Programmation par motif et motifs de programmation.

Didier Rémy
2001 - 2002

http://cristal.inria.fr/~remy/mot/8/
http://www.enseignement.polytechnique.fr/profs/informatique/Didier.Remy/mot/8/

Slide 1

Cours	**Exercices**
1. Motif subjet/observateur 2. Protocoles couplés (Abonnements, Rappelle-moi, Relais) 3. Reconfiguration dynamique (Reconfiguration, Méthodes mutables, Crochets, Délégation) 4. Vues	1. Reconfiguration 2. Gestionnaire de fenêtres 3. Modèle multiplexé 4. Le général et son armée

Prologue

La programmation par motif est une notion suggestive et imprécise. Elle désigne le choix d'un protocole de communication ou d'assemblage entre différents types de composants.

Slide 2

Nous donnerons des exemples de motifs d'assemblage qui reviennent fréquemment. On pourrait parler simplement de techniques de programmation avec objets. Lorsqu'un motif particulier (ou une combinaison de plusieurs motifs) devient répétitif voire indispensable dans une application donnée, on peut parler de programmation par motif.

Il est souvent possible d'utiliser différents motifs pour résoudre un problème, mais le choix du motif retenu est rarement indifférent. Nous essayerons de comparer certains motifs entre eux dans des situations particulières.

Quelques exemples

Slide 3

Librairie de structures algébriques

Nous avons vu comment combiner la modularité apportée par les modules (générativité, abstraction) et par les classes (extension, liaison tardive) pour l'application à une librairie de calcul formel.

Une fois le style de programmation fixé, toutes les structures algébriques de la librairie devront suivre le même modèle.

Le protocole subjet/obervateur (voir ci-dessous), avec plusieurs variantes (volontaire ou autoritaire, à entrées multiples ou multiplexées).

De petites motifs qui reviennent fréquemment

– Les fonctions amis.
– Gestion d'événements.
– Abonnements à des services.

Le pattern sujet/observateur.

Présentation

Par exemple, il s'agit d'implémenter un gestionnaire de fenêtres.

Gestionnaire - Fenêtre
– les opérations sont réalisées par les fenêtres.
– les décisions sont prises par ou (ou plusieurs) gestionnaires.
L'implémentation doit rester extensible.

Slide 4

Le motif Subjet / Observateur est une généralisation :
– Les sujets exécutent les actions et rapportent aux observateurs
– Les observateurs décident et ordonnent aux sujets.
(Ici, le sujet est volontaire, i.e. il n'est pas obligé d'exécuter les ordres, mais une variante autoritaire serait similaire.)

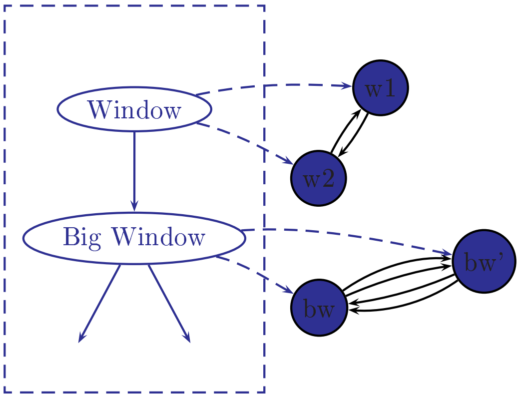
Le motif des listes est similaire, mais plus simple : `nil` *et* `cons` *sont des classes qui peuvent être raffinées,* e.g. *avec une méthode longueur ; de plus une nouvelle classe* `append` *peut être ajoutée. Cependant, la* *communication entre les différentes classes est moins intime que dans le motif subjet/observateur.*

Le motif du sujet/observateur est une généralisation des méthodes binaires où, cette fois-ci, ce sont des objets de classes différentes (et non de la même classe) qui interagissent entre eux.

