PG104 - Programmation Fonctionnelle

D. Renault

Enseirb-Matmeca

11 mars 2025, v.1.2.5

`https://www.labri.fr/perso/renault/working/teaching/functional/functional.php`

Modularité

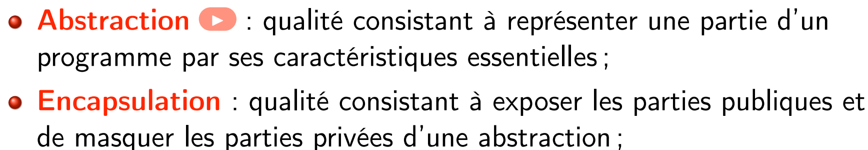
Idée générale

Considérer le code non plus au niveau de la **fonction**, mais des **ensembles de fonctions**, voire des ensembles de fonctionnalités.

- Le terme utilisé pour différencier les deux est celui de **granularité**.
- Transforme les problématiques de programmation en des problématiques d'architecture.

Abtraction, Encapsulation

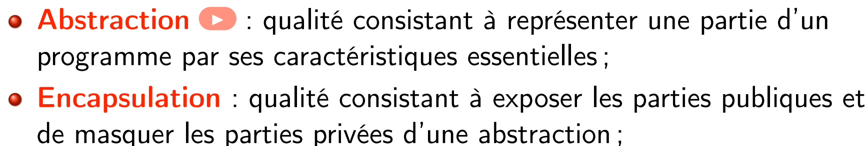
Les propriétés entraperçues jusqu'à présent peuvent être précisées et considérées sous un nouvel éclairage :



- **Abstraction** : qualité consistant à représenter une partie d'un programme par ses caractéristiques essentielles ;
- **Encapsulation** : qualité consistant à exposer les parties publiques et de masquer les parties privées d'une abstraction ;

Abtraction, Encapsulation

Les propriétés entraperçues jusqu'à présent peuvent être précisées et considérées sous un nouvel éclairage :



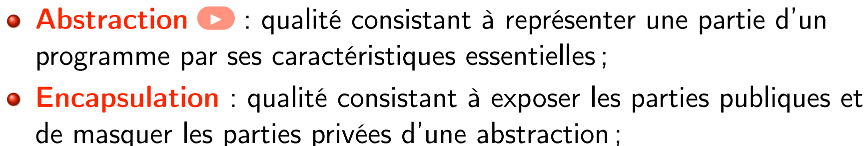
- **Abstraction** ▶ : qualité consistant à représenter une partie d'un programme par ses caractéristiques essentielles ;
- **Encapsulation** : qualité consistant à exposer les parties publiques et de masquer les parties privées d'une abstraction ;

(découpage naïf avec fermeture)

```
function make_rng() {
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };                                  Private
  return rand;                        Public
}
let rand = make_rng();
```

Abtraction, Encapsulation

Les propriétés entraperçues jusqu'à présent peuvent être précisées et considérées sous un nouvel éclairage :



- **Abstraction** : qualité consistant à représenter une partie d'un programme par ses caractéristiques essentielles ;
- **Encapsulation** : qualité consistant à exposer les parties publiques et de masquer les parties privées d'une abstraction ;

(découpage naïf avec fermeture)

```
function make_rng() {
  let seed = 12345;
  const m = 65537; const a = 75;
  function rand() {
    seed = (a * seed) % m;
    return seed;
  };                                    Private
  return rand;                          Public
}
let rand = make_rng();
```

⇒

(cf. Stanford JS Crypto Library)

```
class Aes;
function aes_generate() { /*...*/ }
class EllipticCurve;
function ecc_generate() { /*...*/ }
function sha256() { /*...*/ }
function sha512() { /*...*/ }           Private

function selectCypher() { /*...*/ }
function encrypt() { /*...*/ }
function decrypt() { /*...*/ }          Public
```

Spécification

Quelles peuvent être ces caractéristiques essentielles d'un programme ?
Concrètement, il faut parler de **spécification**. Il en existe plusieurs sortes :

- Spécification **fonctionnelle** :
  - "Il doit être possible d'ajouter ou d'ôter un élément de l'ensemble."
  - "Il doit être possible de tester si un élément appartient ou pas à l'ensemble."
- Spécification **non fonctionnelle** :
  - "L'implémentation doit utiliser des algorithmes de complexité linéaire."
  - "L'implémentation doit pouvoir gérer des ensembles de taille arbitraire."

