Les modèles classiques de conception objet vus par la programmation ML

Ph. Narbel

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

Résumé

Les langages modernes comme C++, Java, ML, proposent plusieurs paradigmes de programmation à la fois. Pourtant, la compréhension de ce qui fait les caractéristiques respectives de ces paradigmes n'est pas encore complètement connu, ni compris. Cet article donne quelques éléments de réponse à ce problème par le biais de l'expérience de programmation suivante : on considère les modèles de conception (*design patterns*) classiques de la programmation objet comme décrits dans [GHJV95], et on tente de les traduire dans leur intention initiale par les moyens standards offerts par la programmation ML, c'est-à-dire la programmation fonctionnelle et surtout la programmation modulaire générique (l'utilisation des foncteurs). Le résultat de cette expérience permet alors de distinguer : (1) des modèles de conception adaptés à la programmation modulaire générique ; (2) des situations où la programmation fonctionnelle apparaît comme une simplification de la programmation orientée objet ; (3) des conditions dans lesquelles la programmation objet est exploitée pour ses spécificités propres.

1. Introduction

Un *modèle de conception* (*design pattern*) décrit et qualifie une solution générale à un problème récurrent de conception lors du développement d'un logiciel. Un modèle de conception aide non seulement le programmeur à produire un logiciel de qualité, mais il est aussi la marque d'une compréhension avancée du problème qui l'a suscité. Si les modèles de conception ont surtout été développés en programmation orientée objet à un niveau « micro-architectural » (c'est-à-dire à un niveau d'interaction n'incluant que quelques classes ou objets) [Gam92, GHJV95, Pre95, CS95], ils existent *a priori* à n'importe quel niveau de conception et dans n'importe quel paradigme ou langage (cf. par ex. [Gab96] et les actes des conférences *PLoP – Pattern Language of Programs*).

Dans cet article, nous décrivons les résultats d'une « expérience programmatoire » qui consiste à considérer tous les modèles de conception orientés objet décrits dans l'ouvrage désormais classique de Gamma et al. [GHJV95], et d'en tenter une traduction en programmation ML au travers de la programmation fonctionnelle et de la programmation modulaire générique (c'est-à-dire l'utilisation des *modules paramétrés* ou *foncteurs*). Il se trouve en effet que les besoins qui mènent aux modèles de conception sont parfois issus de problèmes généraux de programmation et non la conséquence des spécificités d'un paradigme. D'autre part, les points communs des paradigmes peuvent entraîner des modèles de conception similaires, mais néanmoins doués de caractéristiques propres, liées à leur simplicité, à leur sûreté de typage ou à leur extensibilité. Par exemple, la *programmation modulaire* induit des problèmes d'architecture équivalents à la programmation orientée objet lorsque cette dernière est fondée sur des « classes-modules » (comme par ex. en C++ et Java). Le niveau d'encapsulation y est en effet comparable : on associe des données et des fonctions, et les problèmes d'extension et de modification y sont sensiblement les mêmes (cf. par ex. [SGM02]). Lorsqu'un système de modules comme celui de ML inclut des mécanismes d'héritage et des moyens de paramétrage, les ressemblances des techniques utilisées en deviennent d'autant plus probantes. De même, la

programmation fonctionnelle fonde ses mécanismes de base sur la notion de *fermeture*, c'est-à-dire sur l'encapsulation d'une fonction et d'un environnement associé, et donc sur des *objets* particuliers (cf. par ex. [FWH92]). Ce rapprochement *fermeture/objet* permet de comprendre pourquoi des techniques de programmation orientée objet peuvent avoir parfois leur pendant en programmation fonctionnelle.

Les résultats de l'expérience de programmation que nous présentons ici seront donc de plusieurs ordres : tout d'abord, nous serons à même de présenter des modèles de [GHJV95] qui ont un sens en programmation modulaire générique, par ex. le **Pont**, l'**Adaptateur**, la **Façade**, le **Visiteur**, la **Procuration**, le **Singleton**, le **Décorateur**, et l'**Itérateur**. Leurs traductions permettent d'illustrer des situations caractérisées d'utilisation de foncteurs, et de déterminer des versions de ces modèles directement adaptés à la programmation modulaire générique. Deuxièmement, certains des modèles de conception classiques se trouvent être clairement des émulations de la programmation fonctionnelle, en particulier la **Commande**, la **Stratégie**, le **Patron de méthode**. Il est donc possible d'illustrer concrètement le lien annoncé ci-dessus entre les paradigmes objet et fonctionnel. Un modèle – l'**Interpréteur** – sera également discuté dans ce cadre puisqu'il possède une forme naturelle basée sur les *types inductifs*. Troisièmement, certains modèles de conception « résistent » à leur traduction en programmation fonctorielle ou foncionnelle. En effet, un paradigme privilégie des techniques de programmation qui peuvent évidemment mener parfois à des modèles de conception plus spécifiques. Par exemple, lorsque l'autonomie et la diversité d'implémentation des objets sont à la base d'un modèle de conception, ce dernier ne se traduira que laborieusement dans un autre paradigme. C'est le fait par exemple de l'**État**, de l'**Observateur**, du **Mémento**, du **Médiateur**, de la **Chaîne de responsabilité**, ou de la plupart des modèles de création d'objets. Ces modèles peuvent donc mener à mieux circonscrire et à illustrer les spécificités de la programmation objet.

Ainsi, cette étude permet d'obtenir quelques arguments concrets en faveur de l'existence de langages à paradigmes multiples, et en particulier, en faveur de la richesse des constructions que proposent les langages ML comme OCaml, SML ou Moscow ML. Soulignons à ce titre que les traductions des modèles de conception n'utiliseront ici que le langage ML de base. Ainsi, même si elles sont illustrées en OCaml, nous n'exploiterons pas les possibilités de programmation objet de ce langage. Dans le cas contraire, l'expérience de programmation en deviendrait différente (elle reste d'ailleurs à faire, cf. par ex. l'**Observateur** traité dans [Rém05]). D'autre part, nous n'utiliserons en général pas les techniques de programmation ML qui exploitent les *modules avec états internes* en tant qu'émulations de la notion d'objet (cf. [Pau97, Dre05]). Ces techniques se trouvent utiles dans des cas assez limités, et leur développement se trouve surtout justifié lorsque les modules sont eux-mêmes de *première classe* (cf. Moscow ML). L'expérience de programmation en deviendrait alors également différente (et elle reste également à faire). Notons finalement que les descriptions que nous donnerons de chacun des modèles de conception considérés seront évidemment succinctes et se résumeront à en rappeler l'*intention*. Le lecteur est invité à se référer à [GHJV95] et à ses exégètes pour plus de précisions.

