Modularité, Objets et Types.

Didier Rémy
2001 - 2002

http://cristal.inria.fr/~remy/mot/0/
http://www.enseignement.polytechnique.fr/profs/informatique/Didier.Remy/mot/0/

Calendrier des cours

Tous les mercredi

Mois	Octobre					Novembre				Décembre	
EA	3	10	17	24	31	7	14	21	28	5	17–19
Projet							14	21	28	5	

Slide 1

- Cours EA de 13h45 à 15h15 en PC 38
- Travaux dirigés EA de 15h30 à 17h30 en salle info 32.
- Projets EA de 10h15 à 12h15 en salle info 32 ?

Programmation traditionnelle

Slide 2

Les programmes sont écrits de façon linéaire.

On peut seulement ajouter de nouvelles définitions à la fin du programme.

Réutilisation du code par “couper-coller”.

Pas de partage de code
- duplication du code, et des erreurs !
- difficulté de maintenance.

La modularité

Slide 3

Un problème essentiel qui doit être pris au sérieux et considérer dans son ensemble.

Un problème difficile tant en théorie et en pratique.

Des solutions divergentes

Malgré de fortes ressemblances, des motivations presqu'identiques et l'utilisation de presque les mêmes ingrédients, les approches divergent rapidement. Classes ou modules ? Et dans l'approche à classes, les méthodes sont elles portées par les objets ou en dehors comme des fonctions surchargées ?

Cours et projet
- Le cours présente et explique les différents mécanismes disponibles.
- Le projet permet leur mise en œuvre en vraie grandeur.

Quelques mots clés

Abstraction Compositionalité

Extension

Slide 4 Réutilisation Partage

Composants Interface Cacher

Sûreté Typage

Une courte introduction à la modularité

Cette introduction est volontairement générale et peu technique. Elle présente intuitivement, donc très informellement, les différents mécanismes.

- La modularité en général et dans les langages de programmation.
- Sécurité, abstraction, typage.
- La modularité du pauvre
- Les systèmes de modules
- Langages à objets et à classes

Slide 5

La modularité, un concept général

Découpage de l'ensemble en composants indépendants

Rendre les gros projets réalisables

Donner de la structure à l'ensemble

Rendre les gros projets compréhensibles

Spécifier les liens entre les composantes

Slide 6

Rendre les gros projets maintenables

Identifier des sous-composantes indépendantes

Rendre les projets réutilisables

Enfin, en forçant l'abstraction,

La modularité augmente la sûreté.

En informatique remplacer projet par programme.

La modularité dans l'industrie

Les Légos, le Mécano : on construit des objets à partir de briques élémentaires universelles. On peut réutiliser des objets pour produire des assemblages et des mécanismes encore plus complexes.

Dans la mécanique : visserie, roulements à billes ont une interface (nomenclature) très précise pour être interchangeables, réutilisables. Fabrication si besoin d'outils spécialisés ($\approx$ librairies)

Slide 7

Circuits électriques : on forme des opérateurs logiques à partir de transistors, puis des modules (aditioneur, multiplicateur) que l'on assemble ensuite pour formée des unités arithmétiques, etc. Cette composition hiérarchique des circuits électroniques est essentielle.

La modularité en mathématiques

Slide 8

Généraliser pour réutiliser

Résolution d'une équation du second degré : factorisation “intuitive” ou traitement du cas général et application au(x) cas particulier(s).

Un théorème ou une théorie sont souvent étudiés dans le cadre le plus général et on déduit des corollaires pour des cas particuliers.

Généraliser pour simplifier

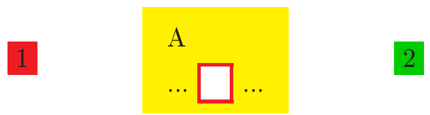
– Disparition des détails (bruit), rendant l'essentiel apparent.
– Meilleure appréhension de l'ensemble (vérification, évolution).

Mais il ne faut pas généraliser... pour la beauté du résultat !

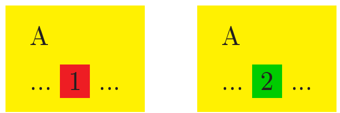
L'abstraction en vaut-elle la peine ? (nombre d'applications, simplification conceptuelle).

