

A Multiparadigmatic Study of the Object-Oriented Design Patterns

Ph. Narbel
LaBRI, University of Bordeaux 1
351, Cours de la Libération, 33405 Talence, France
narbel@labri.fr

Abstract. The understanding of programming paradigms has not been fully established yet, though many mainstream languages, e.g. C++, Java, ML, offer more than one paradigm. This paper addresses this understanding problem through a programming experiment: considering the classic object-oriented programming (OOP) design patterns as described in the GoF book [GHJV95], we systematically look at them from the viewpoint of other paradigms, in particular the generic modular and the functional paradigms. The main results of this experiment are: (1) Many OO design pattern intents are meaningful in a more general setting than OOP, and as such they are good candidates for exploring paradigms; (2) Many OO design patterns have counterparts in generic modular programming, but with different properties, in particular with respect to dynamic/static behavior and type safety; (3) Some OOP design patterns can be implemented by using basic functional programming, justifying the idea that functional programming can also be seen as a simplified OOP having its place in a OO language; (4) Some OOP design patterns seem definitely associated with the OOP paradigm, stressing the intrinsic properties of this paradigm.

1 Introduction

A *design pattern* describes a solution that addresses a recurrent design problem in programming. As such, a pattern helps a programmer in building software architectures and in having new ideas. More generally, a pattern explicits a high level of knowledge in a specific area of programming. Although patterns have been mostly recognized and developed for object-oriented programming [GHJV95], they clearly exist in any paradigm, and even in any style or language (see e.g. [AIS77,CS95,Gab96]). This said, some questions arise: 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?

The purpose of this paper is to develop and synthesize some effective answers to the above questions. As a matter of fact, design pattern intents often happen to be more general than specific needs to be found within one paradigm. Also, paradigms sometimes offer similar constructions even if these show different properties, e.g. with respect to simplicity, encapsulation, type-safety, extensibility, etc. In the light of these remarks, the idea here is to revisit the classic OOP

design patterns as described in the GoF book [GHJV95] from a multiparadigmatic viewpoint: Each pattern is systematically looked at, discussed, attempted to be translated into other paradigms than OOP, in particular into *generic modular programming* and into basic *functional programming*.

Modular programming is roughly speaking the use of syntactic units with encapsulation capabilities (see e.g. [Par72,Sco00,SGM02]). This kind of programming gives rise to some comparable software architecture problems as in OOP, in particular when OOP is based on a *class-is-module* model, like in Java or C++: the main architecture structuring component level is the same, and accordingly similar transmission, extension, adaptation problems occur. Of course, these similarities are encountered when module systems are sophisticated enough so as to be able to compete with the specific mechanisms associated with OOP. For instance, Ada or Modula-3 fall into this category, but even more the ML-like languages [Mac86,HP05,Dre05]. In these module systems, modules are typed, they can statically be transformed by using *parameterized modules* – called *ML-functors* –, they can exploit inclusion-based inheritance, they can handle subtype relationships, etc. In other words, these module systems provide the programmer with fully typed *generic modular programming*. Still, in translating OOP techniques into modular programming techniques, object transmission must be replaced by module transmission, dynamic behavior must often be replaced by static behavior, and specific class constructions must be replaced by specific abstract data type constructions. Nevertheless, general software design qualities can be preserved and compared. For instance, signs of this fact can already be observed in C++ when *class templates* are used to obtain more static behavior and type precision, as well as less execution overhead and various optimization possibilities (see e.g. [CE00,Ale01]).

Functional programming is based on functions seen as independent units, implemented by the concept of *closure* (see e.g., [Rea89,FWH92]), that is, encapsulation of code together with an associated private environment. As such, closures can be seen as a special kind of simple *objects*. Thus, when single behavior transmission or modification is involved in some OOP technique, functional programming may offer more straightforward equivalent forms. In other words, functional programming can sometimes be a good ally to simplify some OOP techniques. For instance, signs of this fact can already be observed in C++ in some overloading uses of the function call operator “()” – called C++-*functors* (see e.g. [Cop92,CE00,Ale01]), and discussions about this paradigm relationship already exist (see e.g. [Küh99,SM00,MS04]). On the other hand, typed functional programming like in ML or Haskell offers a kind of safe *union types* called *inductive types* (or *sum types*). These types allow one to define type structures which are similar to class hierarchies. Thus, as in the case of functional closures, these type constructions may sometimes offer convenient comparable programming forms.