Slide 5

Héritage

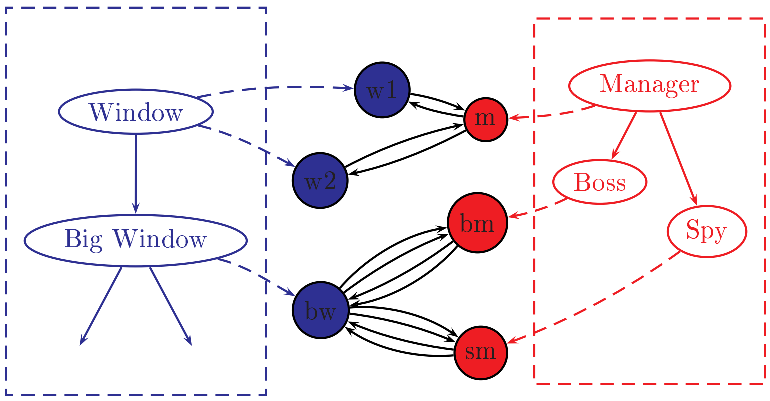
Slide 6



L'héritage fonctionne bien, même avec des méthode binaire : chaque classe est paramétrique en le type de self.

Héritage de composants

Slide 7



La difficulté est d'assurer que la connection entre les objets des classes liées reste correcte et extensible dans les classes héritées.

La classe en bleu est paramértique en le type des objets en rouge et vice-versa.

Classes collaboratrices

Slide 8

De façon générale, deux classes travaillent ensemble à l'accomplissement d'une même tâche. Les objets de l'une et de l'autre s'appellent mutuellement, de façon intime, donc en partageant une certaine interface.

Le protocole ainsi établi entre deux classes doit pouvoir s'étendre en un protocole plus fin entre deux sous-classes tout en préservant la sûreté des échanges (les messages doivent toujours être bien compris).

Pour le protocole sujet/observateur, on fournit un protocole minimal permettant au sujet et à l'observateur de communiquer, sans préciser les valeurs échangées initialement.

Par la suite, on étendra ce modèle par héritage à des situations concrètes en précisant les informations échangées.

Le sujet

Slide 9

C'est le sujet qui maintient à jour l'ensemble des observateurs à qui il devra rendre compte. Il utilise pour cela une variable d'instance `observateurs` et une méthode `ajoute` pour gérér ses abonnés, et une méthode `envoie` pour faire suivre un message à l'ensemble des abonnés.

```
class ['obs] sujet =
  object (self)
    val mutable observateurs = ([]: 'obs list)
    method ajoute m = observateurs <- m :: observateurs
    method envoie (message : 'obs -> 'mytype -> unit) =
      List.iter (fun obs -> message obs self) observateurs
  end;;
```

Le modèle est coopératif : ici c'est l'observateur qui maintient la liste de ses maîtres. On peut facilement inverser le modèle.

L'observateur

Slide 10

Dans la version dépouillée, l'observateur n'a rien à gérer et il n'y a pas encore de communication : c'est une classe vide.

```
class virtual ['subjet] observateur = object end;;
```

Le gestionnaire de fenêtre

Les fenêtres héritent du modèle du sujet :

```
class ['obs] fenêtre =
  object (self)
    inherit ['obs] sujet
    val mutable position = 0
    method déplace d =
      position <- position + d;
      self#envoie (fun x -> x#déplacé)
    method dessine =
      Printf.printf "{Position = %d}" position;
  end;;
```

Slide 11

La méthode `déplace` exécute le déplacement et rend compte d'un déplacement en envoyant par convention le message `déplacé` à l'observateur. La fenêtre a également une méthode dessine.

Le gestionnaire de fenêtre (suite)

Le gestionnaire hérite du modèle de l'`observateur` :

```
class ['sujet] gestionnaire =
  object
    inherit ['sujet] observateur
    method déplacé (s : 'sujet) : unit = s # dessine
  end ;;
```

Slide 12

La méthode `déplacé` de l'observateur prend acte du déplacement et demande à l'objet (reçu en argument) de se redessiner.