Abstraction

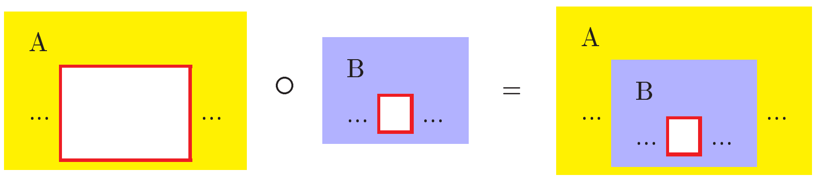
Slide 9



Plusieurs instances partagent le même corps (preuve, code) :



Les abstractions peuvent être composées :



Par exemple à partir d'une algèbre sur un anneau et de l'anneau des polynômes, on peut former l'algèbre des polynômes.

Abstraction *v.s.* Extension

L'abstraction se fait *a priori* (il faut deviner à l'avance les besoins de partage)

Sait-on toujours ce qu'il faut abstraire ?

On ne peut pas tout abstraire...

Slide 10

L'extension se fait *a posteriori*

Par exemple, pour spécialiser une preuve, il faut peut-être ajouter des cas particuliers, ou dédoubler certains cas trop généraux, mais la plupart restant inchangés.

Dans quelles conditions peut-on spécialiser un résultat général, sans refaire une preuve complète ?

Extension et raffinement

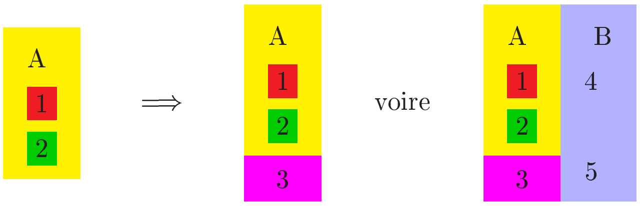
À partir d'un composant, en créer de nouveaux par :

– extension (verticale), *i.e.* ajout de fonctionnalités ;

– modification (verticale) ; ou

– adjonction (horizontale), *i.e.* raffinement de la structure de donnée.

Slide 11



Questions

(a) Est-ce que 1 et 2 peuvent être récursivement définis ?

(b) Est-ce que 3 peut remplacer ou modifier 2 ?

(c) En combinant (a) et (b) est-ce que 1 dépend de 3 ?

Difficultés de l'extension

Un couper/coller n'est pas une extension (pas de partage — par exemple, il faut refaire toute la preuve).

En mathématiques, on peut laisser l'utilisateur reconstruire des morceaux de preuve, mais on ne peut pas laisser le compilateur réécrire des morceaux de programme.

Slide 12 Difficulté supplémentaire dans le cas d'une preuve par récurrence (bon fondement).

Dans quelle mesure peut-on encore traiter les composants comme des boîtes noires (les compiler séparément) ?

Organisation modulaire

Quelque soit le domaine (industrie, mathématiques, informatique), on est amené à :

- Construire des composants élémentaires,
- Combiner ces composants.
- Utiliser une structure pyramidale : les composants sont les éléments de composants plus complexes.

Slide 13 En général, certaines parties peuvent être réutilisées inchangées, mais d'autres doivent être adaptées ou remplacées.

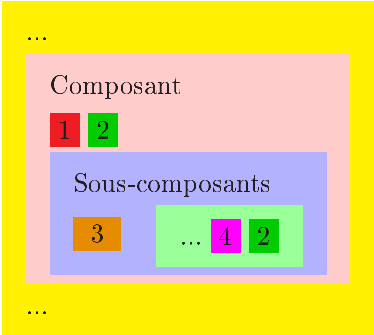
Décomposer le tout en parties peu indépendantes permet de réutiliser celles-ci telles quelles.

Une preuve qui utilise des lemmes auxiliaires difficiles peut elle-même être simple : la preuve peut alors facilement être refaite si les lemmes eux restent inchangés. Au pire, on ne retouchera que certains lemmes.

Organisation modulaire

Slide 14

Avec partage de sous-composantes, en profondeur (importance de la structure pyramidale) :



Voir par exemple
http ://www-sor.inria.fr/~java/doc/javasoft/jdk/docs/api/tree.html
l'arbre des classes du langage Java ou le graphe de dépendance d'un système de module (ou d'une chaîne de fabrication d'airbus...).