Spécification

Quelles peuvent être ces caractéristiques essentielles d'un programme ?
Concrètement, il faut parler de **spécification**. Il en existe plusieurs sortes :

- Spécification d'**interface** ou de **type** :

```
;; T is the type of the
;; elements inside the set.
newtype set[T];
```

```
set_empty     : set[T]
set_is_empty  : set[T] → boolean
set_add       : set[T]*T → set[T]
set_remove    : set[T]*T → set[T]
set_find      : set[T]*T → boolean
```

- Spécification **formelle** :

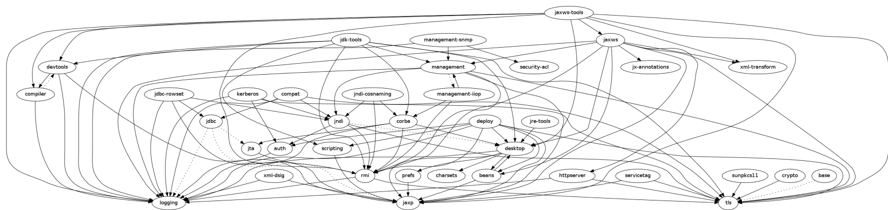
```
set_is_empty(set_empty)        = true
set_is_empty(set_add(s,i))     = false

set_find(set_empty, i)         = false
set_find(set_add(s, i), i)     = true
set_find(set_add(s, i), i')    = set_find(s, i')
```

Modularité

Définition (Modularité)

Propriété logicielle qualifiant le fait de décomposer un programme en un ensemble de composants de manière à ce que :

- **Cohésion** : chaque composant constitue une unité de code cohérente ;
- **Couplage** : les dépendances entre composants sont faibles.



Graphe de dépendances de l'OpenJDK
Source : Project Jigsaw

Module

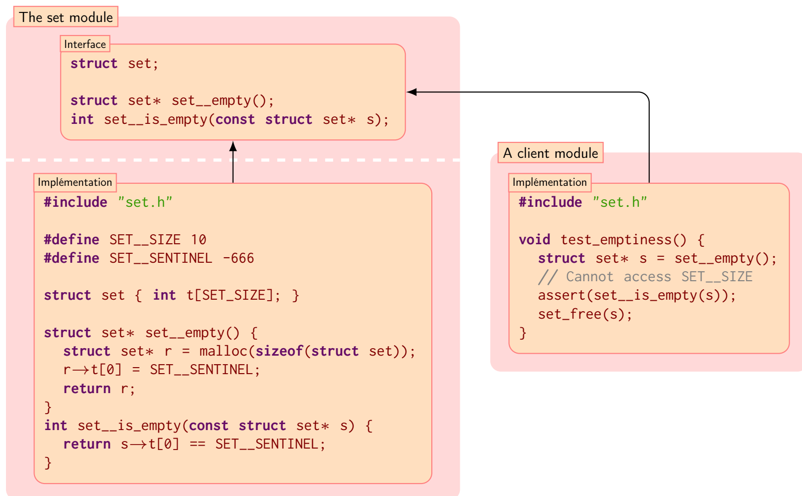
Définition (Module)

Un **module** est une construction du langage représentant une unité de code (types, valeurs, fonctions, et toute forme d'expression autorisée par le langage) et satisfaisant :

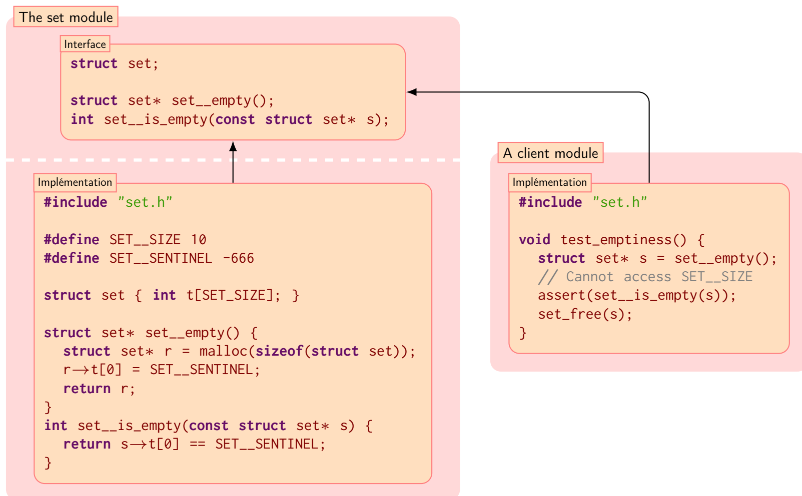
- **Interface** : un module peut fournir ou requérir un ensemble de composants de manière publique ;
- **Encapsulation** : un module peut masquer ou rendre abstrait une partie de ses composants ;

- Des ensembles de modules peuvent se connecter selon le graphe des dépendances induites par leurs interfaces ;
- Un module ne devrait **dépendre que des interfaces** de ses dépendances (et non pas de leur contenu concret).