Raffinement du gestionnaire de fenêtre

On teste d'abord le modèle :

```
let fenêtre = new fenêtre in
fenêtre#ajoute (new gestionnaire); fenêtre#déplace 1;;
```

{Position = 1}− : unit = ()

Slide 13

Le classes `fenêtre` et `gestionnaire` peuvent être raffinées par héritage, indépendemment l'une de l'autre. Par exemple, on pourra :

– ajouter des méthodes `retaille` et `devant` aux fenêtres.
– améliorer la méthode dessine...
– faire prendre en compte de nouveaux messages au gestionnaire.

Le contremaître surveille

Slide 14

On peut aussi ajouter un nouveau maître indépendant, par exemple qui se contente de tracer les opérations effectuées :

```
class ['sujet] inspecteur =
  object
    inherit ['sujet] observateur
    method retaillé (s: 'sujet) = print_string "<R>"
    method déplacé (s: 'sujet) = print_string "<D>"
  end;;

let f = new fenêtre in
  f#ajoute (new gestionnaire);
  f#ajoute (new inspecteur :> 'a gestionnaire);
  f#déplace 2;;
```

<D>{Position = 2}− : unit = ()

Exercise 1 (Gestionnaire de fenêtres) *Dériver une classe* *`grande_fenêtre` qui implémente une méthode `retaille` (ce qui implique aussi de redéfinir la méthode `dessine` pour afficher la taille) qui rend compte en appelant la méthode `retaillée` de l'observateur. On ajoutera également une méthode `en_avant` pour placer une fenêtre sur le devant (représenté par un booléen).* Answer

Dériver une classe `grand_gestionnaire` capable de gérer les grandes fenêtres (par exemple, à la récep Answer

Slide 15

Étendre la classe `inspecteur` en `big_brother` qui trace tous les ordres. Answer □

Exercise 2 (Modèle multiplexé) *Un problème est de ne pas pouvoir écrire pouvoir considérer l'inspecteur comme un observateur pour les grandes fenêtres, alors qu'il pourrait se contenter d'ignorer les messages qu'il ne connait pas.*

Pour cela on peut choisir un autre protocole, dans les messages sont mutiplexés, i.e. *ils passent tous par le même canal; pour cela, ils sont taggés en entrée, par exemple en utilisant les variantes), et on peut leur associer un comportement par défaut consistant à ne rien faire.*

Reprendre le sujet/observateur selon ce modèle. □

Exercise 3 *Utiliser une variante du motif sujet/observateur pour modéliser l'exemple du "général et du soldat".* □

Slide 16

Protocoles couplés.

Un motif, à la loupe

Slide 17

Dans la plupart des cas, on se retrouve dans une situation analogue au client/serveur, maître/esclave, *etc.*, *i.e.* deux classes doivent s'appeler l'une et l'autre en se passant de l'information, des messages, *etc.* **tout en restant extensibles**...

Dans le modèle maître/esclave, nous avons vu plusieurs solutions :
- Multiplexer les messages sous forme d'un type variante pour les faire passer par le même canal.
- Séparer les messages, en utilisant une paire de noms (émetteur/récepteur) par type de message.

En pratique, on retrouve souvent ce dilemme, et il est important de choisir la bonne combinaison.

Événements clavier et souris

Multiplex

Tous les événements sont séquentialisés et transmis de la même façon. Le récepteur trie et traite les événements de façon appropriée.

Séparation

Slide 18 Le gestionnaire trie les événements (par exemple clavier/souris) selon les différents types (ou combinaison) d'événements et appelle directement des méthodes dédiées.

Événements clavier et souris (comparaison)

– Le multiplex nécessite de figer dès le départ l'ensemble des événements, à moins d'utiliser des types variantes.
Est-ce un problème dans le cas traité ? (cela dépend de l'application considérée)

– Le multiplex traite tous les événements de la même façon, le code est donc bien partagé.

Slide 19

– La séparation paraît plus souple : chaque type d'événement est associé à une méthode dédiée et peut donc être traité de façon appropriée.

– En contrepartie, le nombre de méthodes peut augmenter rapidement, beaucoup étant semblables.

– La multiplication du nombre de méthodes rend souvent le protocole plus confus et l'héritage plus contraignant (tous les composants doivent définir toutes les méthodes).

