

# A Multiparadigmatic Study of the Object-Oriented Design Patterns

Philippe Narbel

LaBRI, Bordeaux 1 University

MPOOL'07

# Situation and Tactics

- The understanding of paradigms is **not so satisfactory yet**.
- Though, most mainstream languages, C++, Java, ML, etc., **are multiparadigm languages**.
- ⇒ **Difficult choices, dilemma** for the programmer...
- **Possible tactics** : To develop, use, study multiple realizations of the same programming problem with various paradigms, styles, languages (⇒ **“programming experiences”**).

# Design Patterns

- A **design pattern** describes a solution that addresses a recurrent design problem in programming.
- Mostly recognized and developed for object-oriented programming at a “**micro-architectural**” level.  
(cf. *Design Patterns. par Gamma, Helm, Johnson, Vlissides. Addison-Wesley, 1995*)  
(the so-called **GoF book**).
- **Exist in any paradigm, style or language.**
- NB: there is **no formal** definition of a design pattern.

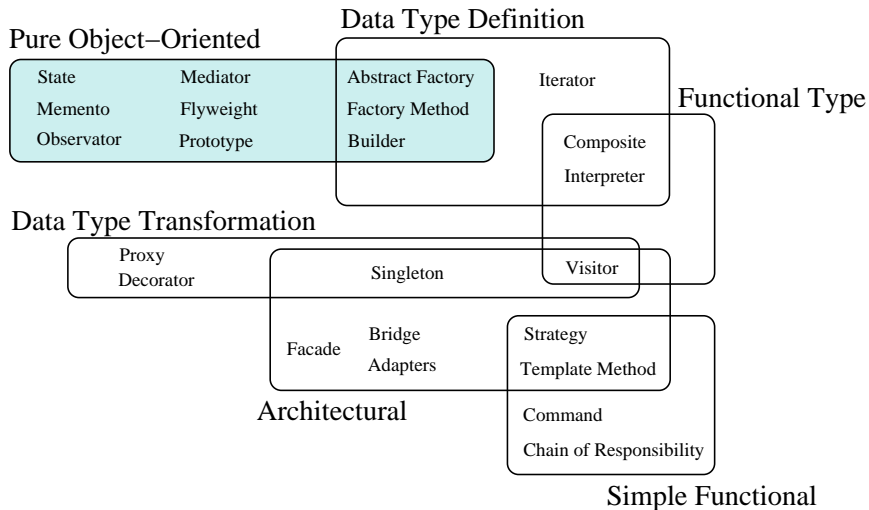
# Some General Questions

- Are patterns **intrinsic** to paradigms?
- May patterns **contribute to the understanding** of paradigms?
- May patterns in one paradigm **help in finding patterns** in other paradigms?

# A Programming Experience

- **To consider the OOP design patterns of the GoF book, and develop, use, study multiparadigmatic realizations of them.**
- Indeed, the **intents** of these patterns are **more general than OOP-oriented**. They deal with:
  - **General architectural problems.**
  - **Data type definition/modification/organization problems.**
  - **Behavior flexibility problems.**
- Also, paradigms sometimes **offer similar constructions** with sometimes different properties, e.g. wrt. **simplicity, encapsulation, type-safety, extensibility**, etc.)

# A Multi-paradigmatic Vision of the OOP Design Patterns



# Some References

- **OOP with generics (with more static modular flavor):**
  - Alexandrescu. *Modern C++ Design. Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.
  - Vandevorode, Josuttis. *C++ Templates*. Addison-Wesley. 2003.
  - Czarnecki, Eisenecker. *Generative Programming*. Addison-Wesley. 2000.
  - Naftalin, Wadler. *Java Generics*. O'Reilly. 2006.
- **OOP with functional programming:**
  - Kühne. *Functional Pattern System for Object-Oriented Design*. Verlag Kovac, 1999.
  - McNamara, Smaragdakis. *Functional programming with the FC++ library*, J. Funct. Program., 2004.