Exemple : des modules en C



Exemple : des modules en C



Interface ✓ Encapsulation ✓ Indépendence ✓

Exemple : modules et TADs

Un exemple classique de module est le suivant :

Définition (Type abstrait de données)

Un **type abstrait de données** (*abstract data type*) consiste en :

- un ensemble de valeurs ;
- une interface / représentation abstraite de cet ensemble de valeurs (e.g sous la forme d'une liste des opérations qu'on peut leur appliquer) ;
- une ou plusieurs implémentations de cette interface.

L'abstraction permet ici :

- de proposer plusieurs implémentations **différentes** et dans l'idéal **substituables** du type de données ;
- d'autoriser différents clients à manipuler le type de données en restant **indépendant** de la représentation concrète.

Modules et modules ...

Selon les langages de programmation, les modules prennent des formes variées et satisfont des propriétés différentes :

- Les interfaces peuvent permettre différents niveaux de **vérification** : uniquement les noms (Racket, Python), ou les types (C, OCaml)
- L'**encapsulation** peut ne pas exister (cf. Python module privates) ;
- Les interfaces sont parfois attachées au module (Racket, Haskell) ou sont des éléments **indépendants** (C, OCaml).

Pour la suite, nous allons examiner les possibilités modulaires du langage Ecmascript.

Ecmascript et modules

Les modules Ecmascript ont une histoire désordonnée :

- Originellement, la norme ne contenait pas de spécification pour les couches modulaires.
- A partir de 2009, plusieurs entités ont proposé des systèmes de modules différents : **CommonJS**, AMD, RequireJS . . .
- La norme Ecmascript 6 a proposé un système de module en 2015, parfois nommé **Ecmascript** modules, incompatible avec les précédents.
- Node.js utilise par défaut les modules CommonJS, en offrant un support "expérimental" des modules Ecmascript (depuis 2015 !).

Développement “sans modules”

Principe

Le code est écrit comme un seul bloc, avec de faibles spécifications.

```
function cons(hd, tl)      { return { car: hd, cdr: tl }; }
function set__empty()      { return undefined; }
function set__is_empty(s)  { return s === undefined; }
function set__add(x, s)    { return cons(x, s); }
function set__find(x, s)   { return (s === undefined) ? false :
                                 (x === s.car || set__find(x, s.cdr)); }
set__is_empty(set__empty());              // → true
set__find(1, set__add(1, set__empty())); // → true
```

Se marie bien avec :

- REPL (simplifie le développement incrémental)
- Système de type dynamique (retarde la vérification)

Langages adaptés : Lisp, Scheme, Javascript, Racket, Python, Ruby

Limites du “sans modules”

Une telle pratique induit un développement **monolithique** :

- Des parties du code avec des **finalités différentes** sont mélangées dans un unique fichier (e.g. implémentation et tests) ;
- La **réutilisation** et la **modification** d’une partie du code est gênée (e.g. modification de la représentation d’un ensemble) ;
- La **vérification** du code est plus complexe (tout ou rien) ;

Une telle méthode est adaptée au **prototypage**, mais montre ses limites lors du passage à l’échelle : travail en équipe compliqué, pas de compilation séparée, pas de tests séparés . . .

Principe de la programmation modulaire

Décomposer un code monolithique en une famille de modules cohésifs et faiblement couplés.

```
function is_set(s) { /* ... */ }
function set_add(s, e) { /* ... */ }
function set_empty() { /* ... */ }
function set_find(s, e) { /* ... */ }
function test_add() { /* ... */ }
function test_empty() { /* ... */ }
function test_find() { /* ... */ }
function set_recipes() { /* ... */ }
function find_recipe(key) { /* ... */ }
function cook_recipe(key) { /* ... */ }
```