Événements clavier OU souris

On va souvent choisir une solution mixte, partitionant les événements en différents groupes, traités par des méthodes différentes pour chaque groupe mais par la même méthode au sein d'un même groupe.

Remarque : si besoin, on peut fabriquer des événements composites au niveau du gestionnaire (*e.g.* le double clique).

Slide 20

Les abonnements

Aucune des deux solutions précédentes ne permet de modifier dynamiquement la gestion des événements. Pour changer les événements reçus, il faut le faire au niveau de la classe par héritage.

On peut abstraire le traitement des événements par un système d'abonnement. Le client s'abonne à des services auprès du serveur. L'abonnement peut être effectué et annulé dynamiquement.

Slide 21

En contrepartie, les services fournis doivent être homogènes (opérer sur le même type).

Abonnement exemple

L'abonnement le plus simple est un service dont les messages sont envoyés par effet de bord.

```
let remove c = List.find_all (fun (x,_) -> x <> c)
class ['c] serveur = object
  val mutable clientèle : ('c * ('c -> string -> unit)) list = []
  method enregistre c f = clientèle <- (c, f) :: clientèle
  method désabonne c = clientèle <- remove c clientèle
  method send m = List.iter (function c, f -> f c m) clientèle
end
class ['serveur] client (s : 'a #serveur) = object (self : 'a)
  method enregistre f = s#enregistre self f
  method désabonne = s#désabonne self
  method attend_le_résultat =
    self#enregistre (fun z s -> z#imprime_le_résultat s)
  method imprime_le_résultat s = print_string s;
end;;
```

Slide 22

Abonnement (test)

```
let s = new serveur;;
let c = new client s;;
c # attend_le_résultat;;
s # send "Bonjour, "; s # send "tout le monde!";;
c # désabonne;;
s # send "Toc! toc!"; s # send "Y'a quelqu'un?";;
```

À noter en passant, l'emploi d'un relais : l'utilisateur appelle `self#désabonne` qui en retour appelle `s#désabonne f` de telle façon que l'utilisateur n'ait pas à connaître (ou rappeler) le serveur `s`.

Slide 23

Inconvénient

Les messages sont homogènes, ce qui ne permet pas de gérer plusieurs clientèles.

Messages hétérogènes

Slide 24

On peut homogénéiser les messages de la façon suivante : un abonnement est une paire de type `client * client -> int -> unit` qui expose le type du client et le type du message (`int` ici). Le type du client peut être caché en appliquant la fonction à l'argument de façon retardée :

```
 val mutable clientèle : (string -> unit) list = []
 method enregistre c f =
   clientèle <- (fun z -> f c z) :: clientèle
```

Il est alors possible de gérer des abonnements de clients hétérogènes.
Mieux encore :

```
 val mutable clientèle : (string -> unit) list = []
 method enregistre fc = clientèle <- fc :: clientèle
```

Pour pouvoir se désabonner, on peut maintenir l'objet comme première composante, mais de type `< >`.

Messages hétérogènes

Slide 25

```
type obj = < >
class serveur = object (self)
 val mutable clientèle : (obj * (int -> unit)) list = []
 method désabonne f = clientèle <- remove f clientèle
 method enregistre (f : obj) fc = clientèle <- (f, fc) :: clientèle
 method send m = List.iter (function _, fc -> fc m) clientèle
end

class ['serveur] client (s : #serveur) = object (self)
 constraint 'serveur = #serveur
 method enregistre f = s#enregistre (self :> obj) (f self)
 method désabonne = s#désabonne (self :> obj)
 method attend_le_résultat =
  self#enregistre (fun z s -> z#imprime_le_résultat s; z#désabonne)
 method imprime_le_résultat s = print_int s
end;;
```

Messages hétérogènes

L'utilisation est la même, mais les types sont moins exposés.

```
let s = new serveur in let c = new client s in
c # attend_le_résultat; s # send 1; s # send 2;;
```

1− : unit = ()

On peut maintenant soumettre des abonnements de clients hétérogènes.

