Une Approche Fonctionnelle du « Modèle-Vue-Contrôleur »

Julien Signoles[1]

1: Laboratoire de Recherche en Informatique CNRS UMR 8623,
Université Paris-Sud
Bât. 490, 91405 Orsay Cedex, France
`Julien.Signoles@lri.fr`

Résumé

Le « Modèle-Vue-Contrôleur » (MVC) est un patron de conception fréquemment utilisé pour aider au développement d'interfaces graphiques en séparant l'état d'une entité (son modèle) de ses interactions avec l'utilisateur, notamment ses diverses représentations graphiques (ses vues). À l'instar de tous les patrons de conception, le MVC est toujours lié au paradigme de la programmation objet alors que les seules notions nécessaires à sa mise en œuvre sont les types, le polymorphisme, les interfaces et les implantations. Dans cet article, nous étudions le MVC dans un autre cadre tout aussi propice à sa mise en œuvre : celui d'un système de modules à la `ML`. Cette étude nous conduit à implanter le MVC de manière générique ; ce qui limite le schéma de code à implanter à chacune de ses utilisations.

1. Introduction

Un *patron de conception*, plus connu sous l'appellation anglophone de *design pattern*, décrit un problème récurrent et un schéma de sa solution, de manière à pouvoir l'appliquer autant de fois que désiré. Alors que cette notion a été introduite par un architecte, Christopher Alexander [1], on constate que, dans le monde de l'informatique, elle est presque toujours liée au paradigme de la programmation objet, ce qui provient de trois facteurs. Le premier est historique : le premier patron de conception a été introduit dans `Smalltalk-80` [5], un langage objet. Il s'agit du *Modèle-Vue-Contrôleur* [6] que nous allons présenter en détail ultérieurement. Les deux derniers facteurs sont plus pragmatiques : d'une part, le paradigme objet permet la modularité (au sens génie logiciel du terme) qui est indispensable au développement d'une application de grande taille et, d'autre part, ce paradigme fournit les notions nécessaires à la mise en œuvre d'un patron de conception, à savoir — pour paraphraser [3] — les types, le polymorphisme, les interfaces et les implantations. Or, un langage de modules à la `ML` [9], qui est un petit langage fonctionnel typé, possède aussi toutes ces qualités. Il y a donc tout lieu d'étudier les patrons de conception dans un tel cadre. Dans cet article, nous nous concentrerons sur l'étude d'un d'entre eux qui, en plus d'être probablement le plus célèbre, est un des plus difficiles à mettre en œuvre : le *Modèle-Vue-Contrôleur* (ou MVC). Cette étude va nous conduire à implanter le MVC de manière générique, limitant de ce fait le schéma de code à implanter à chacune de ses utilisations.

Dans la section 2, nous expliquerons l'utilité et le fonctionnement du MVC puis nous présenterons un système de modules à la `ML`, celui d'`Objective Caml` [8], dans la section 3. La section 4 détaillera une première approche générique du MVC utilisant ce système de modules et la section 5 explicitera une seconde approche étendant la première pour offrir plus de flexibilité. Enfin, nous indiquerons dans la section 6 comment utiliser le MVC.

2. Le Modèle-Vue-Contrôleur

De manière générale, le MVC a pour but une meilleure structuration d'une application. Il peut s'appliquer à deux niveaux différents : soit au niveau de l'application toute entière (ou, du moins, d'un sous-ensemble important), soit au niveau d'une de ses entités (typiquement, une structure de données particulière). Dans le premier cas, le MVC influe sur l'architecture globale de l'application. Sa mise en œuvre est alors plus un problème de génie logiciel qu'un problème de programmation par patron de conception. Dans cet article, nous nous concentrerons sur le deuxième cas.

Le MVC atteint son but en séparant l'état interne et les fonctionnalités d'une entité de ses interactions avec l'utilisateur. Pour ce faire, le MVC décompose une entité en trois composants distincts :

Le modèle représente l'état de l'entité et fournit ses fonctionnalités.

La vue donne à l'utilisateur une représentation de cet état.

Le contrôleur définit la façon dont l'interface utilisateur doit réagir aux requêtes de l'utilisateur.

Par exemple, pour un agenda, le modèle serait son contenu et les opérations permettant d'ajouter, de supprimer ou de modifier une personne de l'agenda. Une première vue pourrait être un affichage sous forme d'un tableau présentant toutes les personnes de l'agenda, une deuxième vue consisterait en un affichage sous forme d'onglets, chacun d'eux affichant les personnes dont le nom commence par une certaine lettre et, enfin, une autre vue afficherait des diagrammes statistiques résumant le contenu de l'agenda. De son côté, le contrôleur définirait les actions à entreprendre comme, par exemple, la suppression d'une personne lors d'un clic de souris sur son nom.

Le MVC doit permettre, d'une part, d'attacher plusieurs vues et/ou contrôleurs différents au même modèle et, d'autre part, de pouvoir lui en ajouter de nouveaux sans avoir à le réécrire. En outre, une modification du modèle doit entraîner une actualisation des vues. À l'inverse, le modèle doit pouvoir être modifié lorsqu'une requête utilisateur le nécessite ; une telle modification entraînant bien entendu une actualisation des vues. Au vu du rôle de ces composants, il existe des interactions naturelles entre eux, présentées figure 1 : la vue a besoin de connaître le modèle pour pouvoir le représenter et de prévenir le contrôleur lorsque un évènement lui parvient tandis que le contrôleur a besoin d'accéder au modèle et à la vue (en les modifiant éventuellement) pour répondre aux requêtes qui lui parviennent. De son côté, le modèle a besoin d'avertir la vue lors d'une modification de son état.