2. Modèles et programmation modulaire générique

Dans cette section, nous présentons des éléments de comparaison entre programmation modulaire générique et programmation orientée objet. Ensuite, nous introduisons les modèles de conception de [GHJV95] qui induisent des modèles équivalents en programmation modulaire générique.

2.1. Programmation objet et programmation modulaire générique

Tout d'abord, rappelons certaines différences importantes entre programmation modulaire de base et programmation orientée objet, en particulier celles qui concernent leur rapport au typage. La première propose généralement des moyens pour exprimer des abstractions de type susceptibles de mener à une analyse statique de type rigoureuse. Au contraire, la programmation orientée objet

privilégie une flexibilité de typage réalisée par des relations naturelles de compatibilité entre les types d'objets, exploitables dynamiquement. En cela, ce style de programmation est souvent tenu de faire des compromis à propos de la précision de typage. C'est par exemple le cas pour les méthodes n-aires agissant sur des objets aux types supposés plus cohérents que la compatibilité n'est capable de l'assurer, par ex. les méthodes de comparaison (cf. [BCC+95]). Ceci dit, il existe des similarités entre la programmation modulaire et la programmation orientée objet, en particulier lorsque les classes se trouvent être les encapsulateurs principaux dans un langage orienté objet (comme par ex. C++ et Java, et contrairement par ex. à CLOS) :

(1) Le rapport *classe abstraite/classe implémentée* est comparable au rapport *signature/module*.

(2) L'*héritage de classes* est comparable à l'« *héritage de modules* », i.e. un mécanisme concrétisé par des facilités de recopie d'un module dans un autre. En particulier, OCaml permet l'*inclusion directe* :

```
module M1 : S1 = struct ... end
module M2 : S2 = struct include M1 ... end
```

De même, l'*héritage d'interfaces* est comparable à l'« *héritage de signatures* », également concrétisé par l'*inclusion directe* en OCaml :

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

En ML, ce parallèle est d'ailleurs plus profond qu'il n'en a l'air : la signature `S2` est en effet *compatible* avec `S1`. Ainsi, si `M2` satisfait `S2`, il satisfait aussi `S1`. De même, si un foncteur possède pour entête `F(X:S1)`, alors `F(M2)` est une application valide de `F`.

Ajoutons à cela qu'un équivalent de la notion de *redéfinition* (par *shadowing*) peut être exploitée dans le cadre modulaire. En ML, les définitions multiples sont autorisées au sein d'un module (seule la dernière est prise en compte). Il est donc possible de redéfinir des éléments issus d'une inclusion.

D'autre part, tout lien de client/fournisseur entre des composants modulaires peut être généralisé en un *lien générique* : si `M2` est client de `M1`, alors `M2` peut être paramétré de manière à abstraire ce lien par la définition d'un foncteur `M2'(X:S1)` où `S1` est une signature de `M1`. Cette technique permet ainsi par ex. de généraliser un lien d'héritage de modules en un *héritage générique* :

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

(3) La notion de *délégation* dans le cadre de la programmation orientée objet (cf. par ex. [Cra02, GHJV95]) se définit ainsi : on dit que l'élément E d'un objet `obj1` est *délégué* à un objet `obj2`, si `obj2` apparaît comme attribut d'`obj1` et que E dépend d'`obj2`. Le comportement de `obj1` est donc dépendant (en général dynamiquement) du comportement d'`obj2`. Par exemple, en C++ :

```
class Window {
       Window(WindowSys* win);
       WindowSys* _imp;    // délégué
         ...
    };

void Window::DrawRect (const Point& p1, const Point& p2) {
        _imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
    }
```

Cette notion possède un équivalent dans le cadre modulaire : on peut en effet admettre qu'un module `M2` « délègue » certaines de ses définitions à un autre module `M1`, si `M1` est contenu dans `M2` en tant que sous-module et que certains éléments de `M2` sont le fait de `M1`. Par exemple :

```
module type WINDOW = sig module Imp : WINDOW_SYS ... end

module Win : WINDOW = struct module Imp = ...  let draw_rect = Imp.device_rect ... end
```

Évidemment, l'affectation d'un module à `Imp` ne peut s'établir ici que statiquement au cas par cas, contrairement à la délégation d'objet. Dans le cadre de la programmation modulaire générique, on peut néanmoins faire mieux en utilisant un foncteur. Par exemple :

```
module F (X : WINDOW_SYS) : WINDOW = struct module Imp = X ... end
```

L'affectation du composant modulaire de « délégation » est réalisée par une application de foncteur. On peut d'ailleurs considérer qu'un foncteur est par lui-même une mise en forme statique de délégation puisque ses paramètres expriment explicitement une dépendance affectable de modules :

```
module F (X : WINDOW_SYS) : WINDOW = struct ... end
```

Ces relations entre délégation orientée objet et foncteurs sont importantes pour comprendre pourquoi certains modèles de conception objet peuvent se traduire en programmation modulaire générique. C'est particulièrement le cas lorsque le rôle d'une délégation d'objet dans un modèle consiste avant tout à simplifier une architecture (par ex. pour limiter le nombre de sous-classes).

2.2. Modèles génériques architecturaux

Les relations générales que nous venons d'évoquer s'appliquent tout particulièrement aux modèles de conception de [GHJV95] qui résolvent des problèmes généraux d'architecture logicielle, et qui ne sont donc pas le propre des constructions fondées sur des classes.

2.2.1. Le Pont modulaire générique

Le **Pont** (*Bridge*) est l'un des modèles les plus généraux pour améliorer les qualités d'une architecture logicielle : *il permet de découpler une abstraction et son implémentation de manière à pouvoir faire évoluer indépendamment l'un et l'autre.* En d'autres termes, un **Pont** applique une abstraction pour affranchir un composant de certains choix définitifs d'implémentation. Dans le cadre de la programmation orientée objet, ce modèle de conception est important pour résoudre les difficultés liées à une utilisation trop intensive de l'héritage (par ex. la multiplication de classes qui peut résulter d'une opération d'extension). Un **Pont** est réalisé dans la pratique par une réorganisation d'un ensemble de classes en plusieurs graphes d'héritage, reconnectés ensuite par le biais de *délégations*.