# Some References

- **Modern module-systems:**

- Harper, Pierce. *Design considerations for ML-Style module systems*, MIT Press, 2005.
- Dreyer. *Understanding and Evolving the ML Module System*, PhD, CMU, 2005.

- **Other general discussion about the GoF patterns:**

- Norvig, *Design Patterns in Dynamic Programming*. Harlequin, Inc., 1996.
- Baumgartner, G. et a. *On the Interaction of Object-Oriented Design Patterns and Programming Languages*. 1998. Purdue University.
- Gibbons. *Patterns in datatype-generic programming*. DPCOO'2003.

# General Remarks about the GoF Design Patterns

- Most of the GoF design patterns address problems **about encapsulated units**, ensuring flexibility while preserving encapsulation.
- Most of the GoF design patterns do **not appear directly as missing language constructs**.

# OOP and Generic Modular Programming

- Some respective advantages of OOP and modular programming:
  - **Advantages of OOP**: dynamic variation/choice of encapsulated code, and exploitation of object type compatibilities.
  - **Advantages of modular programming**: static verification and type abstraction and safety, and more optimization possibilities.
- Indeed, OOP  $\Rightarrow$  static typing problems with ***n*-ary methods** and **late bindings**.
- But there are also **many similarities** (in particular with ML module systems).

# Interfaces and signatures

- The relationship **interface/implemented class** is similar to **signature/module**.
- NB: ML module signatures are interfaces which **can be instantiated by more than one module**.
- In OCaml:

```
module type S = sig
  type t
  val f : t * t -> int
  [...]
end
```

```
module M : S = struct
  type t = int
  let f (x, y) = 2*(x + y)
  [...]
end
```

# Module Inheritances

- **Inclusion** is a simple way of implementing inheritance:

- **Module inheritance:**

```
module M1 = struct [...] end
module M2 = struct include M1 [...] end
```

- **Signature inheritance:**

```
module type S1 = sig [...] end
module type S2 = sig include S1 [...] end
```

- S2 is **compatible** with S1, that is, if a module M2 implements S2, it also implements S1.
- For a parameterized module  $F(X:S1)$ ,  $F(M1)$  and  $F(M2)$  are **both valid**.

# Generic Inheritances

- Inheritance generalized into **generic inheritance**:

```
module F (X : S1) = struct
  include X
  [...]
end
```

```
module M2 = F (M1)
```

- Some **overrides** may be available when multiple definitions are allowed (though with no **late binding** capabilities).

# Module Delegations

- **Module Delegation:** the same as in OOP but with nested modules.

```
module type WINDOW_SYS = sig
  val device_rect = [...]
  [...]
end
```

```
module type WINDOW = sig
  module _Imp : WINDOW_SYS
  val draw_rect : [...]
  [...]
end
```

```
module Win : WINDOW = struct
  module _Imp : WINDOW_SYS = [...]
  let draw_rect = _Imp.device_rect [...]
end
```

# Module Delegation is Static

- The assignment of the module delegate is **static**.
- Generic modular programming is of effective help:

```
module Win (X : WINDOW_SYS) : WINDOW = struct
  module _Imp : WINDOW_SYS = X
  let draw_rect = _Imp.device_rect [...]
end
```

- $\Rightarrow$  Whenever delegation is used as a technique to improve the global qualities of an architecture, **object and modular delegation are quite similar**.