Programmation modulaire

Principe de la programmation modulaire

Décomposer un code monolithique en une famille de modules cohésifs et faiblement couplés.

```
function is_set(s) { /* ... */ }
function set_add(s, e) { /* ... */ }
function set_empty() { /* ... */ }
function set_find(s, e) { /* ... */ }
function test_add() { /* ... */ }
function test_empty() { /* ... */ }
function test_find() { /* ... */ }
function set_recipes() { /* ... */ }
function find_recipe(key) { /* ... */ }
function cook_recipe(key) { /* ... */ }
```

Implémentation (is_set, set_add, set_empty, set_find)

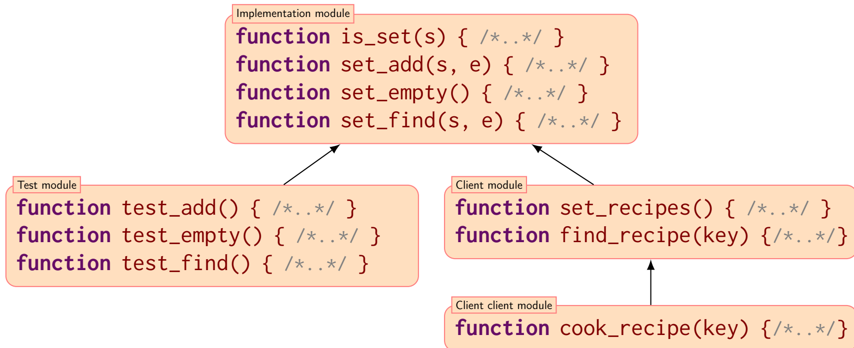
Tests (test_add, test_empty, test_find)

Code client (set_recipes, find_recipe, cook_recipe)

Programmation modulaire

Principe de la programmation modulaire

Décomposer un code monolithique en des ensembles de modules cohésifs et faiblement couplés.



Modules CommonJS (non utilisés dans ce cours)

Les modules CommonJS sont connectés par **require** et **exports** :

set-impl.js

```
function set_empty() { /*..*/ }
function is_empty(s) { /*..*/ }

exports.set_empty = set_empty;        // ‘exports‘ is a dict containing
exports.is_empty  = is_empty;         // the exported identifiers
```

set-test.js

```
const Set = require("./set-impl.js");   // ‘require‘ returns the
                                         // aforementioned dict

function test_empty() { return Set.is_empty(Set.set_empty()); }
console.log(`Empty set is empty : ${test_empty()}`);
```

set-client.js

```
const Set = require("./set-impl.js");

let set_recipes = Set.set_empty();
function add_recipe(r) { set_recipes = Set.set_add(r, set_recipes); }
```

Modules Ecmascript

Les modules Ecmascript sont connectés par **import** et **export** :

set-impl.mjs

```
function set_empty() { /*..*/ }
function is_empty(s) { /*..*/ }

export { set_empty, is_empty,          // 'export' provides a list of
         set_add, set_find };          // exported identifiers
```

set-test.mjs

```
import * as Set from "./set_impl.mjs";  // 'import' supplies the
                                        // requested identifiers

function test_empty() { return Set.is_empty(Set.set_empty()); }
console.log(`Empty set is empty : ${test_empty()}`);
```

set-client.mjs

```
import * as Set from "./set_impl.mjs";

let set_recipes = Set.set_empty();
function add_recipe(r) { set_recipes = Set.set_add(r, set_recipes); }
```

Premières conclusions

Avantages :

- Facile la **réutilisation de code** : à la fois les tests et les clients peuvent réutiliser le même module d'implémentation ;
- Autorise des **implémentations multiples** : le client peut choisir une implémentation adéquate sans modifier son propre code ;
- Permet une **vérification** de la présence des fonctions importées et exportées.

Limitations en Ecmascript :

- Vérification réduite à la présence/absence des noms demandés ;
- Notion d'interface (d'un module) presque inexistante.

Développement de code modulaire

Recette pour construire une architecture modulaire :

1. Identifier les objets/**types de valeurs**/composants manipulés ;
   Ex. : "les ensembles d'entiers positifs"
   ⇒ leur associer un module.
2. Identifier les **opérations** sur chacun de ces types de valeurs ;
   Ex. : "`setAdd` : ajouter un élément, `setRemove` : retirer un élément"
   ⇒ contruire une interface associée.
3. Construire des **représentations concrètes** ;
   Ex. : "sentinel, dynamic, link"
   ⇒ implémenter ladite interface.

Principe général de développement modulaire

Lorsque l'on conçoit une architecture logicielle, il vaut **toujours mieux** dépendre d'une interface que d'une représentation concrète.