Dans le cadre modulaire, la programmation générique permet d'appliquer une solution très similaire à celle préconisée par un **Pont** objet. Par exemple, voici un avatar modulaire de l'exemple proposé dans [GHJV95]. Il s'agit d'un type de *fenêtres* pour lequel il existe une dépendance entre les possibilités de ces fenêtres et le système d'affichage sous-jacent. Tout d'abord, considérons la situation initiale :

```
module type WINDOW = sig type t val start : t -> unit end;;
```

```
module Win_Graphics : WINDOW = struct
  type t = ...(* dépend de Graphics *)
  let start w = ...(* dépend de Graphics *)
end;;
```

```
module Win_LablTk : WINDOW = struct
  type t = ... (* dépend de LablTk *)
  let start w = ... (* dépend de LablTk *)
end;;
```

L'extension des possibilités de la signature `WINDOW` – son *extension « verticale »* par héritage – implique alors une extension distincte pour chacun des modules qui la satisfait :

```
module type WINDOW_EXT = sig include WINDOW  val draw_text : t -> string -> unit end;;
```

```
module Win_Graphics_Ext : WINDOW_EXT = struct
  include Win_Graphics
  let draw_text w s = (* dépend de Graphics *)
end;;
```

```
module Win_LablTk_Ext : WINDOW_EXT = struct
  include Win_LablTk
  let draw_text w s = (* dépend de LablTk *)
end;;
```

Toute nouvelle extension de `WINDOW` ou de `WINDOW_EXT` impliquera une telle implémentation multiple. Ainsi, afin de limiter cette augmentation du nombre des composants, on peut appliquer un **Pont** : on extrait ce qui est spécifique au système d'affichage, on le place dans d'autres composants, et on les relie ensuite aux *fenêtres* par foncteur interposé :

```
module type SYS_WINDOW = sig
  type t
  val device_start : t -> unit
  val device_draw : t -> 'a -> unit
end;;
```

```
module type WINDOW = sig
  module W : SYS_WINDOW
  type t = W.t
  val start : t -> unit
end;;
```

```
module Win (X : SYS_WINDOW) : WINDOW = struct
  module W = X
  type t = W.t
  let start w = ... (* dépend de X *)
end;;

module Sys_Graphics = ... (* dépend de Graphics *)
module Sys_LablTk = ... (* dépend de LablTk *)
module Win_Graphics = Win (Sys_Graphics);;
module Win_LablTk = Win (Sys_LablTk);;
```

Il y a ici une séparation entre l'ensemble des systèmes d'affichage et l'ensemble des implémentations du type de *fenêtres*. Une extension « verticale » de cet ensemble de modules est alors réalisée par un unique héritage générique :

```
module Win_Ext (X : WINDOW)  : WINDOW_EXT = struct
  include X
  let draw_text w s = ... (* dépend de X *)
end;;

module Win_Graphics_Ext = Win_Ext (Win_Graphics);;
module Win_LablTk_Ext = Win_Ext (Win_LablTk);;
```

Le **Pont** étant essentiellement une abstraction concrétisée par *délégation*, il est naturel que ce modèle se traduise dans le cadre de l'utilisation de foncteurs. L'exemple ci-dessus montre que cette version induit non seulement une évolutivité simplifiée, mais aussi une exploitation possible des compatibilités sous-jacentes.

2.2.2. L'Adaptateur modulaire générique

L'**Adaptateur** (*Adapter*) est également un modèle de conception qui agit au niveau général d'une architecture logicielle en facilitant la réutilisation entre ses composants : *il convertit l'interface d'un composant en une autre afin qu'elle se conforme à l'attente d'un composant client.* La version simple de ce modèle – originellement, l'**Adaptateur de classe** – consiste à obtenir le composant adapté par le biais d'un héritage simple du composant à adapter. Lorsque cette adaptation doit s'appliquer sur un ensemble de composants qui satisfont la même signature, il en existe une version plus technique – originellement, l'**Adaptateur d'objet** – qui consiste à obtenir le composant adapté par le biais d'une *délégation* au composant à adapter. Cette délégation peut se traduire en foncteur. Par exemple, Paulson dans [Pau97] propose une signature pour les structures arithmétiques. Cette signature diffère de celle des modules arithmétiques `Int32`, `Int64` et `Complex` de la bibliothèque standard d'OCaml. Par un **Adaptateur générique**, il est possible de remédier à cette situation :

```
module type ARITH1 = sig
  type t
  val sum : t * t -> t
  val prod : t * t -> t
  ...
end;;
```

```
module type ARITH2 = sig
  type t
  val add : t -> t -> t
  val mul : t -> t -> t
  ...
end;;
```

```
module Arith_Adapter (X : ARITH2) : ARITH1 =
struct
  type t = X.t
  let sum = uncurry X.add
  let prod = uncurry X.mul
  ...
end;;

module Arith_Int64 = Arith_Adapter (Int64);;
module Arith_Int32 = Arith_Adapter (Int32);;
module Arith_Complex = Arith_Adapter (Complex);;
```

Notons que l'adaptation peut également exploiter les possibilités de « redéfinition modulaire » (cf. p. 47) :

```
module Adapter (X : S1) : S2 = struct
  include X
  ...  (* <-- redéfinitions de certains éléments de X *)
end;;
```

2.2.3. La Façade modulaire générique

La **Façade** (*Façade*) est également un modèle de conception susceptible de s'appliquer à n'importe quelle type d'architecture logicielle : *il consiste à définir un nouveau composant qui rend un ensemble de composants existants plus facile à exploiter*. En d'autres termes, une **Façade** permet de faciliter le travail des clients d'une architecture en intégrant directement un couplage récurrent de certains de ses composants. Dans le cadre orienté objet, ce modèle est d'ailleurs souvent réalisé par une classe qui ne contient que des méthodes de classe (*utility class* [BRJ99]), c'est-à-dire une classe en forme de module. Par exemple, nous adaptons ici l'exemple de [GHJV95] à propos d'un compilateur dont les phases de calcul sont représentées par des composants distincts. Voici leurs principales signatures :

```
module type SCANNER = sig
  type in_stream
  type token_stream
  val scan : in_stream -> token_stream
end;;

module type PARSER = sig
  module S : SCANNER
  module Parse_Tree : TREE
  val parse : S.token_stream -> Parse_Tree.t
end;;
```

```
module type TREE = sig type t  ... end;;


module type CODE_GENERATOR = sig
  module Parse_Tree : TREE
  type code
  val code_gen : Parse_Tree.t -> code
end;;
```

Une **Façade** consiste donc en un module qui inclut des implémentations de ces signatures, tout en gérant et en organisant leurs liens réciproques :

```
module Compiler = struct
  module S = struct type in_stream = ... end
  module P = struct module S = S module Parse_Tree = ... end
  module C = struct type code = ... end
  let compile x = C.code_gen (P.parse (S.scan x))
end;;
```

Soulignons que cette version n'illustre pas toutes les simplifications dont pourraient bénéficier les clients d'une telle **Façade**, comme par exemple la gestion explicite des *partages de types* (*sharing types*) entre les composants utilisés.