Slide 26

Il ne s'agit pas d'un affaiblissement du typage, l'exposition du type de client au serveur n'étant pas nécessaire. Une autre solution aurait été d'utiliser des types existentiels :

Types existentiels

En fait, la paire `(f, c)` a le type $\exists\alpha.(\alpha \times (\alpha \to \texttt{int} \to \texttt{unit}))$ dans lequel la variable α n'est pas libre (elle est donc cachée et on parle de type abstrait ou existentiel).

Rappelle-moi (callbacks)

L'exemple précédent est une forme dite de *callback*.

On donne au serveur une fonction permettant de rappeler le client. Typiquement, un callback est une fonction de type `unit -> unit` mais comportant dans sa fermeture des effets de bords permettant d'informer le client d'événements détectés par le serveur.

Slide 27

Un rappelle-moi est donc un moyen de geler une évaluation dans un objet (A), on dit que l'on forme un glaçon (G), et de passer (G) à un autre objet (B). C'est alors (B) qui a la responsabilité de réveiller (G) et de relancer son exécution.

Rappelle-moi (force)

Slide 28

La force de cette technique réside dans le fait d'encapsuler le calcul dans une fermeture. Cela augmente à la fois deux propriétés souvent antagonistes :

- la sécurité en limitant le risque d'intrusion
- la flexibilité en rendant la plupart des informations non visibles (élimination des contraintes de typage).

Attention La force en fait sa faiblesse...

Les callbacks, en général de type `unit -> unit`, peuvent être accidentellement interchangés. La sûreté (différent de sécurité) est donc diminuée.

Rappelle-moi (faiblesse)

Slide 29

Inversement, la faiblesse de cette technique est que contrairement aux structures de données qui peuvent être analysées et interprétées (*i.e.* les opérations possibles ne sont pas contenues dans les données), les évaluations gelées ne peuvent qu'être réveillées.

Voir le contournement de cette difficulté dans l'exemple des abonnements pour implanter le désabonnement. On ne peut même pas tester l'égalité de deux continuations, sinon leur égalité physique, car ce sont des fonctions.

Rappelle-moi (callbacks)

Slide 30

Le mécanisme de “rappelle-moi” est très utilisé dans un monde concurrent. En effet, lorsque plusieurs processus contribuent ensemble à l’évaluation, le calcul n’est plus un flux linéaire de fonctions qui se passent des arguments et retournent des résultats.

Un processus doit fréquemment geler son exécution, et attendre qu’un autre lui envoie un événement pour continuer. On retrouve un mécanisme un peu analogue à une évaluation gelée passée à quelqu’un d’autre qui doit en assurer le réveil...

Les exécutions gelées sont aussi appelées des *continuations*.

Rappelle-moi (exemple)

Slide 31

On reprend l’exemple du client-serveur dans le contexte d’un gestionnaire d’événements.

Le client indique au serveur qu’il veut recevoir tel événement, et passe simultanément la façon de se faire rappeler. Cet exemple est combiné avec un mécanisme de table, ici délocalisée.

Les mécanismes d’abonnements ont aussi presque toujours besoin d’un rappelle-moi pour retourner les messages.

```
class serveur = object (self)
  val mutable filtres : (char * (unit -> unit)) list = []
  method écoute c f = filtres <- (c, f)::filtres
  method étape = List.assoc (input_char stdin) filtres ()
  method boucle = while true do self#étape done
end;;
```

Rappelle-moi (exemple)

Slide 32

```
class client (s: #serveur) = object
  val mutable position = 0
  method position = print_int position
  method avance = position <- succ position
  method recule = position <- pred position
  method si_faire c f = s#écoute c f
end;;
let s = new serveur in let c = new client s in
c # si_faire 'a' (fun() -> c#avance);
c # si_faire 'r' (fun() -> c#recule);
c # si_faire 'p' (fun() -> c#position);
s # boucle;;
apaaaaaaaappp.
```

1999Uncaught exception: Not_found.

