Programmation avancée

Modularité, Spécifications algébriques

ENSIIE

Semestre 2 — 2011–12

Modularité

GCC : ∼4 millions de lignes de code
Noyau Linux : ∼12 millions de lignes de code

- besoin de partage des tâches entre développeurs

On sépare les différentes composantes d'un logiciel dans des modules

Chaque module possède une interface qui indique quelles sont les fonctionnalités qu'il propose

En dehors du module, seul ce qui est déclaré dans l'interface peut être utilisé (boîte noire)

Modules

Permet :

- séparation des tâches
- création d'un espace de nom
- réutilisabilité (par exemple, bibliothèques)
- encapsulation
- abstraction (en particulier des types)
- maintenabilité

Séparation des tâches

- Chaque module peut être implémenté par un développeur différent
- Le seul point sur lequel ils doivent s'entendre est l'interface des modules
  (et la sémantique des fonctions...)

Création d'un espace de nom

Dans certains langages, chaque module crée un nouvel espace de nom :

- On peut donner le même nom de fonction dans des modules différents
  ex. en OCaml : `List.length` `Array.lenght` `String.length`
- Pour accèder à un fonction, on utilise son nom complet
  ex. en OCaml : `Nom_du_module.nom_de_la_fonction`
- En général, on peut ouvrir les modules pour accèder aux fonctions qu'il contient sans avoir à rappeler le nom du module

Ex. en OCaml :

```
open String
open Array

let comp l a = List.length l < lenght a
(* val comp : 'a list -> 'b array -> bool = <fun> *)
```


Aparté : surcharge VS polymorphisme

- surcharge : même nom de fonction pour deux algorithmes différents

  Exemple : + en C sur les `int` et les `double`

- polymorphisme : fonction générique qui peut s'appliquer à n'importe quel type de données

  Exemple : `fun x -> x` en OCaml

Réutilisabilité

Chaque module est indépendant, il peut donc être réutilisé dans un autre projet sans avoir à tout reprendre.
Ex. : Structure de donnée pour stocker des chaînes de caractères
Permet de ne pas réinventer la roue

On peut regrouper des modules pour créer des bibliothèques
En particulier, on dispose en général pour chaque langage d'une bibliothèque standard

- pour C, définie dans le standard C ANSI, implémentée par exemple par la GNU C Library
- en OCaml, bibliothèque standard fournie avec le langage (`List`, `Array`, `String`, `Map`, `Set`, `...`)

Encapsulation

- Seules les fonctions déclarées dans l'interface permettent d'accéder aux objets
- Le développeur n'a pas à se soucier d'objets mal formés créés à l'extérieur du module
- Permet de maintenir des invariants (via des constructeurs intelligents par exemple)

Abstraction

Les structures de données utilisées pour implémenter tel ou tel objet peuvent rester abstraites

- définition de type abstrait dans l'interface

  En C :
  `typedef struct contenu_type* type_abstrait`
  dans le `.h`
  la structure `struct contenu_type` n'est pas définie dans l'interface, elle n'est donc pas visible de l'extérieur

  En OCaml : `type type_abstrait` dans le `.mli`

Pas besoin de connaître l'implémentation concrète depuis l'extérieur

Maintenabilité

Permet de pouvoir changer l'implémentation concrète sans avoir à changer tout le code

Exemple : utilisation de tableaux au lieu de listes pour implémenter des piles

Exemple : changement du tri par bulle (complexité $O(n^2)$) en tri fusion ($O(n \log n)$)

Invisible depuis les autres modules

Extensions

Suivant les langages de programmation, modules plus avancés :

- modules imbriqués
- modules paramétrés par des modules (foncteurs)
  - généricité
- modules de première classe : peuvent être manipulés comme des valeurs du langage

Dictionnaire

Types abstraits

Au moment de la conception du programme, on est amené à définir des types abstraits en spécifiant quelles propriétés ils doivent vérifier

- Le développeur du type devra respecter ces propriétés, mais sera libre de l'implémentation concrète
- L'utilisateur pourra supposer que les propriétés sont vérifiées pour son code

Spécifications algébriques

Les spécifications algébriques sont une des manières de procéder ainsi. Elles contiennent

- les prototypes des fonctions manipulant le type abstrait
- des équations mettant en relation ces fonctions, expliquant ainsi les propriétés qu'elles doivent vérifier

De la spécification algébrique à l'implémentation

prototypes $\longrightarrow$ interface d'un module

comment vérifier que l'implémentation vérifie les propriétés ?

- test
  - à la main
  - génération de tests

  exhaustivité ?
- certification (preuve)
  - à la main
  - formelle (interactive ou automatique)

  $\pm$ facile suivant le langage

Dictionnaire

Exemple concret :
les dictionnaires (aussi appelés tableaux associatifs ou tables d'association)

On veut associer des clefs $k \in \mathcal{K}ey$ à des valeurs $v \in \mathcal{V}al$

Il est possible

- de créer un dictionnaire vide
- d'insérer une nouvelle association
- de rechercher à quelle valeur est associée une clef
- de supprimer une association

En général, $\mathcal{K}ey$ est supposé totalement ordonné.
Dans la suite, on supposera qu'à une clef n'est associée qu'une seule valeur au maximum (le dictionnaire est une fonction au sens mathématique)

Exemples d'utilisation

Omniprésent en informatique :

- Table des symboles dans un compilateur
  $\mathcal{K}ey=$ symboles, $\mathcal{V}al=$ informations (type, visibilité, ...)
- Système de fichier
  $\mathcal{K}ey=$ chemins, $\mathcal{V}al=$ emplacements disque
- Mémoïsation
  $\mathcal{K}ey=$ arguments, $\mathcal{V}al=$ résultats
- Moteur de recherche
  $\mathcal{K}ey=$ mots-clefs, $\mathcal{V}al=$ pages associées
- Représentation d'un ensemble
  $\mathcal{K}ey=$ élément, $\mathcal{V}al=\{est\_dedans\}$
- ...

Spécification algébrique d'un dictionnaire

```
type ('k,'v) dict

creer : int -> dict
inserer : dict -> 'k -> 'v -> dict
rechercher : dict -> 'k -> ('v | ERR)
supprimer : dict -> 'k -> (dict | ERR)

rechercher(creer(i), k) = ERR
rechercher(inserer(d,k,v),k) = v
rechercher(inserer(d,k,v),k') = rechercher(d,k')  k≠k'
rechercher(supprimer(d,k),k) = ERR
rechercher(supprimer(d,k),k') = rechercher(d,k')  k≠k'
```

Interface : C

```
typedef int key;
typedef char* value;

typedef struct dict_base *dict;

dict creer(int);

value rechercher(dict, key);

dict inserer(dict, key, value);

dict supprimer(dict, key);
```


Interface : OCaml

```
type ('key,'value) dict

val creer : int -> ('key,'value) dict

val rechercher :
  ('key,'value) dict -> 'key -> 'value

val inserer :
  ('key,'value) dict -> 'key -> 'value
    -> ('key,'value) dict

val supprimer :
  ('key,'value) dict -> 'key -> ('key,'value) dict
```


Implémentations

Dans ce cours, trois implémentations

- listes d'association
- arbres bien équilibrés
- table de hachage

Comparaison de la complexité en temps de chacune des fonctions

- en moyenne
- dans le pire des cas