Une **Façade** peut évidemment devenir générique si certains de ses éléments sont mis en paramètre. Par exemple (ici, les partages de types restent encore à expliciter dans l'entête du foncteur) :

```
module type COMPILER = sig
  module S : SCANNER
  module C : CODE_GENERATOR
  val compile : S.in_stream -> C.code
end;;
```

```
module Compiler (X1 : SCANNER) (X2 : CODE_GENERATOR)
   : COMPILER = struct
  module S = X1
  module P = struct
               module S = X1 module Parse_Tree = ...
             end
  module C = X2
  let compile x = C.code_gen (P.parse (S.scan x))
end;;
```

2.2.4. Le Visiteur modulaire générique

Le **Visiteur** (*Visitor*) permet de rendre un ensemble de classe plus facilement extensible en *définissant une nouvelle opération sans avoir à modifier les classes des éléments sur lesquels elle opère*. Rappelons une des formes du problème sous-jacent : dans une hiérarchie de composants, les « composants-feuilles » sont souvent ceux à partir desquels une extension est effectuée. Ces composants doivent souvent être tous étendus séparément, engendrant alors une augmentation importante de leur nombre. L'idée du **Visiteur** est de construire des objets qui contiennent toutes les fonctions d'extension des classes à étendre. Des **Visiteurs** peuvent alors être transmis aux instances de ces classes, qui appelleront leur fonction spécifique associée. Soulignons que cette technique n'est pas complètement satisfaisante dans ses formes basiques car elle demande d'expliciter le mécanisme de sélection. Elle est d'ailleurs beaucoup discutée dans la littérature (cf. par ex. [PJ98]).

Plus généralement, le problème auquel s'attaque le **Visiteur** consiste à pouvoir mêler *extensions horizontales* (nombre des sous-classes) et *extensions verticales* (héritage) (cf. par ex. [FF99]). En ce sens, ce problème n'est nullement exclusif à la programmation orientée objet. Des situations comparables peuvent en effet se produire en programmation modulaire. La plus simple est la suivante : supposons que `M1, M2, ..., Mn` satisfassent `S` en usant de types internes distincts. Pour chaque extension de `S`, les modules `M1, M2, ..., Mn` devront être étendus séparemment. Comme dans le cadre orienté objet, il y a donc multiplication de modules. L'idée du **Visiteur** peut alors être adaptée à la programmation modulaire afin de remédier à cela. Tout d'abord, considérons une situation initiale simple composée de modules qui implémentent la même signature :

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

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

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

On peut alors faire en sorte d'étendre ces modules au moyen d'un **Visiteur générique** qui met en œuvre la connexion effective entre les modules étendus et le code de leur extension :

```
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;;
```

Dès lors, les extensions spécifiques peuvent prendre la forme suivante :

```
module Visitor_Mul = struct
  type t_int = int
  type t_float = float
  let visit_int = ( * )
  let visit_float = ( *. )
end;;
```

```
module Visitor_Div = struct
  type t_int = int
  type t_float = float
  let visit_int = ( / )
  let visit_float = ( /. )
end;;
```

Soulignons que cette technique est ici strictement statique contrairement au **Visiteur** original. Toute extension demande une application particulière des foncteurs d'extension. Afin d'obtenir un comportement plus flexible et plus dynamique, le programmeur ML pourrait être tenté d'utiliser la programmation fonctionnelle. On notera également que l'exemple ci-dessus se situe plutôt dans le cadre des *types de données modulaires*[1] (ce n'est pas inhérent à cette technique).

[1] Ce terme n'est pas classique, mais il est rendu nécessaire au sein de la programmation ML. En effet, rappelons qu'un

2.3. Modèles génériques de modification de types de données

Parmi les modèles de conception de [GHJV95], il en existe qui agissent sur les classes de manière à induire des transformations sur l'ensemble des objets qui en sont issus. Ces modèles sont donc susceptibles de s'appliquer à d'autres constructions de types, comme les *types de données modulaires*.

2.3.1. La Procuration modulaire générique

La **Procuration** (*Proxy* ou *Surrogate*) est un modèle de conception qui *permet de contrôler l'accès aux instances d'un type (les objets) par d'autres instances qui servent alors d'intermédiaires ou de remplaçants*. Il s'agit donc de modifier le comportement général d'un type, et de faire en sorte d'assurer une maîtrise de l'accès à ses instances. Par exemple, une **Procuration** peut permettre de déterminer des niveaux différents de protection des instances (« **Procuration de protection** »), ou de différer la création complète des instances jusqu'à leur utilisation effective (« **Procuration virtuelle** »), ou plus généralement d'ajouter des comportements particuliers aux liaisons/références sur les instances (cf. par ex. [Ale01]). D'autre part, si la **Procuration** est avant tout une manipulation qui se fait au cas par cas, composant par composant, sa forme est parfois répétée sur des types satisfaisant une interface. On peut donc la généraliser en une **Procuration générique**. L'exemple développé dans [GHJV95] est celui d'une **Procuration virtuelle** qui consiste à émuler un *mécanisme de paresse* au sein même de la programmation objet. La base de ce mécanisme existe dans la plupart des langages ML (cf. le module `Lazy` en OCaml), et une **Procuration virtuelle générique** consiste donc à l'utiliser afin d'ajouter génériquement un comportement paresseux aux instances d'un type :

```
module type IMAGE = sig
  type color
  type image
  val rgb : int -> int -> int -> color
  val make_image : color array array -> image
  val dump_image : image -> color array array
  val draw_image : image -> int -> int -> unit
end;;
```

```
module Image_Proxy (X : IMAGE) : IMAGE = struct
  module I = X
  type color = I.color
  type image = I.image lazy_t
  let rgb = I.rgb
  let make_image x = lazy (I.make_image x)
  let dump_image x = I.dump_image (Lazy.force x)
  let draw_image x = I.draw_image (Lazy.force x)
end;;
```

Dans cet exemple, tout type satisfaisant `IMAGE` est rendu paresseux par l'application d'`Image_Proxy` : les instances d'images ne seront construites que lors de leur première utilisation.