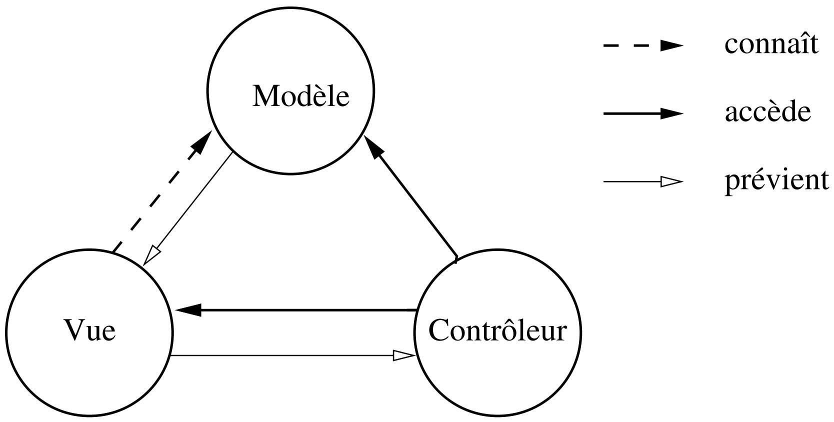


FIG. 1 – Interactions entre modèle, vue et contrôleur

Alors que le modèle repose sur l'état interne d'une entité, la vue et le contrôleur sont, quant à eux, intrinsèquement liés à l'interface graphique : de manière un peu schématique, la vue gère le rendu graphique alors que le contrôleur gère les évènements. Dans beaucoup de langages, ces évènements ne peuvent pas facilement être séparés du rendu graphique et il devient alors difficile d'effectuer une distinction claire et précise entre vue et contrôleur. C'est la raison pour laquelle ces deux composants sont, en pratique, souvent combinés en un seul ; on se contente alors de parler du modèle et de la vue, le contrôleur étant inclus dans la vue. C'est l'option choisie dans cet article.

Le MVC est largement répandu dans les applications orientées objets. On citera ici trois exemples parmi d'autres :

- `Smalltalk-80` a été, historiquement, le premier langage à l'utiliser intensivement [6].
- Plus récemment, `Java` base également nombre de composants graphiques sur ce concept, mais ne distingue pas vue et contrôleur (voir, par exemple, [10]).
- En `OCaml`, il en est de même pour la bibliothèque graphique `LablGTK2` [4, 12].

3. Le système de modules d'OCaml

Cette section présente un système de modules à la `ML` [9] : celui d'`Objective Caml` (ou `OCaml`) [8]. Ce sera celui que nous utiliserons dans la suite de cet article. Les différences avec les autres systèmes de modules à la `ML` — en particulier celui de `SML` [11] — sont principalement syntaxiques ; on peut consulter [13] pour les obtenir. Seules les constructions importantes pour notre étude sont présentées ici. Le lecteur intéressé peut se référer au manuel d'utilisation d'`OCaml` pour plus de détails sur les autres constructions. De plus, comparer un tel langage de modules avec un langage objets peut être intéressant, en particulier dans notre étude qui consiste à plonger un concept traditionnellement objet dans un langage modulaire. Une telle comparaison n'est pas effectuée ici mais un chapitre lui est entièrement consacré dans [2].

Un système de modules à la `ML` peut être vu comme un langage fonctionnel typé, relativement indépendant du langage de base sous-jacent [7]. Les éléments de base d'un tel langage sont les *structures*. Une structure regroupe un ensemble de définitions comme un type de données muni d'opérations le manipulant. Par exemple, la structure `OrdInt` ci-dessous définit le type des entiers relatifs munis de sa relation d'ordre usuelle :

```
module Entier_Ordonné = struct
  type t = int
  let egal = (=)
  let inf = (<)
end
```

On accède à la fonction `egal` en utilisant la notation pointée `OrdInt.egal`. Le type d'une structure est appelé une *signature* et correspond à son interface. Ainsi, le type de module `ORDONNÉ` ci-dessous est une signature possible pour `Entier_Ordonné` :

```
module type ORDONNÉ = sig
  type t
  val egal : t → t → bool
  val inf : t → t → bool
end
```

L'abstraction du type `t` permet de rendre la signature `ORDONNÉ` plus générique. Un autre exemple est celui des dictionnaires polymorphes dont une signature possible est :

```
module type DICTIONNAIRE = sig
  type clé                                (* type des clés du dictionnaire *)
  type α t                                      (* type d'un dictionnaire *)
```

```
  val vide : α t                                          (* un dictionnaire vide *)
  val trouve : clé → α t → α           (* trouve une donnée dans un dictionnaire *)
  val ajoute : clé → α → α t → α t   (* ajoute une donnée dans un dictionnaire *)
end
```

En plus des définitions du langage de base, les structures et les signatures peuvent également contenir des modules et des types de module. Par exemple, une autre signature possible pour les dictionnaires polymorphes est :

```
sig
  module Clé : ORDONNÉ                   (* implantation des clés du dictionnaire *)
  type α t
  val vide : α t
  val trouve : Clé.t → α t → α
  val ajoute : Clé.t → α → α t → α t
end
```