Les relais

Slide 33

Les méthodes "`enregistre`" et "`désabonne`" du client sont des exemples de relais : Elles font suivre les messages adressés au client vers des messages associés du serveur.

```
  method enregistre f = s#enregistre self f
  method désabonne = s#désabonne self
```

Encapsulation

Le relais permettent au client de ne pas exposer certains paramètres (ici le serveur) qui restent encapsulés dans le client.

Indépendemment, un appel distant (en dehors de la classe) peut ne pas connaître le nom du serveur. Le message est alors envoyé au client, qui le complète avec éventuellement des informations seulement connues par le client et le retourne au serveur.

Les relais (liaison tardive)

Sobriété Un relais évite de rappeler certains paramètres à chaque appel. Lorsque ces appels sont dans la classe elle-même, alors le relais joue le rôle d'une liaison locale visible dans plusieurs méthodes (on utilise alors un relais privé).

Slide 34

Modularité Des relais peuvent aussi être utilisés pour leur mécanisme de liaison tardive, permettant de modifier le comportement dans une sous-classe. Par exemple, la méthode `désabonne` peut faire des mises à jour, tracer, ou clore d'autres opérations.

```
class ['serveur] client' s = object (self)
    inherit ['serveur] client s as sup
    method désabonne = print_newline(); sup#désabonne self
  end;;
```

Les relais (collecteur)

Le relais peut aussi être utilisé pour faire automatiquement un travail de collecte de détail, visitant plusieurs objets qui lui fournissent des informations personnelles, avant d'atteindre sa cible finale et de s'exécuter.

Slide 35

Modularité (statique) et reconfiguration (dynamique).

Flexibilité

Slide 36

La modularité fait référence à la possibilité de réutiliser certains composants en les adaptant ou en les spécialisant. Cette extensibilité, a priori statique, revient donc a définir de nouveaux composants à partir d'anciens.

La reconfiguration permet aussi d'adapter des composants en changeant leurs propriétés ou leurs comportements. Cependant à la différence de la modularité, la reconfiguration est dynamique.

Par son caractère dynamique la reconfiguration est plus expressive que la modularité. En contrepartie, elle ne peut se faire que de façon limitée, en particulier par le typage : la reconfiguration ne peut pas changer les types.

Extensibilité

Slide 37

Les exemples d'extensibilités sont bien connus. Elle repose essentiellement sur l'héritage dans les classes et l'application de foncteurs dans les modules.

Par exemple, à partir d'une classe `figure` et d'un comportement de bouton, on fabrique une classe bouton. Ou encore, on personnalise une classe figure en lui ajoutant des menus locaux, *etc.*

Reconfiguration

Slide 38

Dans sa forme la plus simple, la reconfiguration peut ne changer que des paramètres tels que la couleur, l'orientation, les droits *etc.*

Par exemple, on peut ainsi dire ou interdire à un composant de se "retailler automatiquement" ou d'accepter l'ajout ou le retrait de nouveaux éléments. Ces exemples ne font souvent que changer des variables d'états (drapeaux) du composant.

Dans une forme plus expressive, la reconfiguration peut aussi modifier le comportement de certaines méthodes souvent en ajoutant une action au comportement d'origine.

Enfin dans une forme plus poussée, un objet va complètement se restructurer et se comporter d'une toute autre façon sans pour autant changer d'identité.

Reconfiguration

En pratique, le passage d'une forme à l'autre est souvent clair selon le contexte, mais en théorique cette progression est continue. En particulier, les méthodes de reconfiguration s'appuient presqu'exclusivement sur les effets de bords donc la modification de champs mutables.

Slide 39

Grossièrement on pourrait classer les différents types de reconfiguration en fonction du contenu des variables mutables : données, fonctions, ou objets.

La reconfiguration revient donc à simuler dynamiquement l'effet de l'héritage statique. Cette simulation est souvent approchée et limitée. Elle se fait en général par une indirection, souvent par effet de bord, mais parfois de façon fonctionnelle.

(Nous laissons de coté la reconfiguration simple qui s'implémente évidemment par un champ mutable.)