2.3.2. Le Singleton modulaire générique

Le **Singleton** (*Singleton*) est un modèle de conception qui *permet d'assurer qu'un type (une classe) ne produise au plus qu'une seule instance (un seul objet)*. Comme pour la **Procuration**, il s'agit donc également de modifier le comportement général des instances d'un type, en particulier lorsque celles-ci sont globales au programme : on court-circuite les constructeurs afin qu'ils ne produisent au plus qu'une instance. Il est donc également possible de généraliser ce modèle en un **Singleton générique**. On se doit cependant de considérer ici la différence d'organisation entre les types de données modulaires et les classes (cf. par ex. [Coo90, Nar03]). Dans le cadre modulaire, l'unique instance du type ne contient pas toute l'information, elle requiert encore l'utilisation de son module d'origine (une instance d'un type modulaire n'est pas autonome). Ainsi, une idée possible de réalisation du **Singleton** dans le

type de données est la mise en commun d'une *représentation des données* et de *fonctions dédiées*. Cette mise en commun peut être réalisée par encapsulation dans un module. Si l'on applique un masquage d'implémentation du type, doublée d'une description extérieure par le biais d'une interface, on obtient un *type de données abstrait* (cf. par ex. [Pie02]). Cependant, le masquage d'implémentation est contraignant, et dans les langages ML il n'est pas obligatoire. Les types de données n'y sont donc pas toujours abstraits. D'autre part, la programmation générique peut être entravée par le masquage, car les représentations des données se composent alors difficilement entre elles – en particulier, les *partages de types* (*sharing types*) deviennent délicats à exprimer. On peut donc parfois se contenter de *types de données modulaires* sans masquage, les manipuler et les composer, quitte à les masquer *a posteriori* (cf. [Nar05]).

cadre modulaire consiste plutôt à utiliser un héritage générique qui permet de redéfinir la méthode de création et assurer l'unicité de l'instanciation. Le **Singleton** ne contient alors qu'une information qui contrôle cette unicité, mais qui n'est pas l'instance elle-même. Par exemple :

```
module type COUNTER = sig
  type t
  val make : unit -> t
  val get_value : t -> int
  val incr : t -> unit
end
```

```
module Counter_Single (X : COUNTER) : COUNTER =
struct
  include X
  let first_call = ref true
  let make () =
    if !first_call then
      (first_call := false;
       make ())
    else failwith "No new value"
end;;
```

2.3.3. Le Décorateur modulaire générique

Un **Décorateur** (*Decorator*) *permet d'attribuer des comportements supplémentaires aux éléments d'un objet.* À la base du **Décorateur**, il y a une *délégation* d'objets, mais celle-ci est fondée sur un objet typé par une classe mère de l'objet déléguant. En d'autres termes, la délégation d'un **Décorateur** définit une relation cyclique qui permet une composition d'actions provenant de délégations successives. Dès lors, si la possibilité d'agir dynamiquement est hors de portée de la programmation modulaire générique, il est possible de retenir un point important de ce modèle de conception : basé sur la délégation, un **Décorateur** permet d'enrichir le comportement et les possibilités d'une instance sans en changer l'interface. Cette situation généralise celle de la **Procuration** et du **Singleton** : un **Décorateur générique** modulaire est capable de modifier uniformément le comportement général d'un type existant tout en n'en changeant pas la signature. Cette propriété permet ainsi des « compositions fonctorielles » de **Décorateurs** aussi longues que l'on veut. Par exemple, traduisons l'un des exemples proposés dans [GHJV95] à propos d'un type d'éléments graphiques qui s'affichent dans des fenêtres, et dont on veut pouvoir augmenter les comportements :

```
module type VISUAL = sig
  type t
  val draw : t -> unit
  val bounding_box : t -> (int*int) list
  ...
end;;
```

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

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

Puisque les modules résultats satisfont également `VISUAL`, l'opération de décoration est transparente à ses clients. On peut donc composer ces décorateurs :

```
module Text : VISUAL = struct ... end;
module G = DropShadow (Border (Text));;
```

Soulignons que les **Décorateurs** peuvent également servir à préciser une spécification : ils permettent par exemple d'associer des assertions, des contrôles, des affichages d'information associée à chaque application de fonction [Nar05].

2.3.4. Le Composite modulaire générique

L'intention originale du **Composite** (*Composite*) est *de pouvoir considérer uniformément des objets individuels et leurs combinaisons.* Concrètement, l'idée est de représenter ces combinaisons par des

références stockées dans des conteneurs, traités alors comme de simples attributs. Cependant, dans un monde strictement typé, le mélange de simples valeurs et de leurs compositions ne peut pas avoir directement lieu. On peut en revanche retenir l'idée de construire un type de données T à partir de combinaisons d'instances de type T, et ce de manière générique pour toute implémentation de T. Par exemple, reconsidérons les éléments graphiques qui ont illustré le **Décorateur** ci-dessus ; un **Composite générique** possible revient à définir des types de données constitués de listes de ces éléments graphiques :

```
module type VISUAL = sig
  type t
  val draw : t -> unit
  val bounding_box : t -> (int*int) list
  ...
end;;
```

```
module Visual_Set (G : VISUAL) : VISUAL =
struct
  type t = G.t list
  let draw l = List.iter G.draw l
  let bounding_box l = List.fold_left ... l
  ...
end;;
```

Remarquons qu'une telle construction peut sauvegarder une autre propriété du **Composite** original : `Visual_Set` est de signature `VISUAL -> VISUAL`. En d'autres termes ce foncteur est itérable :

```
module Line : VISUAL = struct ... end;;
module Line_Set = Visual_Set (Line);;
module Line_Set_of_Set = Visual_Set (Line_Set);;
module Line_Set_of_Set_of_Set = Visual_Set (Line_Set_of_Set);;
```

Toutefois, même si les types de données `Line`, `Line_Set` etc., satisfont tous `VISUAL`, leurs instances ne pourront pas être directement combinées. Ce manque de flexibilité est le prix d'une meilleure sûreté de typage. En effet, si par exemple la classe mère du modèle **Composite** original inclut des méthodes n-aires pour lesquelles les paramètres doivent être du même type (par ex. une fonction de comparaison), il sera difficile d'en assurer statiquement le bon typage (cf. p. 47).

2.3.5. L'Itérateur modulaire générique

L'**Itérateur** (*Iterator*) *fournit un moyen d'accès séquentiel aux éléments d'un agrégat d'objets sans en exposer la représentation interne.* Plus précisément, l'idée de ce modèle est de résoudre le problème de la multiplicité possible des formes d'accès à des types différents de structures de données (par ex. représentation interne, algorithmique, opérations disponibles). L'idée est donc de séparer les moyens d'itération et les définitions des structures de données, et d'organiser leurs liens réciproques. Il est évident que l'on peut implémenter des types de données modulaires avec ce but à l'esprit. Voici une implémentation possible dans le cadre des types de listes :

```
module type LIST = sig
  type 'a lst
  val empty : 'a lst
  val cons : 'a -> 'a lst -> 'a lst
  val hd : 'a lst  -> 'a
  val tl : 'a lst -> 'a lst
end;;

module L : LIST = struct ... end
```

```
module type ITERATOR = sig
  type 'a iterator
  type 'a iterated
  val make : 'a iterated -> 'a iterator
  val next : 'a iterator -> ('a iterator * 'a)
  val empty : 'a iterator -> bool
end;;

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

La connexion entre les itérateurs et les listes peut alors être établie par foncteur interposé :

