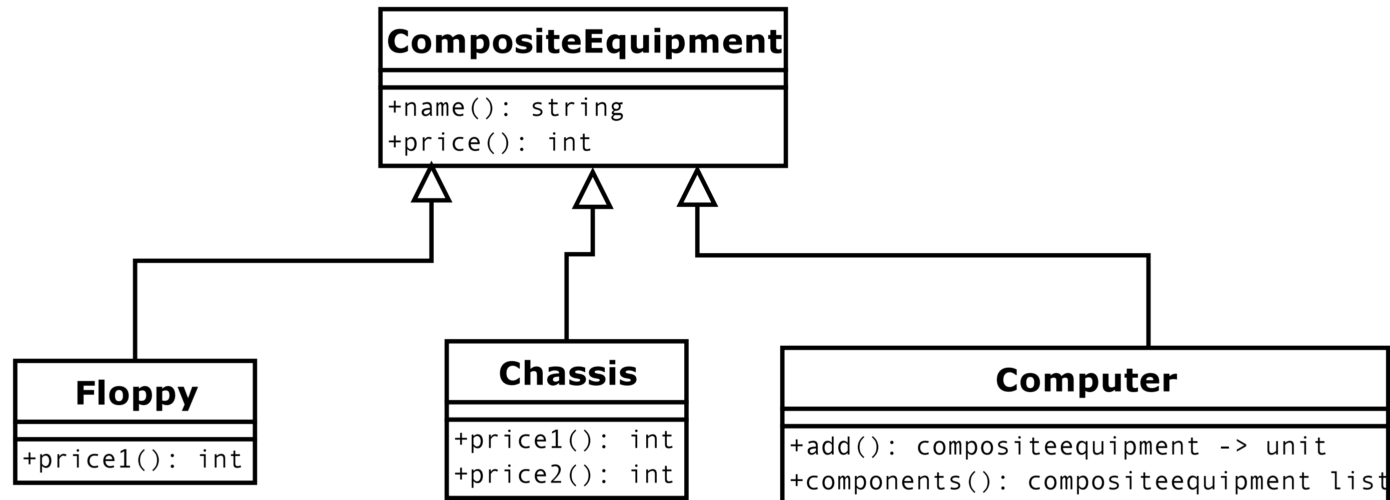
14 Design patterns

Quand on programme en orienté objet, dans des langages sans typage fort, il est très important de respecter une certaine discipline de programmation, telle qu'elle peut se trouver par exemple dans

> Design Patterns (1994)
> Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Un exemple : le prix à la tête du client



Étant donné un objet de la classe *CompositeEquipment*, nous voulons calculer son prix, en appliquant une politique de prix à la tête du client[a].

[a] ceci est la “business logic”dans 01 Informatique

Approche objet naïve

Chaque politique de prix p_i est réalisée par une méthode

$$prix_p_i : int$$

- on doit ajouter une nouvelle méthode à chaque classe…
- on doit recompiler toutes les classes !…
- *combien* de politiques de prix y-a-t il ?

C'est inacceptable !

On doit y arriver sans ajouter des méthodes ad hoc pour chaque politique des prix aux classes !

Approche hybride avec filtrage Ocaml

```
(** Price policies with pattern matching *)

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

class floppy () =
  object
    inherit equipment
    method name () = "floppy"
    method price1 () = 10
    method price () = 5
  end
```

```
class chassis () =
  object
    inherit equipment
    method name () = "chassis"
    method price2 () = 15
    method price1 () = 12
    method price () = 10
  end

class virtual equipmentvisitor =
  object
    method virtual visitFloppy : floppy -> unit
    method virtual visitChassis : chassis -> unit
    method virtual gettotal : unit
  end
```

```
class pricevisitor () =
  object
    inherit equipmentvisitor
    val mutable totalprice = 0
    method visitFloppy f = totalprice <- totalprice + f#pric
    method visitChassis c = totalprice <- totalprice + c#pri
    method gettotal = print_int totalprice
  end

type equip = Floppy of floppy | Chassis of chassis

let visit e (v:equipmentvisitor) =
  (* in Java, this code would be a bunch of i-t-e and instac
  match e with
    (Floppy e1) -> v#visitFloppy e1
  | (Chassis e2) -> v#visitChassis e2
```

```
(* use of the classes *)

let a = new floppy ();;
let b = new chassis ();;

let equips = [Floppy a ; Chassis b];;

let v = new pricevisitor();;
List.iter (fun e -> visit e v) equips;;

v#gettotal;;
```

```
(* If we go this way, we can be much more concise: *)

let visitprice e tot =
  match e with
    (Floppy e1)  -> tot + e1#price1 ()
  | (Chassis e2) -> tot + e2#price2 ()
;;

List.fold_right visitprice equips 0;;
```

Le design pattern des visiteurs

Problème : nous avons une structure de données complèxe, qui permet de construire des grappes G d'objets entrelacés de classes différentes.
Nous souhaitons maintenant définir *une fois pour toutes* une architecture qui nous permet :