Now, looking at each OOP design pattern in [GHJV95] with the two above paradigms in mind will lead us to the following general view and classification of these patterns:

1. (*Generic Modular Design Patterns*). Some OOP design patterns have a direct meaning and translation into generic modular programming:
 - (*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, i.e., **Facade**, **Bridge**, **Adapters**, **Visitor**, **Strategy**, **Template method**, **Singleton**.
 - (*Data Type Definition Patterns*): OOP patterns able to make specific object/data type definitions more adaptable and more composable, i.e., **Composite**, **Interpreter**, **Iterator**, **Abstract Factory**, **Factory Method**, **Builder**.
 - (*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, i.e., **Proxy**, **Decorator**, **Singleton**, **Visitor**.

The rationale of these distinctions is the following: the first subset contains very general patterns which are naturally not bound to any paradigm. The second and third subsets are about data types, which generally take a central role within a paradigm and ask for specific design questions. Defining new types or modifying existing types often make use of different constructions.

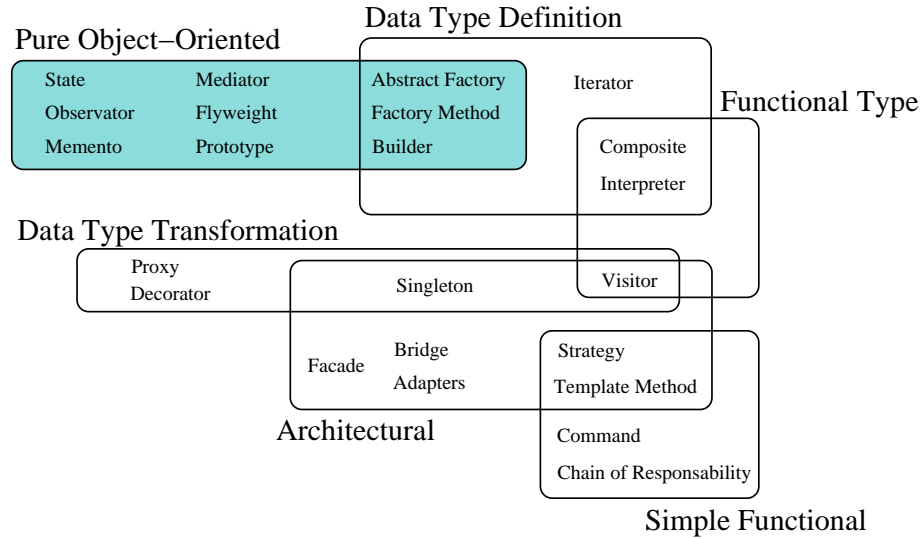
2. (*Functional Design Patterns*). Some OOP design patterns are close to plain functional programming techniques because of closures and inductive types:
 - (*Simple Functional Patterns*): OOP patterns addressing single method encapsulation/transmission problems, i.e., **Command**, **Strategy**, **Template Method**, **Chain of Responsibility**.
 - (*Functional Type Patterns*): OOP patterns addressing union-like and recursive type definition problems, i.e., **Composite**, **Interpreter**, **Visitor**.

This last subset consists of patterns dealing with a very sensitive programming matter, that is, balancing between flexibility, staticness and encapsulation. These patterns have already been most debated, in particular under the name of “*expression problem*” (see e.g. [Wad98,App97,Bru03,Tor04,ZM05]).

3. (*“Pure” OO Design Patterns*). Some OOP patterns are not easily translated into other paradigms. This is the case when *objects* takes their full part into these patterns, that is, patterns specifically dealing with autonomous values with individual mutable states and behavior:
 - (*State-Oriented Patterns*): OOP patterns addressing managing encapsulated state problems, i.e., **State**, **Memento**, **Mediator**, **Observer**, **Flyweight**.
 - (*Object Independent Behavior Patterns*): OOP patterns devoted to building objects of the same type but with different implementations, i.e., **Prototype**, **Abstract Factory**, **Factory Method**, **Builder**.

These “irreducible” patterns allow one to more precisely understand what is essential to the OO paradigm. No surprise, the characteristic features are to be found in the objects themselves, rather than in the usual OOP facilities like classes, interfaces, or inheritance.

The next figure gives a synthetic view of this multiparadigmatic way of looking at the OOP design patterns. There are 17 patterns out of the 23 presented in [GHJV95] which can be seen as being not exclusive to OOP. Overlapping occurs since some patterns can be associated with more than one of the above pattern categories, and can have more than one translation:



Note that this kind of programming experiment has already been undertaken (see e.g. [Nor96,BLR98]). However, we stress here on two ideas: First, most classic OOP patterns in [GHJV95] mainly address problems raised when dealing with encapsulated units. Most of them provide techniques to ensure flexibility while preserving encapsulation. When no encapsulation takes place, these patterns may indeed become trivial or even irrelevant. An example is given by the **Visitor** pattern. Second, most of the ideas behind the OOP patterns in [GHJV95] are general enough so as not to be interpreted as missing language constructs. Of course, this could be sometimes a matter of interpretation. For instance, **Singleton** can be seen as an emulation of a plain module construct, but it can also be interpreted as a technique for using a class and enforcing a specific behavior over its instantiation.