```
module It1 (X : LIST) : ITERATOR_LIST = struct
   : module L = X
  type 'a iterated = 'a L.lst
  type 'a iterator = 'a L.lst
  let make l = l
  let next iter = (L.tl iter, L.hd iter)
  let empty iter = (iter = L.empty)
end;;
```

L'indépendance entre l'ensemble des implémentations qui satisfont `LIST` et celles qui satisfont `ITERATOR` est réalisée. Il sera maintenant possible de se donner de nouvelles implémentations des itérateurs et de les connecter avec celles des listes. De même, des extensions génériques des itérateurs sont envisageables sans avoir à modifier les types associés aux listes :

```
module type ITERATOR_REVERSE = sig
  include ITERATOR
  val preced : 'a iterator -> ('a iterator * 'a)
end;;

module type ITERATOR_REVERSE_LIST = sig
  module L : LIST
  include (ITERATOR_REVERSE with type 'a iterated = 'a L.lst)
end;;

module It_Reverse (X : LIST) : ITERATOR_REVERSE_LIST = struct ... end;;
```

Afin de rendre cette traduction encore plus fidèle, on pourrait intégrer la relation mutuelle préconisée dans le modèle original entre les structures de données et les types d'itérateurs. Une de ses conséquences consiste à permettre qu'un itérateur différent puisse être affecté à chaque instance de structure de données. Toutefois, lorsque le typage est strict, le mélange d'itérateurs provenant de modules distincts se voit proscrit (cf. aussi le cas du **Composite**). Un programmeur ML pourrait donc être tenté ici d'utiliser plutôt la programmation fonctionnelle pour exprimer une telle flexibilité. Grâce aux fonctions de première classe, il lui sera par exemple aisé de changer la stratégie de parcours d'un itérateur sans en changer le type. Toutefois, la cohérence entre plusieurs appels sera alors plus difficile à assurer.

3. Modèles et programmation fonctionnelle

3.1. Modèles purement fonctionnels

Les modèles de conception analysés jusqu'ici ont permis d'obtenir des traductions en programmation modulaire générique (même si la programmation fonctionnelle a pu également être parfois invoquée). Il existe cependant d'autres modèles du catalogue de [GHJV95] qui se trouvent plus directement liés à la programmation fonctionnelle. Trois d'entre eux concernent en effet implicitement la notion de fonction de première classe, en se basant sur la propriété suivante : les objets d'un langage objet sont des valeurs de première classe qui peuvent contenir (ou faire référence à) des fonctions. Les objets permettent donc de remédier au manque de mécanismes spécifiques de programmation fonctionnelle (comme c'est le cas en Java et en C++).

3.1.1. La Commande

Une **Commande** (*Command*) permet *d'encapsuler une « requête » – l'application d'une fonction – en un objet, autorisant ainsi le paramétrage des clients par différentes requêtes.* Une requête devient alors une valeur de première classe, facilement transmissible et stockable. La technique qui réalise cela en programmation orientée objet consiste à définir une classe abstraite contenant une méthode virtuelle représentant l'application d'une requête (ou d'un ensemble de requêtes – la **Macro-Commande**), dérivée ensuite en requêtes concrètes. Par exemple (cf. [GHJV95]) :

```
class Command {
public :
    vitural ~Command();
    virtual void Execute() = 0;
}

class OpenCommmand : public Command { ... }
class PasteCommand : public Command { ... }
```

Le bon fonctionnement de chacune de ces requêtes nécessite l'encapsulation de leur environnement (c'est-à-dire l'idée même de la notion de *fermeture*). Évidemment, nulle difficulté à traduire cette idée dans un langage basé sur le modèle fonctionnel. L'exemple d'un menu utilisant des commandes comme ci-dessus peut s'obtenir en quelques lignes de ML :

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

On notera cependant qu'une **Commande** objet peut par exemple aussi inclure l'objet capable d'invoquer la méthode qui constitue la requête (le « receveur »). La programmation orientée objet rend cela aisé, et l'autonomie de la requête peut en être augmentée d'autant. Il n'en demeure pas moins que les versions simples de ce modèle de conception sont rendues immédiates par la programmation fonctionnelle (cf. par ex. [Küh99, SM00]). Ce modèle trouve d'ailleurs des variations particulières en C++ [Ale01], langage qui propose une construction proche des fermetures : la notion d'« objets-fonction » (objets surchargeant l'opérateur `()` – eux aussi appelés *foncteurs*).

3.1.2. La Stratégie

Une **Stratégie** (*Strategy*) permet *aux algorithmes d'évoluer indépendamment des clients qui les utilisent.* Il s'agit donc ici d'abstraction fonctionnelle spécialisée dans la mise en œuvre d'algorithmes : tout comportement spécifique d'un algorithme peut être remplacé par un paramètre fonctionnel. En programmation orientée objet, cela est réalisé comme pour la **Commande**, c'est-à-dire au travers d'une classe abstraite contenant une méthode virtuelle représentant l'algorithme ou la partie d'algorithme. La différence avec la **Commande** réside dans la nécessité moindre de stocker le « receveur » et/ou l'environnement de la requête. Par exemple, considérons le cas où un système de gestion de processus a besoin d'un algorithme de tri sans que celui-ci soit fixé d'avance. La programmation fonctionnelle permettra facilement de concrétiser cette idée sans passer par une encapsulation dans un objet :

```
let bin_sort arr cmp = ...;;
let quick_sort arr cmp = ...;;    (* algorithmes de tri*)
let merge_sort arr cmp = ...;;
let process_manager algo_sort = ...algo_sort...;;
```

3.1.3. Le Patron de méthode

Un **Patron de méthode** (*Template method*) définit *un squelette d'un processus ou d'algorithme en déléguant certaines étapes à ses sous-classes.* Il s'agit donc également ici d'abstraction fonctionnelle, mais dans un cadre plus statique : celle-ci passe par le biais de méthodes abstraites dont l'implémentation est le fait des classes dérivées (contrairement à la **Commande** et à la **Stratégie** qui représentent en elles-mêmes les fonctions à encapsuler, et qui nécessitent des clients extérieurs pour être utilisées). Dans le cadre simplifié de la programmation fonctionnelle, une telle distinction est peu opportune. Si l'on considère l'exemple donné dans [GHJV95] à propos d'un processus de visualisation qui dépend d'une séquence d'opérations dont l'une d'entre elles est sujette à variation, la programmation fonctionnelle OCaml offre ici aussi une solution directe :