# Abstract Classes

- Although no similar mechanism as abstract classes exists for modules, a similar code factoring effect can be obtained:

```
module type S1 = sig
  val f : [...]
  [...]
end
```

```
module type S2 = sig
  include S1
  val g : [...]
  [...]
end
```

```
module AbstractComponent (X : S1) : S2 = struct
  include X
  let g = [...] /* implemented elements */
  [...]
end
```

# Summing up the Relationships

## Object-Oriented

interface/implemented class  
inheritance  
delegation  
abstract classes

↔  
↔  
↔  
↔

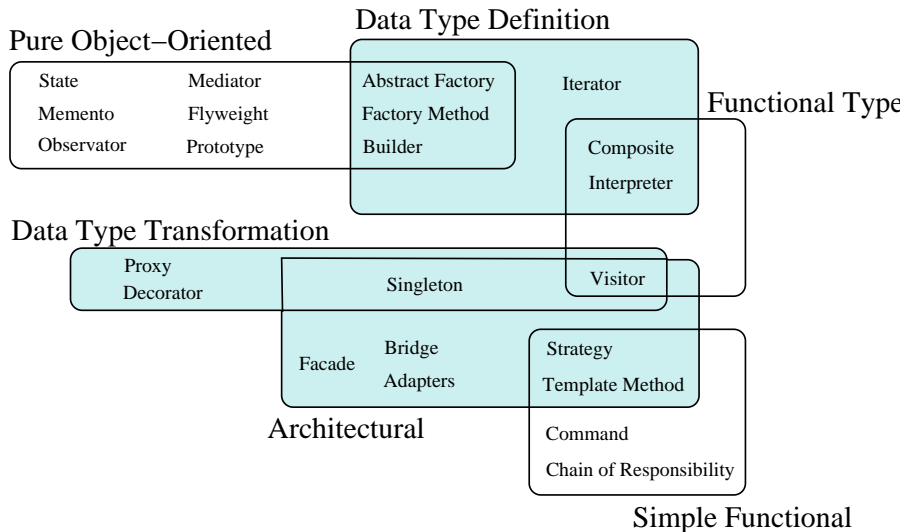
## Generic Modular

module signature/module  
inclusion  
nesting + generics  
inclusion + generics

# Generic Modular Design Patterns

- **Architectural Patterns:** OOP patterns addressing general software architecture problems, i.e., patterns able to make units more adaptable and more composable in a general setting.
- **Data Type Definition Patterns:** OOP patterns able to make specific object/data type definitions more adaptable and more composable.
- **Data Type Transformation Patterns:** OOP patterns having a global and uniform effect on existing user-defined data types, and on the objects/values they produce.

# Generic Modular Design Patterns



# Architectural Pattern: Bridge

- **Bridge** *decouples an abstraction from its implementation so that the two can vary independently (using delegation).*

```
module type GRAPHIC_SYS = sig
  type t
  val device_draw : t -> unit
  [...]
end
```

```
module type WINDOW = sig
  module W : GRAPHIC_SYS
  type t = W.t
  val draw : t -> unit
end
```

```
module Win (X : GRAPHIC_SYS) : WINDOW = struct
  module Sys = X
  type t = Sys.t
  let draw w = [...] Sys.device_draw [...]
end
```

# Architectural Pattern: Adapters

- **Adapter** *converts an interface of a component into another interface clients expect.*

```
module type ADAPTEE = sig
  type t
  val f : t -> t
end
```

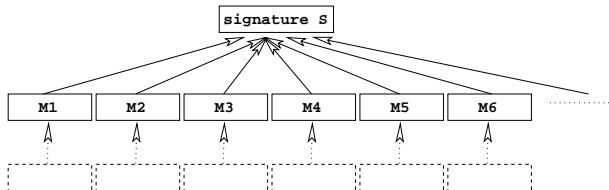
```
module type ADAPTER = sig
  include ADAPTEE
  include TARGET
end
```

```
module type TARGET = sig
  type s
  val eff : s -> s
end
```

```
module Adapter (A : ADAPTEE) : ADAPTER =
struct
  include A
  type s = A.t
  let eff = A.f
end;;
```

# Architectural Pattern: Visitor

- **Visitor** lets you define a new operation without changing the classes of the elements on which it operates.
- **Visitor** lets an **encapsulated unit hierarchy be more extensible**.
- In modular programming, the same extension problems as in OOP may occur:



# Architectural Pattern: Visitor

```
module type MATH = sig  
  type t  
  val f : t -> t -> t  
end;;
```

```
module MathInt = struct  
  type t = int  
  let f = ( + )  
end;
```

```
module MathInt_Div = struct  
  include MathInt  
  let g = ( / )  
end;
```

```
module MathFloat = struct  
  type t = float  
  let f = ( +. )  
end;
```

```
module MathFloat_Div = struct  
  include MathFloat  
  let g = ( /. )  
end;
```

# Architectural Pattern: Visitor

```
module type MATH_VISITOR = sig
  type t_int
  type t_float
  val visit_int : t_int -> t_int -> t_int
  val visit_float : t_float -> t_float -> t_float
end;;

module Math_Int_Visitor
  (M : MATH)
  (V : MATH_VISITOR with type t_int = M.t) = struct
  include M
  let accept = V.visit_int
end;;

module Math_Float_Visitor
  (M : MATH)
  (V : MATH_VISITOR with type t_float = M.t) = struct
  include M
  let accept = V.visit_float
end;;
```

# Data Type Definition Pattern: Composite

- **Composite** allows one *to compose objects into tree structures to let clients treat individual objects and compositions of objects uniformly.*
- Instance mixing *a priori* not possible for modular abstract data types.
- But, the idea of **composing instances** can be kept:

```
module type TYPE =  
sig  
  type t  
  val f : t -> unit  
  [...]  
end
```

```
module Composite (X : TYPE) : TYPE =  
struct  
  type t = X.t list  
  let f l = List.iter X.f l  
  [...]  
end
```

- **Static type safety is ensured.**

# Data Type Definition Pattern: Iterator

- **Iterator** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

```
module type LIST = sig
  type 'a lst
  val empty : 'a lst
  [...]
end;;
```

```
module type ITERATOR = sig
  type 'a iter
  type 'a iterated
  val make : 'a iterated -> 'a iter
  val next : 'a iter -> ('a iter * 'a)
  [...]
end;;
```

```
module type ITER_LIST = sig
  module L : LIST
  include (ITERATOR with type 'a iterated = 'a L.lst)
end;;
```

# Data Type Definition Pattern: Iterator

```
module Iterator1 (X : LIST) : ITER_LIST = struct  
  module L = X  
  type 'a iter = [...]  
  type 'a iterated = 'a L.lst (* type coherence checked *)  
  [...]  
end;;
```

```
module Iterator2 (X : LIST) : ITER_LIST = struct  
  module L = X  
  type 'a iter = [...]  
  type 'a iterated = 'a L.lst (* type coherence checked *)  
  [...]  
end;;
```

# Data Type Transformation Pattern: Proxy

- **Proxy** provides a surrogate or placeholder for another object to control access to it.

```
module type SUBJECT =  
sig  
  type t  
  val make_t : unit -> t  
  val use_t : t -> unit  
  [...]  
end
```

```
module Proxy (X : SUBJECT) : SUBJECT =  
struct  
  type t = X.t lazy_t  
  let make_t () = lazy (X.make_t ())  
  let use_t x = X.use_t (Lazy.force x)  
  [...]  
end
```

# Data Type Transformation Pattern: Decorator

- **Decorator** *attaches additional responsibilities to an object dynamically.*

```
module type GRAPHIC = sig
  type t
  val draw : t -> unit
end;;
```

```
module Border (G : GRAPHIC) : GRAPHIC = struct
  type t = G.t
  let draw_border g = [...]
  let draw g = draw_border (G.draw g);
end;;
```

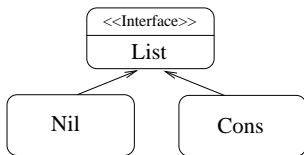
```
module DropShadow (G : GRAPHIC) : GRAPHIC = struct
  type t = G.t
  let drop_shadow g = [...]
  let draw g = drop_shadow (G.draw g);
end;;
```

```
module Text : GRAPHIC = struct [...] end;
module G = DropShadow (Border (Text)) ;;
```

# OOP and Functional Programming

- At least two sources of similarities between OOP and functional programming:
  - **Closures**. First-class functions in functional programming are represented by *closures*, i.e., private environments associated with function code, i.e. a special kind of **objects**.
  - **Inductive types**. Functional data types are *union types* or *inductive types* for which instances take the form of function compositions.