La programmation modulaire

Slide 15

La complexité d'un programme :

- 100 lignes : script, un seul fichier, mais utilisation intensive des librairies.
- 1000 lignes : petite application, quelques fichiers. La modularité facilite l'écriture et la maintenance.
- 10000 lignes : application sérieuse. La modularité permet de mieux organiser le code et le développement en parallèle de plusieurs parties, la réutilisation de certaines composantes...
- 100000 lignes et plus : système complexe, probablement développé par plusieurs personnes devant évolué dans le temps, nécessitant une architecture robuste bien conçue et probablement le développement de librairies spécialisées.

Comparer à la complexité d'une usine pétro-chimique...

Un enjeu : la sûreté

La programmation modulaire a besoin de plus de sûreté

Un composant peut être utilisé en dehors du contexte dans lequel il a été écrit, par d'autres personnes, etc. Il faut donc une rigueur accrue : spécification claire, robustesse, résistance aux cas pathologiques, (*exemple d'Ariane 5*).

Slide 16

La modularité améliore la sûreté (et la sécurité)

Les composants peuvent être développés et testés séparément.

L'utilisation de librairies générales ou spécialisées tend à réduire la taille du noyau, les parties non spécifiques à l'application deviennent des opérations fournies en librairies.

Cacher les détails (un composant est une boîte noire avec une interface) évite de casser accidentellement des invariants (les variables internes sont inaccessibles).

Par un jeu de cache-cache

Cacher les détails de l'implémentation – augmente la sécurité – augmente la réutilisabilité.	Le faire *le plus tôt possible !*

Deux approches orthogonales :
– cacher les valeurs
– cacher les types

Slide 17

Cacher les valeurs (gestion de l'espace des noms)

Une valeur interne devient invisible et inaccessible de l'extérieur.

Très facile à mettre en œuvre, mais limité : il faut parfois communiquer des informations à d'autres modules (par exemple pour les analyser/trier) tout en souhaitant que l'autre module traite (une partie) de ces informations comme des boîtes noires.

Par un jeu de cache-cache (suite)

Slide 18

Cacher les types (rendre les types abstraits)

Une valeur ne peut être analysée (détruite) ou reconstruite qu'en connaissant sa structure (son type).

Par exemple, le facteur distribue le courrier, mais le courrier est cacheté et le facteur ne peut pas l'ouvrir.

Méthode plus puissante, mais légèrement plus difficile à mettre en œuvre : types (partiellement) abstraits.

Interface des composants

Slide 19

Décrire ce qui est utilisé (importé) et fourni (exporté) par un composant.

Exemple des prises électriques :
- Prise 20A *v.s.* 5A.
- Prise mâle (entrée) *v.s.* femelle (sortie).
- Une prise mâle avec terre ne peut pas être branchée dans une prise femelle sans terre.
- L'inverse est possible (sous-typage au niveau des interfaces)

Le typage est utilisé pour forcer le respect des interfaces (vérification statique et syntaxique).

Les branchements sont effectués par le compilateur.

La modularité du pauvre

Slide 20

La modularité est d'abord un style de programmation

Les langages avancés offrent des mécanismes pour favoriser la programmation modulaire, mais ne la force pas.

Inversement, des langages comme C qui offrent peu de mécanisme spécifiques, permettent quand même à l'utilisateur d'écrire des gros systèmes, mais avec beaucoup de discipline.

Se rappeler les premiers tuyaux du tronc commun, par exemple...
- Donner des noms aux constantes numériques.
- Partager le code en utilisant des fonctions auxiliaires. Par exemple, remplacer `C[A] ; ... ; C[B]` par

  ```
  c(x) { C[x] } c(A); ...; c(B)
  ```

  en C, ou par `let c(x) = C[x] in c(A) ... c(B)` en ML.

Utilisation de types abstraits

Slide 21

Il est important de cacher la représentation des données de telles façon que celle-ci puisse être changée sans affecter le reste du programme.

Pour cela on fournit des constructeurs (fonctions de création) et des destructeurs (fonctions d'accès) pour écrire et lire de telles données.

Le reste du programme doit être indépendant de la représentation choisie. Si le langage le permet, la représentation sera cachée par le typage dans le reste du programme (vérification automatique), sinon l'utilisateur doit être suffisamment discipliné pour respecter cette convention lui-même.