```
let view_display ?(do_display = fun () -> ()) =
  set_focus ();
  do_display ();
  reset_focus ();;
```

Évidemment, si l'on voulait ici être plus fidèle au caractère statique original de ce modèle, il serait facile d'en donner une version fonctorisée.

3.2. Un modèle lié à la notion de types inductifs : l'Interpréteur

L'**Interpréteur** (*Interpreter*) est un modèle de conception qui s'attache à un problème particulier de construction de programme : *il décrit une représentation d'une grammaire G_L d'un langage L utilisable directement par un mécanisme d'interprétation de L.* La version orientée objet de ce modèle se fonde sur la construction d'une hiérarchie de classes qui reflète la structure de G_L. Une de ses formes les plus naturelles consiste principalement à utiliser des classes abstraites pour les *non-terminaux* de G_L, et des sous-classes pour définir les membres droit des règles de G_L. Dans le cadre de la programmation fonctionnelle typée ML, une telle grammaire est plutôt représentée par un ensemble de *types inductifs* et de filtrages associés. On observera par exemple cette différence de style dans les versions ML et Java de l'ouvrage d'Appel sur la compilation [App97]. Évidemment, la solution utilisant les types inductifs possède des inconvénients – en particulier pour ce qui concerne son extensibilité – mais elle est en revanche sûre du point de vue du typage.

Notons que cette dualité de représentation est un lié au dilemme appelé l'« *Expression Problem* » [Wad98] (lié également au problème traité par le **Visiteur**), c'est-à-dire celui qui consiste à pouvoir concilier les extensions horizontales et verticales. Ce problème a été beaucoup étudié dans le cadre de l'**Interpréteur** (cf. par ex. [Bru03, ZM05]), et les solutions récentes ont tendance à exploiter des constructions hybrides et multi-paradigmes.

4. Modèles et programmation objet

Il existe également des modèles de conception présentés dans [GHJV95] qui sont basés pour beaucoup sur des caractéristiques intrinsèques aux objets dont nous rappelons ici les principales : (1) les objets encapsulent un *état*, susceptible de changer dynamiquement ; (2) les objets entretiennent des références à cet état et à leurs opérations associées ; (3) les objets d'un même type peuvent posséder des implémentations distinctes (ce sont des valeurs autonomes du point de vue de leur comportement). Si la seconde propriété rapproche la programmation objet de la programmation modulaire (en particulier pour ce qui concerne les types de données), la première et la troisième propriétés en sont plus spécifiques. Lorsque ces dernières fondent un modèle de conception, la solution qu'il concrétise se trouvera donc plutôt inhérente à la programmation objet.

4.1. Modèles basés sur la notion d'état

Tout d'abord, certains modèles de conception exploitent et mettent en exergue la notion d'état et d'automie des objets. En voici la description :

Un **État** (*State*) *permet à un objet de modifier dynamiquement son comportement lorsque son état interne l'est aussi.* Il s'agit simplement ici de *délégation*, mais dans un cas où les possibilités dynamiques de cette technique sont mises en exergue. En effet, contrairement à d'autres modèles comme le **Pont** ou l'**Adaptateur** qui utilisent la délégation avant tout comme technique de simplification d'une architecture, l'**État** préconise que l'objet délégué soit réaffectable pendant l'exécution. La construction la plus ressemblante dans le cadre du ML de base consiste probablement à faire appel à un module avec état interne tel que :

