice + c#price ()
end

class ['floppy, 'chassis]othervisitor () =
object
  inherit ['floppy, 'chassis]equipmentvisitor
  val mutable totalprice = 0
  method visitFloppy f = print_string (f#name ())
  method visitChassis c = print_string (c#name ())
end

(*these 8 lines are essential, they make the visitor and the visitee
cross reference types*)

let a = new floppy ();;
let b = new chassis ();;
let c = new othervisitor ();;
let d = new pricevisitor ();;
```

```
let _ = a#accept c;;
let _ = b#accept c;;
let _ = a#accept d;;
let _ = b#accept d;;
```

7 References

1. X. Leroy, D. Doligez, J. Garrigue, D. R'emy, and J. Vouillon. *The Objective CAML system, documentation and user's manual*, 2000. http://caml.inria.fr/ocaml/htmlman/.

2. E. Chaillox, P. Manoury, B. Pagano. *Developing Applications with Objective Caml*, 2002 http://caml.inria.fr/oreilly-book/html/index.html

3. G. Sullivan. *Advanced Programming Language Features for Executable Design Patterns "Better Patterns Through Reflection"*, March 2002. MIT AI Lab, AI Memo 2002-005