In the following pages, we precise and justify the above viewpoint. Our main translation language will be ML (its dialect OCaml [LDG⁺06]). In fact, OCaml also offers a full object system, but according to our programming experiment, we restrict ourselves to generic modular and functional programming capabilities. Note that most of the generic modular programming translations could also have been discussed in Ada95 or Modula-3.

2 Design Patterns and Generic Modular Programming

Let us first recall some important differences between OOP and modular programming by stating their main respective advantages:

- *Advantages of OOP*: dynamic variation/choice of encapsulated code, and exploitation of object type compatibilities.
- *Advantages of modular programming*: static verification and type safety, and more optimization possibilities.

OOP promotes type compatibilities between object types, a very powerful tool to obtain flexibility at run-time. However, static verification and type precision are sometimes difficult to ensure. Well-known intrinsic problems occur for instance when n -ary methods need coherent object arguments, e.g. in *comparison methods* (see e.g. [BCC⁺95,CMLC06]). Also, execution overhead may come from *late bindings* (see e.g. [Cra02]), even if this effect must not be overestimated.

This said, many similarities can be observed between OOP and modular programming, particularly when classes are the main encapsulating construction, like in Java or C++, and when sophisticated typed modular generics is available. The following points make this fact more precise, being the basics to understand how some OO design patterns may have admissible counterparts in generic modular programming:

1. (*Interfaces*). The relationship *interface/implemented class* is similar to the relationship *module signature/module*. In particular, ML module signatures are interfaces which can be instantiated by more than one module (an important property from the point-of-view of parameterized modules whose parameters are typed by signatures). ML-like module signatures and modules take the following form 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
```

2. (*Inheritances*). Inclusion is a simple way of looking at inheritance. Module inclusion has indeed some similarities to class inheritance:

```
module M1 = struct [...] end
module M2 = struct include M1 [...] end (* M1 is copied into M2 *)
```

Interface inheritances are similar to signature inclusions as well:

```
module type S1 = sig [...] end
module type S2 = sig include S1 [...] end (* S1 is copied into S2 *)
```

The above similarities are deeper than just syntactic sugar constructions: the derived signature **S2** is *compatible* with **S1**, that is, if a module **M2** implements **S2**, it also implements **S1** (since **M2** implements everything in **S1**). For the same reason, if the header of a parameterized module has the form **F(X:S1)**, then **F(M1)** and **F(M2)** are both valid applications of **F**. Moreover, some *overrides* may be available when multiple definitions are allowed: the

semantics is generally to consider only the last occurrence of a definition (with no *late binding* capabilities). Also, very naturally in generic modular programming, such inheritance can be generalized into *generic inheritance*:

```

module F (X : S1) = struct
  include X
  [...]
end
module M2 = F (M1)

```

- (*Delegations*). The element E in an object `obj1` is said to be *delegated* to an object `obj2` if `obj2` is an instance variable of `obj1` and if E is dependent on a call to some element in `obj2` (see e.g. [Cra02,GHJV95]). This idea may be translated into modular programming when nested modules are allowed: the element E of a module $M1$ is said to be *delegated* to a module $M2$, if $M2$ is an inner-module of $M1$ and if E is dependent on a call to some element in $M2$. For instance, modular delegation in ML can be specified and implemented:

```

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

```

Of course, the assignment of `Imp` by some module must be static (the main difference with OO delegation). In this respect, 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

```

Staticness makes run-time component composition impossible. Nevertheless, when object delegation is considered as a technique to improve the global qualities of an architecture, this kind of generic modular delegation can play a similar role.

- (*Abstract classes*). An abstract class C is a partially defined class such that all its derived classes include the implemented elements of C . Although no such mechanism exists in modular programming, a similar code factoring effect can be obtained through generic inheritance:

```

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

```

Each time `AbstractComponent` is applied to some X , it generates a new derived modular instance of $S2$ which shares the implementation of g .

Note that the two main constructions exploited in OO design patterns, i.e. delegations and abstract classes, are translated into a single construction in generic modular programming, i.e. parameterized modules with inner/included modules. In fact, the OO constructions have the same general use, that is, code factoring (see e.g. the discussion about **Strategy** and **Template Method** in [Mar03]). Moreover, the two corresponding applications of generics have different intents, though with static flavor in both cases. Summing up the above relationships:

Object-Oriented:	Generic Modular:
interface/implemented class	↔ module signature/module
inheritance	↔ inclusion
delegation	↔ nesting + generics
abstract classes	↔ inclusion + generics

Let us now present in more detail the subsets of OOP patterns which have translations into generic modular programming.