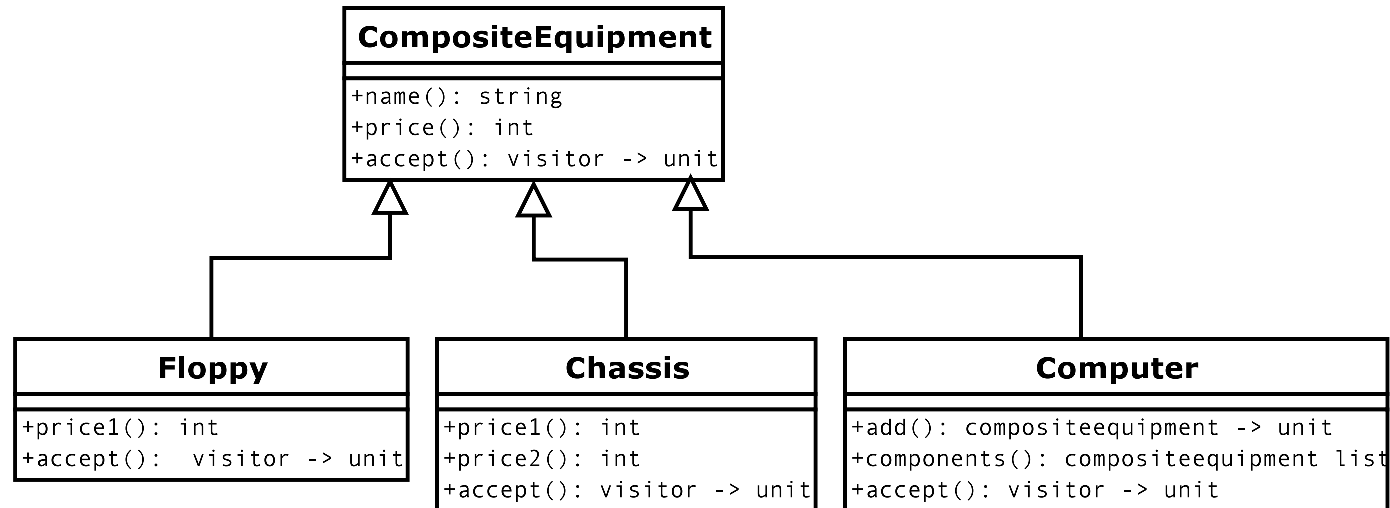
- de réaliser des *visites* de G, objet par objet,
- de réaliser sur chaque objet rencontré dans la visite une tâche *qui depend de la classe* de l'objet lui même,
- et tout cela *sans modifier la structure des classes* quand on change le code de la visite !

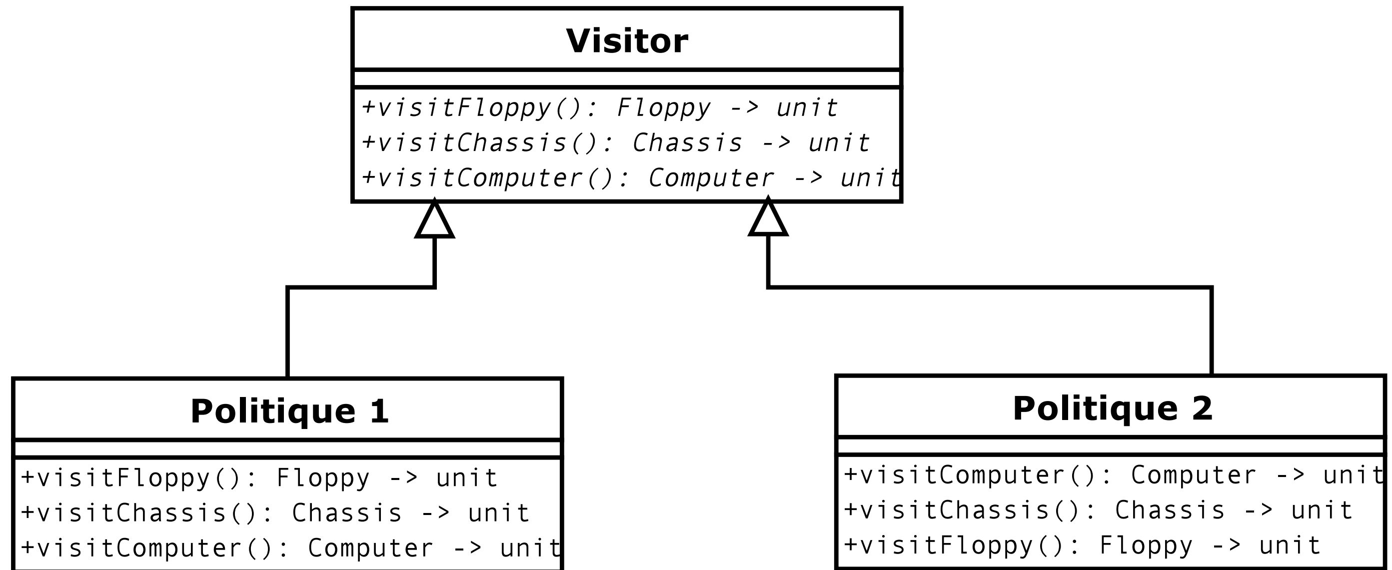
Cela nous permet d'ajouter aux classes qui interviennent dans G des nouvelles fonctionnalités, *sans ajouter des nouvelles méthodes*.

Revenons à notre *équipement composé*, qui peut être :

– un floppy, qui a un prix de liste et un prix de faveur ;
– un chassis, qui a un prix de liste et deux prix de faveur différents ;
– un ordinateur, qui lui, peut contenir plusieurs équipement composites

Le pattern des visiteurs résout le problème en ajoutant une méthode *accept* et une classe *visitor*, comme suit :





Quel est le code des méthodes *accept* et *visit*. . . ?

Maintenant, à vous de jouer en OCaml !