Les fonctions du langage de modules sont appelées *foncteurs*. Un foncteur est un module paramétré par un autre. Par exemple, la signature `DICTIONNAIRE` donnée précédemment peut être implantée en utilisant des arbres binaires équilibrés s'il est possible de comparer des clés deux à deux. On peut donc écrire un module paramétré par une structure représentant les clés du dictionnaire :

```
module Construire_Dictionnaire(Clé : ORDONNÉ) = struct
  type clé = Clé.t
  type α t = Vide | Nœud of clé × α × α t × α t
  let vide = Vide
  let trouve = ... Clé.inf ...
  let ajoute = ...
end
```

On obtient une implantation des dictionnaires à clés entières en appliquant ce foncteur à la structure `Entier_Ordonné` :

```
module Dictionnaire_D'Entiers = Construire_Dictionnaire(Entier_Ordonné)
```

Un problème se pose si on souhaite donner une signature à `Dictionnaire_D'Entiers`. En effet, cette signature ne peut pas être directement `DICTIONNAIRE` car, le type `clé` demeurant abstrait, il ne serait pas possible d'utiliser ce module. On a besoin de rendre ce type manifeste (*i.e.* d'enlever son abstraction) en utilisant une contrainte de signature :

```
module Dictionnaire_D'Entiers : DICTIONNAIRE with type clé = int
```

Structures, foncteurs et applications de foncteurs sont les trois principales constructions d'un langage de modules. En `OCaml`, il existe une autre construction qui facilite leurs utilisations : l'instruction `include`. Cette dernière ajoute dans la structure (*resp.* signature) courante les définitions présentes dans la structure (*resp.* signature) spécifiée. Par exemple,

```
module X = struct
  include Entier_Ordonné
  (* [X] importe toutes les définitions de [Entier_Ordonné]. *)
  let sup = (>)
end
```

définit un module `X` possédant la signature `Entier_Ordonné` augmentée de la fonction `sup`.

4. Implantation du Modèle-Vue-Contrôleur avec des modules

Proposer une implantation du MVC peut paraître étrange, voir absurde, car un des principes de base des patrons de conception est de *ne pas* proposer d'implantation d'une solution mais seulement un schéma de cette solution. Dans cette section, nous montrons cependant comment il est possible d'implanter un MVC générique, s'adaptant à un modèle et à une vue quelconques par de simples applications de foncteurs.

Pour cela, nous séparons le modèle (*resp.* la vue) en deux parties : la première partie, appelée *domaine du modèle* (*resp. domaine de la vue*), implante le modèle (*resp.* la vue) sans prendre en compte le MVC. Cette partie est donc laissée à la charge du programmeur de l'application. La deuxième partie enrobe la première en définissant ou redéfinissant les opérations nécessaires au modèle (*resp.* à la vue) pour prendre en compte le MVC. C'est l'implantation de cette partie que nous présentons ici. Par abus de langage, nous appelons désormais *modèle* (resp. *vue*) cette seconde partie. La figure 2 présente le découpage du MVC incluant les parties sus-mentionnées.

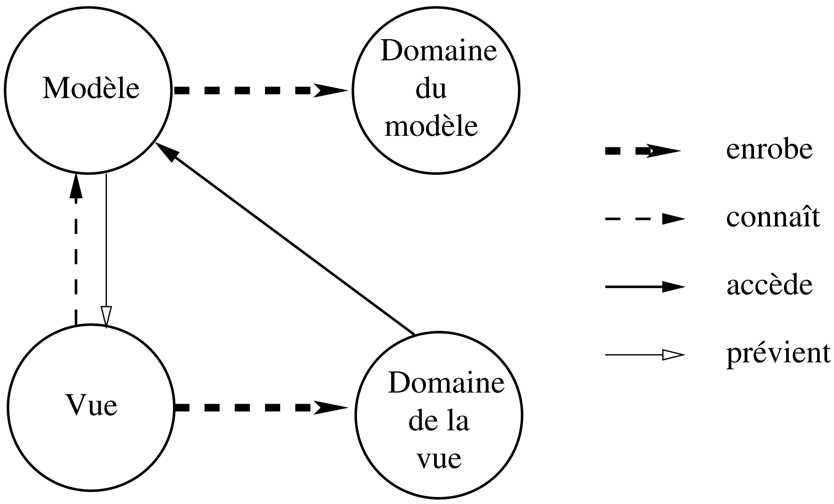


FIG. 2 – Découpage entre modèle, domaine du modèle, vue et domaine de la vue

4.1. Modèle

Le modèle doit assurer deux missions : il doit faire le lien avec le domaine sous-jacent d'une part, et gérer les vues représentant ce domaine d'autre part. La première mission ne pose pas de problème. La seconde consiste à ajouter ou à supprimer une vue ainsi qu'à les mettre à jour dès que le domaine est modifié. Nous avons donc, pour chaque modèle, un ensemble de vues à gérer [1], chacune d'elles pouvant être représentée par une seule fonction *màj* de mise à jour. Ainsi, pour associer une nouvelle vue à un ensemble de vues, il suffit de fournir sa *màj*. Cependant, comme la comparaison de fonctions est impossible, la suppression d'une vue n'est pas réalisable car on ne peut pas retrouver sa *màj* dans un ensemble de vues. Une solution consiste à associer, lors de l'ajout d'une nouvelle vue, un élément unique comparable (par exemple un entier) à sa *màj*, autorisant ainsi une possible suppression ultérieure. Ainsi, une signature possible pour l'ensemble de vues est

```
module type VUES = sig
```