2.1 Architectural Design Patterns

First, there are several OO design patterns in [GHJV95] which solve general component-based software architecture problems and, as such, may have a quite satisfactory translation into generic modular programming:

- **Facade** is a very general design pattern which can be applied to any kind of architecture: *Facade defines a higher-level component that makes a set of components easier to use.* This pattern has clear counterparts in plain or generic modular programming.
- **Bridge** is one of the most general design patterns improving an architecture: *it decouples an abstraction from its implementation so that the two can vary independently.* In other words, **Bridge** extracts specific elements E out of a unit component C in order to make C independent from E . In OOP, reconnecting E and C is based on delegation. **Bridge** is indeed the canonical pattern showing the benefits of delegation: easy selection, separation of concerns, extensions with limited growth of the overall number of pairwise dependent components. As described above, this kind of delegation can be translated into generic modular programming. For instance, let us use an example based on a window type [GHJV95]:

```

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

module Sys_0Graphics = [...] (* OCaml Graphics dependent *)
module Sys_Tk = [...] (* Tk dependent *)
module Win_0Graphics = Win (Sys_0Graphics)
module Win_Tk = Win (Sys_Tk)

```

As expected, a window type built after `Win` does not commit to a concrete implementation of a graphic system. Moreover given a new implementation of `GRAPHIC_SYS`, it is easy to get a new window type:

```
module Sys_OpenGL = [...] (* OpenGL dependent *)
module Win_OpenGL_Ext = Win (Sys_OpenGL)
```

Note that **Bridge** has been discussed for C++ templates (see e.g. [CE00,VJ03]).

- **Strategy** is also a pure application of the delegation idea but with a less general intent than **Bridge**: *it lets an algorithm vary independently from clients that use it*. When restricted to single operations, it also has a translation in functional programming (see below p. 13). **Template method** is very similar to **Strategy**: *it lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure*. This pattern relies on abstract classes instead, but can be translated into generic modular programming as well.
- **Object Adapter** *converts an interface of a component into another interface clients expect*. This pattern relies on delegation too. **Class Adapter** has the same intent but instead relies on class derivation. Accordingly, these two patterns can be translated into generic modular programming. For **Class Adapters**, one may even consider multiple interface inheritance so that the adapter type may be compatible with the adaptee and the target types:

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

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

Any module produced by `Adapter` will be an implementation of `TARGET`, still remaining an implementation of `ADAPTEE`. Note that like **Bridges**, adaptation has also been discussed for C++ templates (see [CE00]).

- **Visitor** can also be considered as a general architectural design pattern since its intent is mainly to make an encapsulating unit hierarchy more extensible: *it lets you define a new operation without changing the classes of the elements on which it operates*. Without the **Visitor** idea, each class has to be derived separately, making the overall architecture size grows out of control. Similar undesirable effects occur in any paradigm where encapsulating units may be related and multiple. For instance, in ML-like modular programming, consider the following programming situation: assume that `M1`, `M2`, ..., `Mn` are modules implementing the same signature `S`. For any extension of `S`, each module `Mi` would need an individual extension, and the same is true for the parameterized modules taking arguments of type `S`. Generic modular programming **Visitors** are possible using the same idea as in the OOP version: one just transmits modular **Visitors** as arguments to the visited parameterized components. Note that when explicitly applied to data type extensions, **Visitor** may also be seen as a type transformation pattern (see below p. 11).

2.2 Data Type Definition Design Patterns

Some design patterns in [GHJV95] are related to defining object types with specific properties related to composition, flexibility and extensibility. These patterns may have translations into generic modular programming when considering the world of *abstract data types* (ADT):

- **Composite** allows one to *compose objects into tree structures to let clients treat individual objects and compositions of objects uniformly*. A single type can thus represent primitives as well as their compositions through recurrent definition calls and containers. In the case of usual modular ADTs, this type flexibility is not possible: type compatibility does not *a priori* hold between the instances/values of different ADTs (see e.g. [Nar05]). Nevertheless, producing composite ADTs from other ADTs remains possible in modular programming. For instance, one can generically construct ADTs made of lists of other ADTs:

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

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

Note here that the parameterized module **Composite** is iterable since it is of type `S_TYPE -> S_TYPE`. However, as remarked above, different implementations of `S_TYPE` will be pairwise incompatible. This seems to be the price to pay for type safety. Indeed, the original OO **Composite** can be unsafe, e.g. if comparison methods exist between different forms of a **Composite** (see discussion in p. 5). Note that as type definition, **Composite** has also possible translations into typed functional programming (see below p. 13).