```
module M = struct
  let state = ref ...
  let set_state st = (state := st)
  let f1 () = ... !state ...
  let f2 () = ... !state ...
end;;
```

Néanmoins, les possibilités de compatibilité et d'extensibilité sont ici bien moindres que pour un **État** réalisé dans le cadre d'un mécanisme adapté aux objets.

Un **Observateur** (*Observer*) permet de *définir une dépendance entre objets de telle manière que lorsque l'un d'eux change d'état, les autres en sont informés et sont également en mesure de changer d'état.* La forme générale de ce modèle consiste donc à gérer un ensemble d'« observateurs » à partir d'un « sujet ». Lorsque ce sujet change d'état, il fait appel aux fonctions de mise à jour de tous ses observateurs, fonctions qui elles-même utilisent les informations liées au sujet. L'exploitation centrale de la notion d'état implique ici un modèle de conception purement objet. Notons cependant que la programmation fonctionnelle permet parfois de simplifier ce modèle. En effet, le sujet peut non pas stocker l'ensemble de ses objets observateurs (afin de faire appel à leur méthode de mise à jour), mais directement l'ensemble des fonctions de mise à jour [SM00]. L'**Observateur** devient alors une spécialisation du modèle **Commande**.

Un **Mémento** (*Memento*) permet de *capturer l'état interne d'un objet de manière à pouvoir le restaurer ultérieurement*, par exemple pour effectuer des action inverses (des « *undos* ») dans de bonnes conditions. À nouveau, l'utilisation importante de la notion d'état implique ici un modèle de conception plutôt objet.

Un **Médiateur** (*Mediator*) *définit un objet qui encapsule la manière dont interagissent un ensemble d'objets, tout en permettant à ces interactions d'être modifiées.* Ce modèle de conception est à rapprocher du modèle **Façade** dans un contexte purement orienté objet. Il encourage le comportement coopératif entre l'objet médiateur et les objets qu'il rassemble. L'idée est ici de simplifier les interactions dynamiques entre les objets en tant qu'entités autonomes.

Une **Chaîne de responsabilités** (*Chain of Responsibility*) permet de *relier des objets entre eux de manière à ce qu'une requête puisse être transmise au travers d'une chaîne de relations vers l'objet le plus à même d'y répondre.* Ces chaînes sont mises en place dynamiquement au travers d'informations qui font partie des états des objets.

Un **Poids Mouche** (*Flyweight*) consiste *à organiser un partage de données afin de rendre efficace la gestion d'un grand nombre d'objets.* Ce modèle résout ainsi un problème général de la programmation objet qui a tendance à multiplier facilement le nombre d'objets mis à contribution. Pourtant, les informations qu'ils contiennent ne leur sont pas toujours spécifiques. Le **Poids Mouche** propose donc une technique de mise en commun des données contenues dans les états de toute une famille d'objets.

Pour tous ces modèles, la gestion des états est un point essentiel. Leur mise en œuvre dans de bonnes conditions requiert donc une notion d'objet prête à l'emploi.

4.2. Modèles de construction d'instances

La plupart des modèles dits « créationnels » présentés dans [GHJV95] indiquent comment construire des familles d'objets aux types compatibles mais aux implémentations variées. Cette propriété échappe quelque peu aux autres paradigmes :

Une **Méthode de fabrication** (*Factory Method*) consiste à *définir un composant capable de créer des objets différents, tout en laissant le détail de cette construction à ses dérivés.* Ce modèle permet ainsi la création de valeurs aux comportements distincts sous la même égide. Dans le cadre de la programmation orientée objet, ce modèle n'est rien d'autre qu'une utilisation de méthodes abstraites associées à la construction d'objets.

Une **Fabrique abstraite** (*Abstract Factory*) donne les moyens de *définir de manière cohérente des familles d'objets aux propriétés similaires.* La concrétisation de ce modèle est souvent constituée d'un ensemble de **Méthodes de fabrication**.

Un **Monteur** (*Builder*) permet de *séparer la construction d'un objet complexe de sa représentation.* En d'autre termes, ce modèle est capable d'aider à la production d'objets complexes et composés de parties multiples, et ce en faisant abstraction de leur implémentation. Par comparaison, les **Fabriques abstraites** n'offrent que les moyens de produire des parties cohérentes d'objets complexes (et non pas ces objets eux-mêmes).

Un **Prototype** (*Prototype*) permet de *spécifier des familles d'objets à partir d'instances prototypes.* Ce modèle constitue une émulation des « langages objet à prototypes », c'est-à-dire des langages orienté objet sans classes (comme par ex. Self). Les prototypes sont des objets qui contiennent des attributs qui en décrivent la construction. Ces attributs sont aptes à être transmis à d'autres objets ou plus simplement d'établir un « clonage » dans de bonnes conditions. À nouveau, il s'agit ici d'exploiter la latitude qui existe dans l'implémentation d'un objet sous l'égide d'un seul type.

Le besoin de faire varier les représentations des valeurs d'un type n'est évidemment pas le propre de la programmation orientée objet. On pourrait effectivement obtenir des versions fonctionnelles ou fonctorielles des modèles ci-dessus (en particulier pour les deux premiers), mais cela en faisant l'impasse sur la variabilité de l'implémentation des instances construites, ce qui en affaiblirait le bien-fondé.

5. Conclusion

Une expérience de programmation est toujours sujette à caution puisque dépendante de la qualité des programmes présentés et de l'habileté des programmeurs impliqués. À ce titre, certains exemples des pages précédentes pourraient peut-être trouver encore de meilleures formes. Toutefois, il nous semble déjà possible d'en tirer quelques conclusions concrètes :

1. Certains modèles de conception de [GHJV95] sont suffisamment proches de problèmes généraux liés à la construction d'une architecture logicielle ou de types de données pour posséder une forme en programmation modulaire générique. En ML, nous avons pu ainsi obtenir des modèles de conception exprimés en termes de foncteurs. Ces modèles se caractérisent par un typage plus précis et plus sûr qu'en programmation orientée objet, mais cela généralement au détriment de la flexibilité (un effet habituel des comportements statiques).

   Évidemment, au sein de ces modèles, les techniques d'utilisation des foncteurs ne sont pas surprenantes, la programmation modulaire générique permettant moins de variété d'interactions que la programmation orientée objet. Ceci dit, les modèles de conception sont plus caractérisés par leur intention que par les techniques qu'ils utilisent (« *The key difference between these patterns lies in their intents* » [GHJV95]). D'autre part, leur description en programmation générique sous forme de modèles de conception dûment nommés pourrait promouvoir quelque peu l'usage des foncteurs (qui reste assez rare dans la littérature ML, exception faite des références classiques [Bia94, BHL01]). Dans le même ordre d'idée, les modèles de conception de la programmation objet correspondent d'ailleurs pour beaucoup à la promotion de la *délégation d'objet* au détriment des mécanismes d'héritage.
2. Certains modèles de conception de [GHJV95] sont fortement liés à la transmission statique ou dynamique de fonctions. Dans leurs formes objet simples, ces modèles constituent des émulations de techniques élémentaires de programmation fonctionnelle. Ainsi, nous avons pu observer que des capacités de programmation fonctionnelle permettent de réaliser plus directement ces modèles (cf. aussi par ex. les travaux sur *Functional C++* [SM00, MS04]).
3. Certains modèles de conception de [GHJV95] « résistent » à une traduction dans d'autres paradigmes. Ce sont ceux qui utilisent les traits intrinsèques de la programmation objet. Deux de ces traits se sont imposés ici : le fait que les objets encapsulent un *état* modifiable dynamiquement, et le fait que les objets d'un même type peuvent posséder des implémentations distinctes.

Ainsi, l'expérience de programmation présentée ici justifie l'existence de langages à paradigmes multiples : le premier et le second point ci-dessus montrent qu'il est possible de construire des solutions similaires dans des paradigmes distincts et avec des propriétés différentes (relativement à la précision de typage, la flexibilité, ou la simplicité). Le troisième point indique comment expliquer et caractériser un paradigme puisqu'il en souligne des traits irréductibles et des techniques qui lui sont inhérentes.

Références

[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.

[Bia94] E. Biagioni. A structured TCP in Standard ML. In *SIGCOMM '94 : Proceedings of the conference on Communications architectures, protocols and applications*, pages 36–45, 1994.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[Bru03] K. Bruce. Some challenging typing issues in object-oriented languages, 2003.

[Coo90] W. R. Cook. Object-oriented programming versus abstract data types. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundations of Object Oriented Languages : Proceedings of REX School/Workshop*, number 489 in Lecture Notes in Computer Science, pages 151–178. Springer, 1990.

[Cra02] I. Craig. *The Interpretation of Object-Oriented Programming Languages.* Springer-Verlag, 2002.

[CS95] J. 0. 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.

[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.

[Gam92] E. Gamma. *Objectorientierte Software-Entwicklung am Beispiel von ET++ : Design-Muster, Klassenbibliothek, Werkzeuge.* PhD thesis, University of Zürich, 1992. Springer-Verlag.

[GHJV95] E. Gamma, R. Helm, R. Jonhnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[Küh99] Th. Kühne. *A Functional Pattern System for Object-Oriented Design.* Verlag Kovac, 1999.

[MS04] B. McNamara and Y. Smaragdakis. Functional programming with the FC++ library. *J. Funct. Program.*, 14(4) :429–472, 2004.

[Nar03] Ph. Narbel. Abstract abstract data types in modular and object-oriented programming. In *FOOL 10, ACM SIGPLAN Intl. Workshop on Foundations of OO Languages*, 2003. New-Orleans, USA.

[Nar05] Ph. Narbel. *Programmation fonctionnelle, générique et objet. Une introduction avec le langage OCaml.* Vuibert, 2005.

[Pau97] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1997.

[Pie02] B. C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[PJ98] J. Palsberg and C. B. Jay. The essence of the visitor pattern. In *Computer Software and Applications Conference, 1998. COMPSAC '98*, pages 9–15, 1998.

[Pre95] W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.

[Rém05] D. Rémy. Advanced examples with classes and modules, 2005. in The Objective Caml System. Documentation and User's Manual.

[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.

[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 *Foundations of Object-Oriented Languages (FOOL 2005)*, 2005. Long Beach, California.