[1] cet ensemble de vues correspond aux *dépendants* de `Smalltalk-80`

```
  type vue                          (* type de l'élément unique associé à une vue *)
  type t                                              (* type d'un ensemble de vues *)
  val créer : unit → t                           (* crée un nouvel ensemble de vues *)
  val nouvelle : t → (unit → unit) → vue                (* ajoute une nouvelle vue *)
  val enlever : t → vue → unit                                   (* enlève une vue *)
  val mettre_à_jour : t → unit                   (* met à jour un ensemble de vues *)
end
```

et il est possible de fournir une implantation de cette signature en utilisant une table de hachage :

```
module Vues : VUES = struct
  type vue = int
  type t = (vue, unit → unit) Hashtbl.t
  let créer () = Hashtbl.create 7
  let ajouter = Hashtbl.add
  let nouvelle =
    let cpt = ref 0 (* permet d'assigner un entier unique à chaque màj *) in
    fun vues màj → màj () (***); incr cpt; ajouter vues !cpt màj; !cpt
  let enlever = Hashtbl.remove
  let itérer = Hashtbl.iter
  let mettre_à_jour = itérer (fun _ màj → màj ())
end
```

Cette implantation ne présente pas de difficulté, en particulier grâce à l'ordre supérieur qui permet de manipuler facilement *màj*. Le seul point à signaler est l'appel immédiat à *màj* lors de la création d'une nouvelle vue (indiqué par `(***)`). Cet appel n'est pas indispensable mais permet d'éviter à chaque vue d'avoir à le faire au moment de sa création, simplifiant ainsi la programmation des vues.

Une fois muni de cet ensemble de vues, il est aisé d'obtenir un modèle générique possédant les opérations nécessaires pour mener à bien ses deux missions :

```
module type MODÈLE = sig
  type α t               (* type d'un modèle paramétré par le domaine sous-jacent *)
  val créer : α → α t                        (* crée un modèle basé sur un domaine *)
  val domaine : α t → α                            (* accès au domaine sous-jacent *)
  val modifier : α t → α → unit                 (* modifie le domaine sous-jacent *)
  val nouvelle_vue : α t → (unit → unit) → Vues.vue  (* ajoute une nouvelle vue *)
  val enlever_vue : α t → Vues.vue → unit                        (* enlève une vue *)
end

module Modèle : MODÈLE = struct
  type α t = { mutable domaine : α; vues : Vues.t }
  let domaine m = m.domaine
  let créer d = { domaine = d; vues = Vues.créer () }
  let modifier m d = m.domaine ← d; Vues.mettre_à_jour m.vues
  let nouvelle_vue m = Vues.nouvelle m.vues
  let enlever_vue m = Vues.enlever m.vues
end
```

Cette implantation est basée sur une structure de données impérative et modifie le domaine en place lors d'une modification. Bien que ce ne soit pas traité dans cet article, il est possible — sans difficulté supplémentaire — d'utiliser un modèle persistant ne modifiant pas le domaine en place et retournant un nouveau modèle.

4.2. Vue

Autant aucun prérequis sur le domaine sous-jacent n'est nécessaire pour implanter un modèle, autant il est nécessaire de supposer l'existence de quelques opérations de base sur le domaine de la vue sous-jacente pour implanter une vue. En effet, la vue doit principalement modifier le modèle qu'elle représente — et plus précisément l'ensemble de vues de ce dernier — lorsque le domaine de la vue est créé ou détruit ou bien lorsqu'il change de modèle sous-jacent (pour en représenter un autre). On effectue ces modifications en redéfinissant les trois opérations sus-mentionnées du domaine de la vue. Par conséquent, la vue et le domaine de la vue possèdent une signature commune :

```
module type VUE_GÉNÉRIQUE = sig
  type domaine_modèle                    (* type du domaine du modèle sous-jacent *)
  type t                                                        (* type de la vue *)
  val créer : domaine_modèle Modèle.t → t               (* crée une nouvelle vue *)
  val détruire : t → unit                                     (* détruit une vue *)
  val changer : t → domaine_modèle Modèle.t → unit
                                        (* change le modèle sous-jacent à une vue *)
end
```

En outre, le domaine de la vue doit pouvoir être mis à jour et fournir un accès au modèle :

```
module type DOMAINE_VUE = sig
  include VUE_GÉNÉRIQUE
  val mettre_à_jour : t → unit                             (* met la vue à jour *)
  val modèle : t → domaine_modèle Modèle.t      (* accès au modèle sous-jacent *)
end
```

tandis que la vue doit fournir un accès au domaine de la vue :

```
module type VUE = sig
  include VUE_GÉNÉRIQUE
  type domaine                                       (* type du domaine de la vue *)
  val domaine : t → domaine                         (* accès au domaine de la vue *)
end
```

Tout comme pour le modèle, il est possible de donner une implantation générique d'une vue. Cette dernière doit cependant dépendre du domaine de la vue. Comme expliqué précédemment, la vue se contente principalement d'enrober les trois opérations de base du domaine de la vue de manière à effectuer les modifications nécessaires sur l'ensemble des vues du modèle. Son implantation ne présente donc pas de difficulté particulière.

```
module Vue(D : DOMAINE_VUE) :
  VUE with type domaine_modèle = D.domaine_modèle and type domaine = D.t =
struct
  type domaine_modèle = D.domaine_modèle
  type domaine = D.t
  (* une vue associe à son domaine l'élément unique correspondant dans le modèle *)
  type t = { domaine : D.t; mutable id : Vues.vue }
  let domaine v = v.domaine

  (* fonctions auxiliaires *)
  let nouveau_domaine m d = Modèle.nouvelle_vue m (fun () → D.mettre_à_jour d)
  let enlever_du_modèle v = Modèle.enlever_vue (D.modèle v.domaine) v.id
```

```
  (* redéfinition des trois opérations de base du domaine *)
  let créer m = let d = D.créer m in { domaine = d; id = nouveau_domaine m d }
  let détruire v = enlever_du_modèle v; D.détruire v.domaine
  let changer v m =
    enlever_du_modèle v; D.changer v.domaine m; v.id ← nouveau_domaine m v.domaine
end
```