- **Interpreter** can be seen as a specialized **Composite**: *Given a language, it defines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language*. Non-terminals are represented by abstract classes, and terminals are derived classes. Since a grammar is often recursive, we get a composite-like organization.
- **Iterator** provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. This pattern consists of independently defining different iterator types able to exploit different data structure representations. Generic modular programming can be used to obtain this separation of concerns. For instance, here is a simple case for lists:

```
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;;
```

An explicit connection can then be set between these two signatures:

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

Next, different kinds of list iterators may be implemented:

```

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;;

```

- **Abstract Factory** provides an interface for creating families of related or dependent objects without specifying their concrete classes. This pattern is based on class derivation and as such, it can also have a generic modular programming translation. However, if the use of this pattern is explicitly to mix up objects with different implementations, this pattern keeps its OO flavor (cf. below p. 15). Generally, **Abstract Factory** is implemented with **Factory Methods** which lets a class defer instantiation to subclasses, and which are just specialized **Template methods**.
- **Builder** separates the construction of a complex object from its representation so that the same construction process can create different representations. This pattern has the same status as **Abstract Factory**.

2.3 Data Type Transformation Design Patterns

Among the design patterns in [GHJV95], some induce a global and uniform transformation of every object produced by a class or a set of classes. The essence of these patterns consists of transforming existing types and their values. Accordingly, they can be applied to the ADT world too:

- **Proxy** provides a surrogate or placeholder for another object to control access to it. This pattern amounts to uniformly modifying or controlling the behavior of the values of a type. For instance, a **virtual Proxy** creates expensive objects on demand only, that is, on their first access. This behavior is obtained by including some *laziness* in the object instantiation. Here is a straightforward way of obtaining this property with generic modular programming (**Lazy** is a module in the standard library of OCaml handling laziness):

```

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

```

Every implementation of **SUBJECT** becomes lazy after the application of **Proxy**. Note that the two properties of the original pattern have been preserved: the interface of **Proxy** is identical to the “real subject” **X** (so that substitution can take place), and **Proxy** explicitly refers to the real subject.

- **Decorator** (or **Wrapper**) *attaches additional responsibilities to an object dynamically*. Based on a recurrent delegation to a superclass **C**, this pattern allows one to compose different actions coming from possibly different objects, all being of type defined by **C**. As a consequence, one may transparently enrich the behavior of a given object. Generalizing this idea, and giving up the dynamical possibilities, a **Decorator** can be seen as a construction over a type which enriches its behavior without changing its interface (ensuring transparency and decoration composition). Thus, the same generic modular programming technique as used in **Proxy** can be applied. Note that **Decorators** have also been discussed for C++ templates (see e.g. [CE00]).
- **Singleton** is a simple design pattern whose intent is *to ensure that a class only has one instance, and provide a global point of access to it*. In fact, **Singleton** amounts to transforming a class into a module: unique in a program, its single instance centralizes some data and behavior. From this point-of-view, modular programming would seem to need no new particular pattern since every module is necessarily unique [BLR98]. However, **Singleton** often acts as an instance of a type. In other words, its intent can also be interpreted as a control over the global behavior of the instantiation of a type: one unique instance must be created. As a result, the same generic modular technique as for **Proxy** can be applied.
- **Visitor** (see p. 8) is often used to extend the possibilities of a type. Therefore, it can also belong to the data type transformation patterns.

3 Design Patterns and Functional Programming

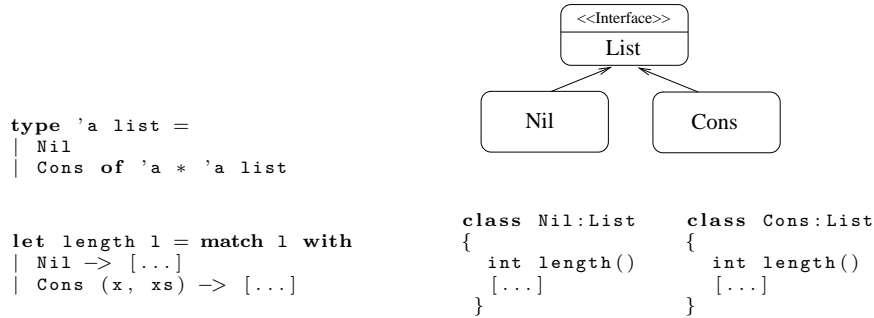
3.1 Comparing OOP and Functional Programming