Méthodes mutables

Slide 40

Les méthodes mutables n'existent pas, mais cet effet peut être obtenu simplement en faisant exécuter par la méthode le contenu d'une variable mutable.

```
class ['a] action =
  object (self : 'z)
    val mutable action : 'z -> 'a =
      fun s -> raise (Failure "No action")
    method action = action self
    method set_action f = action <- f
  end;;
```

En pratique, la classe a évidemment d'autres méthodes statiques et seulement quelques méthodes particulières que l'on veut rendre mutables et pour lesquellles on définit un triplet formé d'une variable d'instance mutable, d'une méthode d'invocation et d'une méthode d'affectation.

Héritage des méthodes mutables

On peut aussi simuler l'héritage, à condition de fixer un protocole d'appel (ici un seul argument)

Slide 41

```
class ['a] super_action = object
  inherit ['a] action
  method post_action f =
    let super = action in
    action <- (fun self z -> f self (super self z))
  method pre_action f =
    let super = action in
    action <- (fun self z -> super self (f self z))
end;;
```

Généralisation et spécialisation On peut facilement remplacer l'action par une liste d'actions à composer (ou à itérer dans le cas où elles ne retournent rien).

Les crochets (hooks)

Slide 42

Les crochets (hooks) permettent de prévoir des emplacements avec des comportements par défaut qui pourront être changés ultérieurement. C'est une variation sur les méthodes mutables.

Emacs est l'exemple typique d'utilisation de cette technique : la plupart des fonctions peuvent être personnalisées en accrochant des fonctions de personnalisation aux modes fournis pas défaut.

Les crochets sont en fait un concept plus général qui ne s'applique pas qu'à l'orienté objets. Les méthodes mutables sont un moyen, parmi d'autres, de l'implémenter, y compris en présence d'objets.

Comme pour les méthodes mutables, il faut prévoir un emplacement modifiable, pour pouvoir augmenter le comportement ultérieurement, souvent initialisé avec du code vide.

Les tables

Les tables sont une généralisation des méthodes mutables. Cela revient à multiplexer plusieurs méthodes. Pour cela, on passe un paramètre particulier utiliser comme clé dans une table de méthodes (peu importe la représentation de la table).

Typiquement, le traitement des événements peut se faire par une table d'association utilisant le type de l'événement comme clé.

Slide 43

Par rapport à l'approche statique, le typage est à la fois contraint (tous les messages ont le même type) et relâché (il y a un message par défaut pour les clés non définies, en général *"ne rien faire"*, afin d'éviter l'erreur *"message non compris"*).

Cette approche permet de modifier la réaction à certains événements. L'ajout et le retrait reviennent à une modification du comportement par défaut.

Exercise 4 *On peut imaginer le de pousser à fond le mécanisme de reconfiguration et traiter tous les messages comme des messages dynamiques.*

1. *Quel serait le problème principal rencontré ?*
2. *Décrire deux façons de le contourner.*
3. *Illustrer l'une des deux sur un exemple simple tels que les points colorés.*

□

Slide 44

La délégation

Slide 45

Cela revient à déléguer certaines méthodes aux méthodes d'un objet d'une autre classe. La création d'un objet (a) d'une classe (A) provoque la création d'un objet (b) d'une autre classe (B).

L'état de l'objet (a) comporte indirectement l'état de l'objet (b). On peut alors adresser un message à (b) en donnant accès directement à (b) au travers de (a) ou en relayant certaines méthodes de (b) dans (a).

La délégation est un style de programmation avec objet utilisé indépendemment de la reconfiguration, à la place de l'héritage. Il facilite certaines reconfigurations.

La délégation (exemple simplifié)

Slide 46

Contexte

```
class texte s0 =
  object
    val mutable s = s0 method s = s
    method imprime = print_string s
  end
class point x0 =
  object
    val mutable x = x0 method x = x
    method imprime = Printf.printf "<%d>" x
  end;;
```

But

Écrire une classe `point_annoté` à partir des classes précédentes.