4.3. Exemple : représentation d'un entier

Dans cette section, nous donnons un court exemple de domaine du modèle et de domaine de la vue. Cet exemple modélise un entier et en donne une représentation graphique sous la forme d'une zone de texte éditable. La modélisation du domaine est triviale :

```
module type DOMAINE_MODÈLE = sig
  type t                                                  (* type d'un entier *)
  val créer : int → t                                (* crée un nouvel entier *)
  val de_chaîne : string → t                 (* convertit une chaîne en entier *)
  val vers_chaîne : t → string             (* l'entier sous forme d'une chaîne *)
end

module Domaine_Modèle : DOMAINE_MODÈLE = struct
  type t = int
  let créer x = x
  let de_chaîne = int_of_string
  let vers_chaîne = string_of_int
end
```

L'interface graphique utilise la bibliothèque LablGTK2 [4] mais aucune connaissance particulière de cette bibliothèque n'est requise pour comprendre son implantation :

```
module type DOMAINE_VUE = sig
  include DOMAINE_VUE
  val empaqueter : t → GPack.box → unit
                                     (* empaquète la vue dans une boîte graphique *)
end

module Domaine_Vue : DOMAINE_VUE with type domaine_modèle = Domaine_Modèle.t = struct
  module D = Domaine_Modèle
  type modèle = D.t Modèle.t
  (* une vue est représentée par un modèle et une zone de texte éditable *)
  type t = { mutable modèle : modèle; zone : GEdit.entry }

  let créer m =
    (* création de la zone de texte *)
    let z = GEdit.entry () in
    (* création de la vue *)
    let v = { modèle = m; zone = z } in
    (* modifie le modèle sur modification du contenu de la zone de texte *)
    let modifier_modèle _ =
      try Modèle.modifier v.modèle (D.de_chaîne z#text) with Failure _ → ()
```

```
    in
    ignore (z#connect#changed ~callback:modifier_modèle);
    (* retourne la vue *)
    v

  let modèle v = v.modèle
  let détruire v = v.zone#destroy ()
  let changer v m = v.modèle ← m
  let mettre_à_jour v = v.zone#set_text (D.vers_chaîne (Modèle.domaine v.modèle))
  let empaqueter v (b:GPack.box) = b#pack v.zone#coerce
end
```

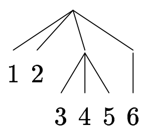
Bien que, en pratique, cet exemple simpliste ne nécessite sûrement pas d'utiliser le MVC, il permet de mettre en évidence deux points forts de la modélisation présentée :

1. Il montre que le domaine du modèle et le domaine de la vue peuvent fournir leurs propres opérations (par exemple, `empaqueter` pour le domaine de la vue). Ainsi, appliquer le MVC ne limite pas le pouvoir d'expression d'un des deux domaines.
2. Il montre concrètement que le domaine du modèle et le domaine de la vue n'ont à s'occuper que de leurs problèmes respectifs et ne sont pas pollués par des opérations concernant l'implantation du MVC. On obtient donc ainsi un code concis et lisible, par conséquent, facilement maintenable et moins bogué qu'à l'ordinaire.

5. Aggrégation d'entités

En pratique, il est fréquent qu'une entité soit constituée de sous-entités. Ainsi, pour poursuivre l'exemple sur les entiers présenté à la section 4.3, on peut souhaiter modéliser des listes ou des arbres d'entiers et en afficher des vues. Dans beaucoup de cas, le MVC présenté à la section précédente le permet : aucune modification du modèle et de la vue n'est nécessaire et il suffit d'implanter les domaines qui conviennent. C'est par exemple le cas si on souhaite modéliser une liste d'entiers affichée sous la forme d'une liste de zones de texte éditables. On pourra en trouver une implantation en annexe A.

Cependant, le MVC présenté jusqu'ici ne fonctionne plus correctement dans le cas où une modification du modèle d'une entité e implique une actualisation des vues d'une entité contenant e. C'est par exemple le cas pour un arbre d'entiers valué aux feuilles qu'on souhaite représenter dans une seule zone de texte, comme dans l'exemple suivant où l'arbre



est représenté par la chaîne `[ 1 2 [ 3 4 5 ] [ 6 ] ]` affichée dans une zone de texte. En effet, dans un tel cas, la modification d'un des entiers composant l'arbre implique un réaffichage des vues de l'arbre, et pas uniquement un réaffichage des vues de cet entier [2]. Pour régler ce problème, plusieurs modifications du modèle doivent être effectuées.