At first, OOP seems quite different from functional programming. However, there are at least two similarities between these two paradigms:

1. (*Functional Closures*). One comparison point between OOP and functional programming lies in the structure of their characteristic values. Indeed, objects contain states, i.e. private environments as well as (possibly implicit) references to methods; functions in functional programming are represented by *closures*, that is, private environments associated with function code, making possible for them to be first-class citizen values. In other words, closures are a special kind of simple *objects*. Note that when a functional language includes imperative features, closures may be used to emulate object systems (see e.g. [FWH92]). As a result, when the implementation of some OO design pattern relies on single method encapsulation or transmission, basic functional programming techniques can be applied instead. This paradigm shift may simplify the architecture by avoiding some class or object constructs. Note however that even if closures are encapsulating units, they are *a priori* not at the same structuring level as modules, classes or objects. The following possible pattern equivalences must therefore be understood in a more general setting than just as a one-to-one mapping from classes/objects

to functions. Occurrences of first-class functions have to be inserted into the code.

2. (*Inductive Types*). There exists another connecting point between OOP and functional programming. Indeed, typed functional programming like in ML or Haskell provides the programmer with a kind of *union types* called *inductive types*. Instances of these types take the form of function compositions, giving them their functional programming flavor. These types define structures which are similar to class hierarchies. For instance, a linked list structure may be based on the following similar definitions (see e.g. [PJ98]):



As illustrated in this example, these two constructions induce two implementation styles of collecting data cases and function cases. This is a well-known fact, discussed e.g. in [Rey75,FF99]. The inductive type style generally implies full type safety, difficult type extensions, and easier associated function set extensions. OOP style generally implies weaker type safety, easy type extensions, and more difficult function set extensions. Because of this uncomfortable balance, attempts were made to combine the advantages of both styles: some recent languages include some of the inductive type style into OOP (see e.g. [Ode07]). Also, inductive types constructions were designed to involve more flexibility (see e.g. [Gar00]).

Let us now present in more detail the subsets of OOP patterns which have translations into functional programming.

3.2 Simple Functional Patterns

First, some design patterns in [GHJV95] are based on encapsulating and transmitting methods, and may be replaced by functional closures:

- **Command** *encapsulates a request as an object, thereby letting you parameterize clients with different requests.* A “request” is here just an operation considered as a first-class value. This pattern is clearly a way of emulating functional closures with objects (see e.g. [Küh99,SM00,MS04]). Indeed, a **Command** generally involves one single method – classically called **execute** – with a simple environment. Recall that C++ offers a specific construction to obtain objects resembling to functional closures by overloading the function call operator “()”. Using these C++-functors for **Commands** is not as convenient as using functional closures (see e.g. [Ale01]).

- **Strategy** lets an algorithm vary independently from clients that use it. Based on object delegation, we already have seen that **Strategy** can be implemented by generic modular programming (cf. p. 8). However, being specifically associated with algorithms, functional decoupling can be very localized and reduced to single operations. In this case, functional programming can be fruitfully applied, preserving the dynamic properties of the OO pattern. For instance, in [Mar03], one can find **Strategies** very similar to some functional programming examples given in [CMP00,Nar05]. The same is true for **Template method**, and thus to some extent for **Factory Methods** too.
- **Chain of Responsibility** is a pattern that *chains the receiving objects and pass the request along the chain until an object handles it*. This pattern has a relationship with a functional programming technique called *continuation passing style* (see e.g. [SF90,FWH92,App97,FS99]). Indeed, a *continuation* is a function which indicates how a computation is carried on. As first-class functions, continuations can be passed as arguments or built within the initial function in order to have an effect on the overall computation. 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.

3.3 Functional Type Patterns

We have seen that inductive types show direct similarities with class hierarchies. In fact, any OO class-based architecture could be translated into a set of inductive types. However, the encapsulation level and the composition mechanisms of inductive types are too weak, and most OOP design patterns could not be satisfactorily be represented by inductive types. Nevertheless, there are at least two design patterns in [GHJV95] which are specifically about defining recursive types, and which have a corresponding natural form using inductive types:

- **Composite** (see above, p. 9) can readily be translated into an inductive functional type by the following kind of definitions:

```

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

```

Compared to the generic modular **Composite**, this functional version has less encapsulating power, but allows one to compose the different cases. As already said, compared to the OOP initial pattern, it induces more type safety, but difficult case extensions.

- **Interpreter**, seen as a specialized kind of **Composite** (cf. p. 9), may also be implemented by using inductive types. The dilemma between the different existing implementations of **Interpreter** has been much debated in particular under the name of the “*expression problem*” or “*extensibility problem*” [Wad98,App97,Bru03,ZM05,CMLC06]. Note that the different versions of **Visitors** are also related to this dilemma (see e.g. [PJ98]).