Exemple de type abstrait en C

```
typedef struct personne { char* nom; int age; } personne;
personne naissance (char* n, int a)
  { personne p; p.nom = n; p.age = a; return p; }
int age (personne p) { return p.age; }
char* nom (personne p) { return p.nom; }
```

Le reste du programme n'utilise que les fonctions `naissance`, `nom`, ... `age`.

Slide 22

La représentation peut alors être modifiée ou enrichie de façon transparente, par exemple en ajoutant un champ `sexe` :

```
typedef strut personne
  { char* nom, long date_naissance, short sexe } personne
int age (personne p) { return date() - r.date_naissance; }
```

Les outils du pauvre

Paramétrisation

En général, le seul outil pour l'abstraction est la fonction.

Le regroupement des paramètres dont dépend une application dans une structure (enregistrement) permet d'écrire le corps du programme comme une fonction de cette structure qui peut être appliqué avec des conditions initiales différentes.

Slide 23

Extensibilité

Traditionnellement (en C, Pascal, Caml, etc.), un programme est organisé autour des données :

1. Définition du type de donnée.
2. Écriture de fonctions pour manipuler les données.

L'ajout de nouvelles opérations sur les données est facile, mais le raffinement des données est difficile sans réécrire l'ensemble.

Exemple d'extension (verticale)

Slide 24

Un programme de la forme suivante

```
(* Définition des données *)
type forêt = Pinède | Chêneraie
(* Opérations sur les données *)
let dessiner_une_forêt = ...
let déboiser = Chêneraie -> couper | Pinède -> brûler
```

peut être étendu avec de nouvelles opérations

```
let reboiser = ...
```

sans changer le code précédent (l'extension peut même souvent se faire dans un fichier séparé).

Par contre ajouter un nouveau type de forêt, `Vierge` par exemple, oblige à réécrire toutes les fonctions.

Le problème de la pauvreté

Slide 25

Les solutions ci-dessus permettent de faire au mieux en l'absence de constructions modulaires spécialisés, mais elles sont limitées :

- Les fonctions permettent de paramétrer par rapport aux valeurs, mais pas par rapport aux types.
- L'application de fonction est une opération dynamique de bas niveau, alors que l'assemblage de composants pourrait se faire pendant le chargement (link). Ce sont des opérations plus macroscopiques.
- L'utilisation de type abstraits sans support dans le langage permet de programmer de façon modulaire, sans bénéficier de la sécurité offerte par un mécanisme de typage.

Il est important d'avoir des constructions (primitives) pour favoriser la modularité.

Deux outils complémentaires

Les systèmes de modules

La notion de module est prise au sérieux.
- langage d'assemblage de structures basé sur l'abstraction.
- n'aide pas à l'extensibilité, n'aime pas trop la récursion...

Les langages à objets (et à classes)

Slide 26

La notion d'extensibilité est prise au sérieux.
- horizontalement par construction (ajout de classes), verticalement par l'héritage.
- l'abstraction de valeur est obtenu par masquage de certaines composantes.
- c'est un style de programmation très rigide, qui a ses terrains de prédilection, mais aussi ses limites. En particulier, la paramétrisation est plus difficile.

Les systèmes de modules

Système stratifié (couche au dessus du langage de base).

Une *structure* regroupe un ensemble de phrases du langage de base (déclaration de types, de valeurs, etc.)

Une *signature* décrit les composantes (types, valeurs, autres structures) publiques implémentées par une structure.

Slide 27

Une structure peut être paramétrée/instantiée par une autre structure (foncteur).

Un langage d'assemblage de structures, simple et puissant, avec une sémantique claire, mais pas (encore) :
- de structures récursives.
- d'extension, donc pas de réutilisation a posteriori.

L’approche à objets et à classes

Slide 28

Approche mixte (objets et classes)

Modularité en ¡0 (forme de peigne)3.5,0.5) schéma très fréquent dans la programmation modulaire :

$$\begin{array}{ccccccc} C_1 & \Longrightarrow & C_2 & \Longrightarrow & C_3 & \Longrightarrow & C_4 \\ | & & | & & | & & | \\ O_1 & & O_2 & & O_3 & & O_4 \end{array}$$

Les classes sont extensibles mais ne sont pas des valeurs ordinaires —ce sont des modèles d’objets.

Les objets sont des valeurs, dérivées des classes (par instantiation), mais ne sont plus extensibles.