Le modèle ne peut pas être géré de la même manière dans le cas de base que nous avons présenté jusqu'alors et dans le cas qui nous intéresse ici. En effet, dans ce dernier cas, effectuer une opération sur le modèle d'une aggrégation c d'entités nous conduit à l'effectuer aussi sur chacun des sous-modèles

[2] En réalité, sur cet exemple précis, il est possible de ne réafficher que les vues de cet entier et pas la vue de l'arbre au prix d'un travail assez important sur la chaîne de caractères représentant l'arbre.

de c[3]. Par exemple, lorsqu'une nouvelle vue v est ajoutée au modèle de c, il faut aussi l'ajouter à chaque sous-modèle de c de façon à mettre v à jour lorsque qu'un des sous-modèles est modifié. Par suite, il faut donc disposer d'une fonction d'itération sur les sous-modèles du domaine du modèle d de c et, donc, connaître le type de d. C'est la raison pour laquelle le type du modèle ne peut plus être polymorphe par rapport au domaine sous-jacent : à présent, le modèle doit être un foncteur paramétré par son domaine.

En outre, d'après ce qui précède, le modèle ne peut pas être implanté de la même façon dans le cas de base étudié à la section 4.1 et dans le cas qui nous intéresse ici, celui d'une aggrégation d'entités. Il faut donc deux foncteurs différents, appelés respectivement `Base` et `Aggrégat`. La signature à donner au modèle est donc la suivante :

```
module Modèle : sig
  module Base(D : sig type t end) : MODÈLE with type domaine = D.t
  module Aggrégat
    (M : MODÈLE)                                 (* implantation des sous-modèles *)
    (D : sig type t val itérer_sous_modèles : (M.t → unit) → t → unit end)
                                              (* implantation du domaine sous-jacent *)
    : MODÈLE with type domaine = D.t
end
```

La signature `MODÈLE` est similaire à la précédente mais le type `t` n'est plus polymorphe, le type `domaine` ayant été introduit pour remplacer la variable de type α.

Il revient à la charge du programmeur d'une entité de décider quel foncteur — `Base` ou bien `Aggrégat` — appliquer. L'implantation de `Base` est similaire à celle de la section 4.1 et n'est pas redonnée ici. L'implantation de `Aggrégat` consiste à redéfinir certaines opérations de `Base` de manière à mettre à jour les sous-modèles d'un aggrégat. Pour ce faire, il est nécessaire d'avoir accès aux deux fonctions de `Vues` non exportées dans `VUES` (à savoir `ajouter` et `itérer`, *cf.* la section 4.1). Cette signature devient donc :

```
module type VUES = sig
  (* ... ; identique à la section 4.1 *)
  val ajouter : t → vue → (unit → unit) → unit
  val itérer : (vue → (unit → unit) → unit) → t → unit
end
```

De même, il est nécessaire d'avoir accès à un enrobage de `ajouter` dans le modèle, similaire à ceux existant déjà à la section 4.1 pour `nouvelle` et `enlever` :

```
module type MODÈLE = sig
  (* ... ; similaire à la section 4.1 nonobstant l'absence de polymorphisme *)
  val ajouter_vue : t → Vues.vue → (unit → unit) → unit
end
```

Muni de ces quelques fonctions supplémentaires, il est possible d'implanter `Aggrégat` :

```
module Aggrégat
  (M : MODÈLE)
  (D : sig type t val itérer_sous_modèles : (M.t → unit) → t → unit end) =
struct
  include Base(D)                       (* importation de [Base] sur le bon domaine *)
```

[3]plus exactement, sur le modèle de chaque sous-entité du domaine du modèle de c

```
  (* fonctions auxiliaires *)
  let appliquer f m = D.itérer_sous_modèles f m.domaine
  let enlever_des_sous_modèles m v = appliquer (fun m → M.enlever_vue m v) m
  let ajouter_aux_sous_modèles m v màj = appliquer (fun m → M.ajouter_vue m v màj) m

  (* redéfinition des fonctions qui diffèrent de [Base(D)] *)
  let enlever_vue m v = enlever_vue m v; enlever_des_sous_modèles m v
  let ajouter_vue m v màj = ajouter_vue m v màj; ajouter_aux_sous_modèles m v màj
  let nouvelle_vue m màj =
    let v = nouvelle_vue m màj in ajouter_aux_sous_modèles m v màj; v
  let modifier m d =
    (* enlever chaque vue de [m] de ses anciens sous_modèles *)
    Vues.itérer (fun v _ → enlever_des_sous_modèles m v) m.vues;
    m.domaine ← d;
    (* ajouter chaque vue de [m] à ses nouveaux sous_modèles et les mettre à jour *)
    Vues.itérer (fun v màj -> ajouter_aux_sous_modèles m v màj; màj ()) m.vues
end
```

Au contraire du modèle, la vue n'a pas à être modifiée dans le cas d'un aggrégat. Cependant, étant donné que l'implantation du modèle n'est plus unique, il est dorénavant nécessaire de paramétrer la vue par le modèle qu'elle représente :

```
module Vue(M : MODÈLE)(V : DOMAINE_VUE with type modèle = M.t) :
  VUE with type modèle = M.t and type domaine = V.t
```

En outre, comme on peut le constater dans le code ci-dessus, le type `domaine_modèle` a été remplacé par `modèle` dans `VUE_GÉNÉRIQUE` afin de simplifier le code.

En guise d'exemple, l'annexe B fournit l'implantation des arbres d'entiers valués aux feuilles qu'on souhaite représenter dans une zone de texte. La simplicité du code montre à quel point les foncteurs présentés dans cet article facilitent l'implantation d'une entité, même lorsqu'il s'agit d'un aggrégat.