4 Design Patterns and OO Programming

4.1 Irreducible Properties of OOP

Some OO design patterns in [GHJV95] 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 and possibly complex individual behavior, that is, *objects*. Note that we just have discussed about functional closures as being objects, but these are only specialized and restricted kind of objects. In modular programming, module with private states can also be found when static permanent definitions are available, e.g. in C, C++, ML. This is a common technique in ML (see e.g. [Dre05]). For instance:

```
module M = struct
  let state = ref [...]
  let set_state st = (state := st)
  let f () = [...] !state [...]
  [...]
end
```

However, modules are generally not first-class citizens, definitely reducing their possibilities. Note that this situation is not inherent to modular programming, e.g., Moscow ML [RRS00] where almost first-class modules are available. Evaluation of the power of these modules has still to be done. By contrast, OCaml includes a complete OO system [LDG⁺06], and as soon as first-class components are needed, a programmer shifts opportunistically to OOP.

Therefore, according to the object characteristics, there are at least two subsets of design patterns which appear irreducible to OOP. We describe them next.

4.2 State-Oriented OO Design Patterns

A first kind of design patterns in [GHJV95] stresses the role of individual and autonomous states for objects:

- **State** allows an object to alter its behavior when its internal state changes. Even if this pattern is based on object delegation, its dynamic behavior intent is too important to be dropped.
- **Memento**: Without violating encapsulation, it captures and externalizes an object's internal state so that the object can be restored to this state later.
- **Observer** defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The relationships between states is at the root of this design pattern. However, some functional programming can be sometimes applied to simplify it. Indeed, this pattern consists of organizing a set of “observers” bound to a “subject”. When the state of the subject changes, update functions of the observers are called. If first-class functions are available, dealing with the update functions can be made easier [SM00].
- **Mediator** defines an object that encapsulates how a set of objects interact.
- **Flyweight** uses sharing to support large numbers of fine-grained objects efficiently.

4.3 Object Independent Behavior Design Patterns

A second kind of design patterns in [GHJV95] stresses the implementation independence of each object of the same type. In other words, these patterns allow one to make easier substitutions of objects with different behavior:

- **Abstract Factory**, **Factory Method**, and **Builder** have already been discussed as data type definition design patterns, and as having possible counterparts for modular ADTs (cf. p. 10). However, when created objects are explicitly expected to be of different implementation while being substitutable, the modular versions would be unable to express this property.
- **Prototype** *specifies the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.* This pattern can only be meaningful when objects have independent implementation. This pattern is an emulation of the idea of OO languages based on prototypes, e.g. Self [US87].

5 Discussion

This proposed classification of the OO design patterns in [GHJV95] (see Fig. p. 4) may probably be discussed and refined further. Still, one can already make several firm observations about this multiparadigmatic study of the OO patterns:

1. Many classic OO design pattern intents are meaningful in a more general setting than OOP. The knowledge we already have about them justifies the idea they are good candidates for comparing paradigms (and not only the **Interpreter** or the **Visitor**, already recognized as such). In particular, we have discussed differences with respect to encapsulation, type safety, flexibility, and extensibility.
2. Many comparison points are possible between OOP and generic modular programming, in particular from the point-of-view of *interfaces*, *inheritance*, and *delegation*. The important differences lie in the dynamic or static-oriented behavior, and thus in terms of type information precision too (see e.g. the case of **Composite** in p. 9). Nevertheless several OO design patterns have been satisfactorily translated into generic modular programming. As a by-product, these translations provide some candidates of patterns in generic modular programming, a context where they have not often been discussed (see however [BHL01,Nar05] for ML, and [Ale01] for C++).
3. Some OOP design patterns are sometimes related to single operation encapsulation and transmission. When this is the case, functional programming can be used to simplify these patterns. Indeed, functional closures are just light specialized objects, showing that the functional paradigm has its place within the OO paradigm (a fact already discussed, e.g. [Küh99,SM00,MS04]).
4. Some OO design patterns seem definitely associated with the OO paradigm, stressing its intrinsic properties, in particular with respect to the *object* concept. Each time a design pattern is based on first-class values with individual states and behavior, it probably belongs to the OO world in an exclusive way.