La sémantique est compliquée : l’extension d’une classe peut redéfinir d’anciennes méthodes, ce qui change le graphe d’appels récursifs (mécanisme dit de la *liaison tardive*)

Sous-typage

Slide 29

Modules et objets ont tous les deux une notion de sous-typage correspondant à la possibilité d’oublier des composantes (*c.f.* prises électriques).

On peut restreindre la signature d’une structure en cachant des composantes, ou le type d’un objet en ignorant certaines méthodes (en général le type d’un objet d’une sous-classe est un sous-type du type des objets de la classe parente).

La notion de sous-typage est une approximation de la réalité.

Ses faiblesses sont compensées par des coercions *i.e.* par du typage *dynamique*) : par analogie électrique, un fil porte une couleur qui indique la phase, le neutre... Si un fil est décoloré, il faut tester pour retrouver lequel est la phase, et un échec est possible (ce qui est encore préférable à un court-circuit ou une erreur fatale).

Sous-typage : trouvez l'erreur !

Slide 30

```
class Point {
  private int x = 0;
  Point (int x0) { x = x0; }
  int getx () { return x; }
  Point max (Point p)
    { if (p.getx() > getx()) return p;
      else return this; }
}
class Bipoint extends Point {
  private int y = 0;
  Bipoint (int x0, int y0) { super (x0); y = y0; }
  int gety () { return y; }
  Bipoint max (Bipoint p)
    { if (p.gety() > gety()) return p;
      else return super.max (p); }
}
```

Les ingrédients d'un exemple complexe

Slide 31

Une armée modèle est composée d'une classe "soldat" et d'une classe "général". La communication est bi-directionnelle.

– Le soldat rend compte au général en s'identifiant (nom ou matricule) et en faisant un compte-rendu.
– Le général ordonne ou punit (le soldat).

(Une instance plus informatique de ce protocole de communication se retrouve dans un window-manager : les fenêtres signalent des événements ; le manager ainsi informé a une vue globale de la disposition des fenêtres et de la souris et décide des actions à effectuer par exemple demander le rafraîchissement de certaines fenêtres.)

Mots Clés

Exemple de Motif. Ici, chaque **classe** est **paramétrée** par rapport aux **objets** de l'autre classe pour rester **extensible**.

Les enjeux

Slide 32

Cet exemple illustre une configuration complexe dans laquelle des objets de plusieurs classes interagissent récursivement. Ces classes peuvent être regroupées dans une structure.

À partir de cette configuration modèle, il faut pouvoir :
- en prendre des instances.
- enrichir le modèle en raffinant le comportement de chaque classe, éventuellement en ajoutant des possibilités de communication entre les objets de chaque classe.
- enrichir le modèle en ajoutant de nouvelles classes interagissant avec les classes précédentes.

Bien sûr, nous exigeons une contrainte forte de sécurité où le typage seul doit garantir que tous les messages sont bien reçus (ce qui exclut l'utilisation du typage dynamique).

Extension du modèle

Slide 33

La configuration précédente est une *armée modèle*. Pour en faire une *armée de combat* on augmente la force de chaque intervenant tout en préservant le protocole de communication. Par exemple, on donne :
- deux étoiles au général pour augmenter son d'autorité.
- un fusil au soldat pour augmenter son d'agressivité.

Le général peut maintenant donner l'ordre de tirer au soldat.

Éventuellement, on étendra la configuration en introduisant de nouvelles classes, par exemple si un politique interagit avec le général, tout en préservant la correction du protocole précédent.

Mots clés Chaque classe **hérite** de la classe correspondante. Il est nécessaire de bien traiter les **méthodes binaires** pour préserver le **typage**.

Généralisation, reproduction en série

Slide 34

D'autres extensions sont naturellement possibles par exemple un soldat peut rapporter à deux généraux, même si elles mènent vite à l'anarchie...

Un pays peut aussi essayer d'exporter son modèle militaire (après avoir essayé plusieurs protocoles de communication).

Il est alors facile d'abstraire le modèle par rapport aux caractérisques propres aux soldats de chaque pays (sexe, âge, courage,...) tout en préservant certains secrets.

Mots clés

Un **foncteur** englobant réalise l'**abstraction**. L'utilisation d'une **signature** abstraite permet de bien préserver les secrets... et d'empêcher les soldats d'obéir au général d'une autre armée.