6. Création d'entités basées sur le MVC

Étant donnée une implantation `Domaine_Modèle` d'un domaine du modèle possédant une certaine signature `DOMAINE_MODÈLE` ainsi qu'une implantation `Domaine_Vue` d'un domaine de la vue portant sur ce domaine et possédant une certaine signature `DOMAINE_VUE`, il est possible de donner le schéma général d'une entité. Ce schéma est très simple et consiste uniquement à effectuer des applications de foncteurs. Dans le cas de base comme l'exemple sur les entiers donné à la section 4.3, on a :

```
module type ENTITÉ = sig
  module DM : DOMAINE_MODÈLE
  module M : MODÈLE with type domaine = DM.t
  module DV : DOMAINE_VUE with type modèle = M.t
  module V : VUE with type modèle = M.t and type domaine = DV.t
end

module Entité : ENTITÉ = struct
  module DM = Domaine_Modèle
  module M = Modèle.Base(DM)
```

```
  module DV = Domaine_Vue(M) 4
  module V = Vue(M)(DV)
end
```

Pour un aggrégat, le schéma de code est similaire quoique légèrement différent. Ainsi, pour des arbres d'entiers similaires à ceux présentés en annexe B, et en supposant que `Entier.Entité` désigne le module `Entité` ci-dessus, on a :

```
module type ENTITÉ = sig
  module DM : DOMAINE_MODÈLE with type entier = Entier.Entité.M.t
  module M : MODÈLE with type domaine = DM.t
  module DV : DOMAINE_VUE with type modèle = M.t
  module V : VUE with type modèle = M.t and type domaine = DV.t
end

module Entité : ENTITÉ = struct
  module DM = Domaine_Modèle
  module M = Modèle.Aggrégat(Entier.Entité.M)(DM)
  module DV = Domaine_Vue(M)
  module V = Vue(M)(DV)
end
```

Ces schémas de code sont les seuls à devoir être réécrits à chaque fois par le programmeur utilisant le MVC. Il est impossible d'être plus générique en créant un foncteur paramétré par `Domaine_Modèle` et `Domaine_Vue` et retournant `Entité` pour des raisons de typage : on ne peut pas connaître de manière à la fois statique et générique leurs signatures respectives `DOMAINE_MODÈLE` et `DOMAINE_VUE`. Ainsi, ces schémas de code ne peuvent pas être factorisés et doivent être dupliqués, un par entité.

7. Conclusion

Cet article montre comment il est possible d'implanter un patron de conception, en l'occurence le « Modèle-Vue-Contrôleur », de manière générique en utilisant un système de modules à la `ML`. Par là-même, il infirme la croyance trop souvent répandue qu'un tel concept est forcément lié au paradigme de la programmation objet. De plus, en écrivant une partie du patron de manière générique (ce qui limite à quelques applications de foncteurs le schéma de code qui doit être refait par le programmeur pour chaque entité), l'implantation réalisée ici n'est pas plus difficile à utiliser, et c'est un euphémisme, qu'une approche traditionnelle utilisant des objets.

Cette approche souffre néanmoins de quelques points négatifs : l'accès aux domaines (du modèle ou de la vue) peut s'avérer assez lourd s'il faut utiliser les fonctions d'accès fournies par le modèle ou la vue. En pratique, on s'aperçoit cependant qu'une telle situation n'est pas contraignante si un bon travail d'interfaçage est effectué. En outre, les indirections engendrées par de tels accès ont un coût non nul à l'exécution, indirections qui doivent être ajoutées à celles des applications de foncteur qui en engendrent également. Mais, tout d'abord, le surcoût engendré n'est généralement pas si important en pratique. Ensuite, un tel surcoût est inéluctable, la programmation étant, à mon avis, un compromis permanent entre efficacité et clarté. Puis, ce surcoût n'est *a priori* pas plus important — si ce n'est moindre — qu'une implantation du MVC dans le paradigme objet étant données les indirections générées par le mécanisme de liaison tardive. Enfin, il est toujours possible d'utiliser un *défonctoriseur* comme `ocamldefun` [14], pour diminuer le coût des indirections liées aux applications de foncteurs.

[4] Afin d'être compatible avec la nouvelle implantation du modèle présentée dans cette section, le domaine de la vue d'un entier doit à présent prendre le modèle en paramètre, ce qui n'était pas le cas dans la section 4.3.

Pour terminer, le lecteur intéressé peut trouver à l'url `http://www.lri.fr/~signoles/mvc.fr.html` le code complet du MVC et de l'exemple présentés dans cet article. En outre, ce MVC est utilisé dans une application réelle (un jeu), actuellement en cours de développement. À l'heure actuelle, cette application fait de l'ordre de 5000 lignes de code et sept entités, dont quatre aggrégats, utilisent le MVC. Dans cette application, un nombre quelconque de vues peuvent être montrées à l'utilisateur. Dans ce contexte, déléguer tous les problèmes de mise à jour au MVC est un gain important, aussi bien pour la correction du code que pour la vitesse du développement. En outre, utiliser le MVC sous la forme présentée dans cet article a permis de réduire d'environ 20% la taille de l'application (un millier de lignes de code gagnées) par rapport à une version précédante utilisant le MVC sans foncteur.

Références

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[2] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'Applications avec Objective Caml.* O'Reilly France, 2000. `http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.html`.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] Jacques Garrigue, Benjamin Monate, Olivier Andrieu, and Jun Furuse. LablGTK2. `http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/lablgtk.html`.

[5] Adele J. Goldberg and David Robson. *Smalltalk-80 : The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[6] Glenn E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80. *Journal of Object Oriented Program*, 1(3) :26–49, 1988.

[7] Xavier Leroy. A Modular Module System. *Journal of Functional Programming*, 10(3) :269–303, 2000.