Même si le développement modulaire peut s'arrêter ici, il est envisageable de demander plus :

- En termes de **vérification** :
  Quitte à préciser des spécifications logicielles, existe-t'il des tactiques pour assurer que des modules vérifient une spécification ?
- En termes d'**architecture** :
  Quitte à décomposer le code en morceaux, peut-on faire évoluer ces morceaux, les augmenter, ou les remplacer facilement ?

Contrats

Définition (Contrat)

Un **contrat** est un ensemble de prédicats qui fixent les comportements des fonctions au sein d'un module (précondition, postcondition, invariants)

Comme pour les assertions, les contrats sont vérifiés **dynamiquement**.

Exemple en Racket

Documentation de la fonction `list-set` :

```
;; Updates element at index 'pos' of 'lst'
(list-set lst pos val) → list?
  lst : list?
  pos : (and/c (>=/c 0) (</c (length lst)))
  val : any/c
```

"`pos` est un indice valide dans `lst`"

Utilisation incorrecte :

```
(list-set '(1 2 3) 100 4)
```

```
<=: contract violation
  expected: (and (>= 0) (< 3))
  given: 100
```

Langages adaptés : Eiffel, Racket, C# with Code Contracts

Signatures

Définition (Signature)

Une **signature** est un ensemble de définitions de types pour les identifiants exportés par un module.

Comme les types, les signatures sont vérifiées **statiquement**.

Exemple en C

Documentation de la fonction `set__find` :

```c
// Returns non-zero if c belongs to se, 0 otherwise
int set__find(const struct set *se, int c);
```

Utilisation incorrecte :

```c
struct set* se = set__empty();
set__find(1, se);
```

```
error: passing argument 1 of 'set__find' makes pointer from integer without a cast
expected 'const struct set *' but argument is of type 'int'
```

Langages adaptés : la plupart des langages à typage statique

Exemples de signatures

Idée

Permettre de **séparer** la signature de l'implémentation.

- **Interfaces** en Java

```java
interface Set<T>
{
    Set<T>  set_add(Set<T> set, T el);
    Set<T>  set_remove(Set<T> set, T el);
    Set<T>  set_empty();
    boolean set_is_empty(Set<T> set);
    boolean set_find(Set<T> set, T el);
    int     set_length(Set<T> set);
}
```

- **Signatures de modules** en OCaml

```ocaml
module type SET = sig
  type 'a set
  val set_add      : 'a set -> 'a -> 'a set
  val set_remove   : 'a set -> 'a -> 'a set
  val set_empty    : unit   -> 'a set
  val set_is_empty : 'a set -> bool
  val set_find     : 'a set -> 'a -> bool
  val set_length   : 'a set -> int
end
```

- Les signatures, comme les contrats, deviennent des éléments de l'architecture, comme les implémentations.
- L'abstraction consiste alors à dépendre uniquement des interfaces.

Conclusion sur la programmation modulaire

- Composer du code de manière modulaire est la bonne manière de travailler sur des architectures de tailles importantes.
- Les qualités d'une bonne décomposition modulaire (cohésion, faible couplage) visent à **contrôler les dépendances** entre les composants.
- La notion d'**abstraction** est au coeur du contrôle de ces dépendances.
- Une bonne décomposition permet d'envisager :
  - des propriétés de **génie logiciel** : réutilisabilité, substitutivité, extensibilité, maintenabilité ...
  - des facilités de **vérifications** des dépendances : spécifications, signatures, contrats, types ...

Conclusion générale

Il n'est pas naïf de considérer en dernier lieu de ce cours une discussion sur la notion de modularité.

- La notion de **dépendance** est un principe incontournable de programmation.
  Mieux, elle est appliquable à la fois au niveau fonctionnel (cf. pureté) et au niveau modulaire (cf. la notion de granularité) ;
- La notion de **composition** est un autre principe de programmation incontournable.
  Elle est aussi appliquable à la fois au niveau fonctionnel (cf. 1ère classe) et au niveau modulaire.

Quelques lectures . . .

Fogus, M.: Functional Javascript.
O'Reilly, 2013.

Narbel, P.: Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml.
Vuibert, 2005.

Reade, C.: Elements of Functional Programming.
Addison-Wesley, 1989.

Bird, R. et P. Wadler: Introduction to Functional Programming.
Prentice Hall, 1988.

Okasaki, C.: Purely functional data structures.
Cambridge University Press, 1999.

Narbel, P.: Techniques Avancées de Programmation.
Cours de Master 2 à l'Université de Bordeaux.