References

- [AIS77] Ch. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*. Oxford Univ. Press, 1977.
- [Ale01] A. Alexandrescu. *Modern C++ Design. Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
- [App97] A. W. Appel. *Modern Compiler Implementation in ML, Java, C*. Cambridge Univ. Press, 1997.
- [BCC⁺95] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Objects Group, G. T. Leavens, and B. Pierce. On binary methods. In *Theory and Practice of Objects Systems*. John Wiley and Sons, Inc., 1995.
- [BHL01] E. Biagioni, R. Harper, and P. Lee. A network protocol stack in Standard ML. *Higher Order Symbol. Comput.*, 14(4):309–356, 2001.
- [BLR98] G. Baumgartner, K. Läuffer, and V.F. Russo. On the interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Purdue University, 1998.
- [Bru03] K. Bruce. Some challenging typing issues in object-oriented languages, 2003.
- [CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison Wesley, 2000.
- [CMLC06] C. Clifton, T. Millstein, G.T. Leavens, and C. Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.
- [CMP00] E. Chailloux, P. Manoury, and B. Pagano. *Developing Applications with OCaml*. O’Reilly, 2000.
- [Cop92] J. O. Coplien. *Advanced C++, Programming Styles and Idioms*. Addison-Wesley, 1992.
- [Cra02] I. Craig. *The Interpretation of Object-Oriented Programming Languages*. Springer-Verlag, 2002.
- [CS95] J. O. Coplien and D. C. Schmidt. *Pattern Languages of Program Design*. Addison-Wesley, 1995.
- [Dre05] D. Dreyer. *Understanding and Evolving the ML Module System*. PhD thesis, CMU, May 2005. Technical Report CMU-CS-05-131.
- [FF99] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN Intl. Conference on Functional Prog. (ICFP ’98)*, volume 34(1), pages 94–104, 1999.
- [FS99] M. Felleisen and A. Sabry. Continuations in programming practice: Introduction and survey. Rice University and University of Oregon, 1999.
- [FWH92] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, 1992.
- [Gab96] R. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford Univ. Press, 1996.
- [Gar00] J. Garrigue. Code reuse through polymorphic variants. In *Workshop on Foundations on Software Engineering*, 2000. Sasaguri, Japan.
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
- [HP05] R. Harper and B. C. Pierce. Design considerations for ML-style module systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 293–345. MIT Press, 2005.
- [Küh99] Th. Kühne. *A Functional Pattern System for Object-Oriented Design*. Verlag Kovac, 1999.

- [LDG⁺06] X. Leroy, Doligez D., J. Garrigue, Rémy D., and Vouillon J. *The Objective Caml System. Documentation and User's Manual*. INRIA, December 2006.
- [Mac86] D. MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13rd ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Languages*, pages 277–286, 1986.
- [Mar03] R. C. Martin. *Agile Software Development (Principles, Patterns and Practices)*. Prentice Hall, 2003.
- [MS04] B. McNamara and Y. Smaragdakis. Functional programming with the FC++ library. *J. Funct. Program.*, 14(4):429–472, 2004.
- [Nar05] Ph. Narbel. *Programmation fonctionnelle, générique et objet (Une introduction avec le langage OCaml)*. Vuibert, Paris, 2005.
- [Nor96] P. Norvig. Design patterns in dynamic programming. Technical report, Harlequin, Inc., 1996.
- [Ode07] M. Odersky. *The Scala Language Specification, Version 2.5*. Programming Methods Laboratory, EPFL, 2007.
- [Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [PJ98] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Computer Software and Applications Conf., 1998. COMPSAC '98*, pages 9–15, 1998.
- [Rea89] Ch. Reade. *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley, 1989.
- [Rey75] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages*, pages 157–168. INRIA, IFIP Working Group 2.1 on Algol, 1975. Schuman, A. editor.
- [RRS00] S. Romanenko, C. Russo, and P. Sestoft. *Moscow ML Owner's Manual*, June 2000.
- [Sco00] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.
- [SF90] G. Springer and D.P. Friedman. *Scheme and the Art of Programming*. MIT Press, 1990.
- [SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software*. Addison-Wesley, 2002.
- [SM00] Y. Smaragdakis and B. McNamara. Bridging functional and object-oriented programming, 2000. Georgia Tech CoC Tech. Report 00-37.
- [Tor04] M. Torgersen. The expression problem revisited (four new solutions using generics). In *ECOOP 2004*, LNCS 3086, pages 123–143. Springer, 2004.
- [US87] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Notices*, 22(12):227–242, 1987.
- [VJ03] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison-Wesley, 2003.
- [Wad98] Ph. Wadler. The expression problem. Java Genericity Mailing List, 1998.
- [ZM05] M. Zenger and Odersky M. Independently extensible solutions to the expression problem. In *FOOL'05, ACM SIGPLAN Intl. Workshop on Foundations of OO Languages*, 2005.