[8] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System.* Institut National de Recherche en Informatique et en Automatique — Projet Cristal. `http://caml.inria.fr`.

[9] David Mac Queen. Modules for Standard ML. In *ACM Conference on LISP and Functional Programming*, pages 198–207, 1984.

[10] Sun Microsystems. How to Use Lists. `http://java.sun.com/docs/books/tutorial/uiswing/components/list.html`.

[11] Robin Milner, Mads Tofte, Robert Harper, and David Mac Queen. *The Definition of Standard ML (Revised).* MIT Press, May 1997.

[12] Tim-Philip Muller and SooHyoung Ho. GTK+ 2.0 Tree View Tutorial using Ocaml. `http://compiler.kaist.ac.kr/~shoh/ocaml/lablgtk2/treeview-tutorial`.

[13] Andreas Rossberg. Standard ML vs. Objective Caml. `http://www.ps.uni-sb.de/~rossberg/SMLvsOcaml.html`.

[14] Julien Signoles. Calcul statique des applications de modules paramétrés. In *Journées Francophones des Langages Applicatifs*, 2003.

A. Liste d'entiers représentée par une liste de zones de texte éditables

```
module type DOMAINE_MODÈLE = sig
```

```
  type entier                                                    (* type d'un entier *)
  type t                                                (* type d'une liste d'entiers *)
  val créer : entier list → t                 (* crée une nouvelle liste d'entiers *)
  val valeur : t → entier list                      (* convertit une liste d'entiers *)
end

module Domaine_Modèle :
  DOMAINE_MODÈLE with type entier = Entier.Entité.DM.t Modèle.t =
struct
  type entier = Entier.Entité.DM.t Modèle.t
  type t = entier list
  let créer x = x
  let valeur x = x
end

module type DOMAINE_VUE = sig
  include DOMAINE_VUE
  val empaqueter: t → GPack.box → unit
end

module Domaine_Vue : DOMAINE_VUE with type modèle = Domaine_Modèle.t Modèle.t = struct
  module D = Domaine_Modèle
  module E = Entier.Entité

  (* Une vue [v] d'une liste [l] d'entiers doit gérer la liste de ses sous-vues.
     Chaque sous-vue correspond à une vue d'un des entiers de [l].
     Le module [Sous_vues] associe à chaque entier la vue correspondante. *)
  module Sous_vues : sig
    type t
    val créer : unit → t
    val ajouter :  t → D.entier → E.V.t → unit
    val détruire : t → unit
  end =
  struct
    type t = (D.entier, E.V.t) Hashtbl.t
    let créer () = Hashtbl.create 7
    let ajouter = Hashtbl.add
    let détruire t = Hashtbl.iter (fun _ v → E.V.détruire v) t; Hashtbl.clear t
  end

  type domaine_modèle = D.t
  type t = { mutable modèle : D.t Modèle.t; sous_vues : Sous_vues.t; boîte: GPack.box }

  let créer m = { modèle = m; sous_vues = Sous_vues.créer (); boîte = GPack.vbox () }
  let modèle v = v.modèle
  let détruire v = Sous_vues.détruire v.sous_vues; v.boîte#destroy ()
  let changer v m = v.modèle ← m
  let empaqueter v (b:GPack.box) = b#pack v.boîte#coerce
  let mettre_à_jour v =
    (* tout détruire puis tout reconstruire
       car on ne sait pas quels éléments sont à mettre à jour  *)
```

```
    Sous_vues.détruire v.sous_vues;
    List.iter
      (fun n →
         let sv = E.V.créer n in
         E.DV.empaqueter (E.V.domaine sv) v.boîte; Sous_vues.ajouter v.sous_vues n sv)
      (D.valeur (Modèle.domaine v.modèle))
end
```

B. Arbre d'entiers représenté par une zone de texte

```
module type DOMAINE_MODÈLE = sig
  type entier                                                  (* type d'un entier *)
  type t = E of entier | L of t list                  (* type d'un arbre d'entiers *)
  val itérer_sous_modèles : (entier → unit) → t → unit
                                          (* itère sur tous les entiers d'un arbre *)
  val vers_chaîne : t -> string                   (* convertit un arbre en chaîne *)
end

module Domaine_Modèle : DOMAINE_MODÈLE with type entier = Entier.Entité.M.t = struct
  type entier = Entier.Entité.M.t
  type t = E of entier | L of t list
  let rec itérer_sous_modèles f = function
    | E n → f n
    | L l → List.iter (itérer_sous_modèles f) l
  let rec vers_chaîne = function
    | E n -> Entier.Entité.DM.vers_chaîne (Entité.Entité.M.domaine n)
    | L l -> List.fold_left (fun s a -> s ^ vers_chaîne a ^ " ") "[ " l ^ "]"
end

module type DOMAINE_VUE = sig
  include DOMAINE_VUE
  val empaqueter : t → GPack.box → unit
end

module Domaine_Vue(M : MODÈLE with type domaine = Domaine_Modèle.t)
  : DOMAINE_VUE with type modèle = M.t =
struct
  type modèle = M.t
  type t = { mutable modèle : modèle; zone : GEdit.entry }
  let créer m = { modèle = m; zone = GEdit.entry () }
  let modèle v = v.modèle
  let détruire v = v.zone#destroy ()
  let changer v m = v.modèle ← m
  let mettre_à_jour v =
    v.zone#set_text (Domaine_Modèle.vers_chaîne (M.domaine v.modèle))
  let empaqueter v (b:GPack.box) = b#pack v.zone#coerce
end
```