La délégation (exemple simplifié)

Slide 47

Version avec héritage

```
class point_annoté x0 s0 =
  object
    inherit point x0 as point
    inherit texte s0 as texte
    method imprime = point#imprime; texte#imprime
  end;;
```

Version avec délégation :

```
class point_annoté x0 s0 =
  object
    val point = new point x0 method point = point
    val texte = new texte s0 method texte = texte
    method imprime = point#imprime; texte#imprime
  end;;
```

Délégation et héritage

Slide 48

Héritage

- Les deux classes parentes sont aplaties en une seule qui conserve la liaison tardive et permet de spécialiser toutes les méthodes.
- L'accès privilégié aux variables d'instance `x` et `s` est normal.

Délégation

- Les méthodes peuvent avoir le même nom dans les deux classes parentes. (on peut former une classe de bipoints avec deux points).
- On peut maintenant facilement reconfigurer un objet de la classe `point_annoté`, en changeant par exemple l'objet `texte` par un objet `texte_éditable` (il suffit de rendre mutables les champs `point` et `texte`.

Délégation et héritage

Les points forts d'une approche sont les points faibles de l'autre.

En particulier, dans la délégation, la classe héritée n'a pas accès aux variables des classes parentes ; cela amène souvent à rendre visibles des champs ou des méthodes qui devraient logiquement rester privées (visibles seulement dans une sous-classe).

Slide 49

Inversement, il n'est pas possible d'avoir deux méthodes différentes portant le même nom dans la version avec héritage.

La version avec héritage est donc plus modulaire (bien que le dernier point soit une limitation). La délégation pallie à l'absence d'un mécanisme de vue.

Les vues

Les vues sont des interfaces (dictionnaires) entre les noms de méthodes et leurs implémentation. Un objet peut alors avoir plusieurs implémentation de la même méthode mais dans des vues différentes. Les vues conservent le mécanisme de la liaison tardive (l'espace des noms est hiérarchisée, mais les implémentations sont aplaties).

Slide 50

Malheureusement... les langages à objets existant n'offrent pas encore de mécanismes de vues (suffisamment sophistiqués).

Objets sans états...

Voir combinaison des modules et des objets.

C'est une utilisation atypique mais expressive des objets pour bénéficier de l'héritage et du mécanisme de la liaison tardive que n'offrent pas les structures.

C'est aussi d'une certaine façon un palliatif à l'absence de mixins.

Slide 51

1 Solutions des exercices

Exercise 1, page 14

```
class ['sujet] grande_fenêtre =
  object (self)
    inherit ['sujet] fenêtre
    val mutable taille = 1
    method retaille x =
      taille <- taille + x;
      self#envoie (fun obs -> obs#retaillé)
    val mutable devant = false
    method en_avant =
      devant <- true;
      self#envoie (fun x -> x#devant)
    method dessine =
      Printf.printf "{Position = %d; Taille = %d}" position taille;
  end;;
```

Exercise 1 (continued)

```
class ['sujet] grand_gestionnaire =
  object
    inherit ['sujet] gestionnaire
    method retaillé (s : 'sujet) : unit = s#en_avant
    method devant (s : 'sujet) : unit = s#dessine
  end;;

new grande_fenêtre#ajoute (new grand_gestionnaire);;
let w = new grande_fenêtre in
  w#ajoute (new grand_gestionnaire); w#déplace 1; w#retaille 2;;
```

```
{Position = 1; Taille = 1}{Position = 1; Taille = 3}- : unit = ()
```

Exercise 1 (continued)

```
class ['sujet] big_brother =
  object
    inherit ['sujet] inspecteur
    method retaillé (s:'sujet) = print_string "<R>"
    method devant (s:'sujet) = print_string "<D>"
  end;;
```

Et voilà le tout ensemble :

```
let w = new grande_fenêtre in
   w#ajoute (new grand_gestionnaire); w#ajoute (new big_brother);
   w#retaille 2; w#déplace 1;;
```

```
<R><D>{Position = 0; Taille = 3}<D>{Position = 1; Taille = 3}- : unit = ()
```