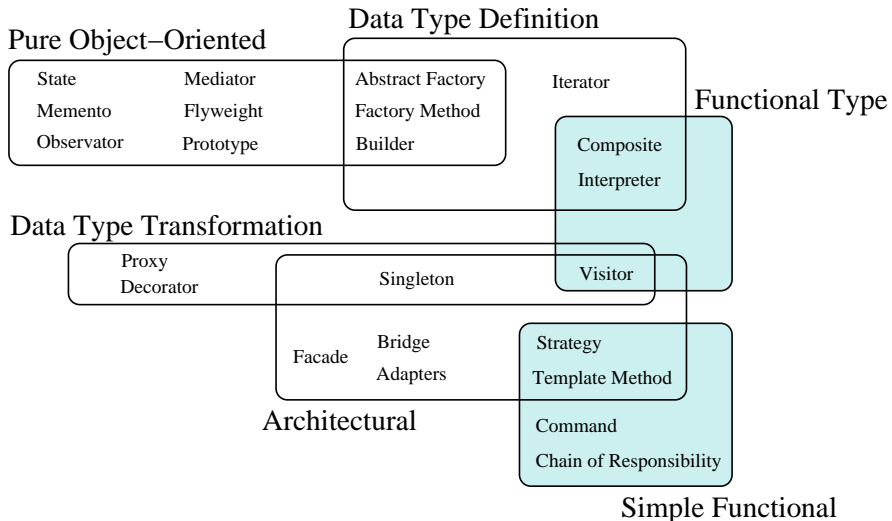
```
type 'a list =  
| Nil  
| Cons of 'a * 'a list
```



# Functional Design Patterns

- **Simple Functional Patterns:** OOP patterns addressing single method encapsulation/transmission problems.
- **Functional Type Patterns:** OOP patterns addressing union-like and recursive type definition problems.

# Functional Design Patterns



# Simple Functional Pattern: Command

- **Command** *encapsulates a request as an object, thereby letting you parameterize clients with different requests.*
- A “request” is just an operation considered as a first-class value.

```
let open () = ...;;  
let paste () = ...;;  
let menu_actions = [| open; paste |];;  
let apply_actions index actions = actions.(index) ();;
```

- NB: **C++ functors** are often used in this case.

# Simple Functional Patterns: Strategy and Template

- **Strategy** *lets an algorithm vary independently from clients that use it.*
- **Template method** *lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.*

# Simple Functional Pattern: Chain of Responsibility

- **Chain of Responsibility** *chains the receiving objects and pass the request along the chain until an object handles it.*
  - A **continuation** is a function which indicates how a computation is carried on.
  - For instance, if a function cannot fully compute its result, it can make use of one of its continuations to handle it in some other way.
  - Related to a functional programming technique called **continuation passing style**.

# Functional Type Patterns: Composite and Interpreter

- **Composite:**

```
type component =  
  | Axiom_1  
  | [...]   
  | Axiom_n  
  | Composite of component list
```

- **Interpreter** *defines a representation for the grammar of a language along with an interpreter that uses the representation to interpret sentences in the language.*

```
type expr = Expr of term | Plus of expr * term  
and term = Term of factor | Mult of term * factor  
and factor = Const of float | Factor of expr;;
```

- Relationships with the so-called “**expression problem**” or “**extensibility problem**”.

# OOP and Inductive Type Style

- Some respective properties of OOP and functional type patterns:
  - **Properties of the inductive type style:** more simplicity, full type safety, difficult type extensions, and easier associated function set extensions.
  - **Properties of the OOP counterparts:** more complex, weaker static type safety, easy type extensions, and more difficult function set extensions.

# Functional Design Patterns

- The functional design patterns are **simpler forms of the original OOP design patterns**.
  - ⇒ Functional paradigm can be **included with some benefits into OO languages** (see e.g. FC++, Javascript, Scala).
  - ⇒ Inductive type style can be **directly included into OOP** (see e.g. Scala).
  - ⇒ Inductive types can be **modified to involve more flexibility** (see e.g. *polymorphic variants*).

# “Pure” OOP

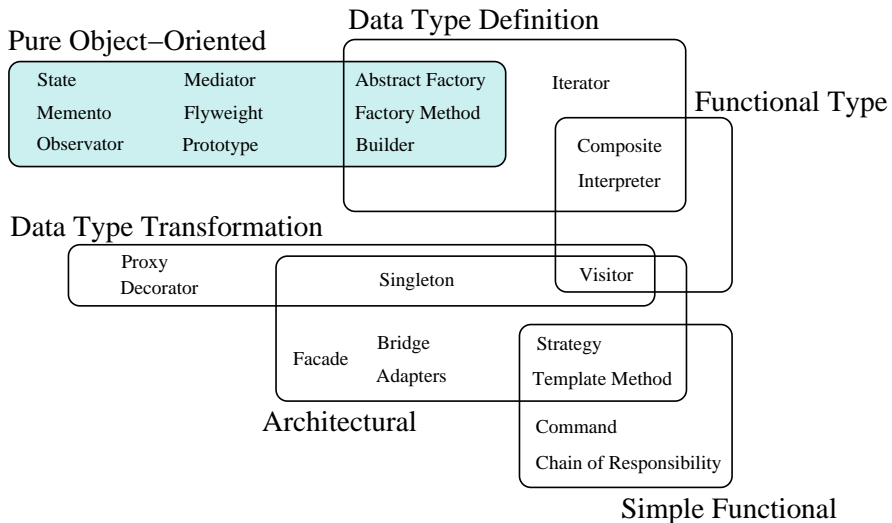
- Some OO design patterns of the GoF Book are **not so easily translated into other paradigms.**
- Indeed, there is at least one feature characteristic to the OO paradigm: first-class citizen values with:
  - **Individual mutable states.**
  - **Possibly complex individual behavior.**

that is, **objects.**

# “Pure” OO Design Patterns

- **State-Oriented Patterns:** OOP patterns addressing managing encapsulated state problems.
- **Object Independent Behavior Patterns:** OOP patterns devoted to building objects of the same type but with different implementations.

# “Pure” OO Design Patterns



# Some Consequences of this Programming Experience

- Are patterns intrinsic to the OOP paradigm? **Only a few of them (at least from the point-of-view of their intents)** but differences wrt. simplicity, encapsulation, type safety, flexibility, extensibility.

# Some Consequences of this Programming Experience

- Are patterns intrinsic to the OOP paradigm? **Only a few of them (at least from the point-of-view of their intents)** but differences wrt. simplicity, encapsulation, type safety, flexibility, extensibility.
- May patterns contribute to the understanding of paradigms? Yes, e.g., **in showing cases where intrinsic *objects* properties are fully exploited**, and e.g., **in showing cases where *functional programming* simplifies OOP.**

# Some Consequences of this Programming Experience

- Are patterns intrinsic to the OOP paradigm? **Only a few of them (at least from the point-of-view of their intents)** but differences wrt. simplicity, encapsulation, type safety, flexibility, extensibility.
- May patterns contribute to the understanding of paradigms? Yes, e.g., **in showing cases where intrinsic *objects* properties are fully exploited**, and e.g., **in showing cases where *functional programming* simplifies OOP.**
- May patterns in one paradigm help in finding patterns in other paradigms? Yes, **in particular in *generic modular programming*.**