

POO

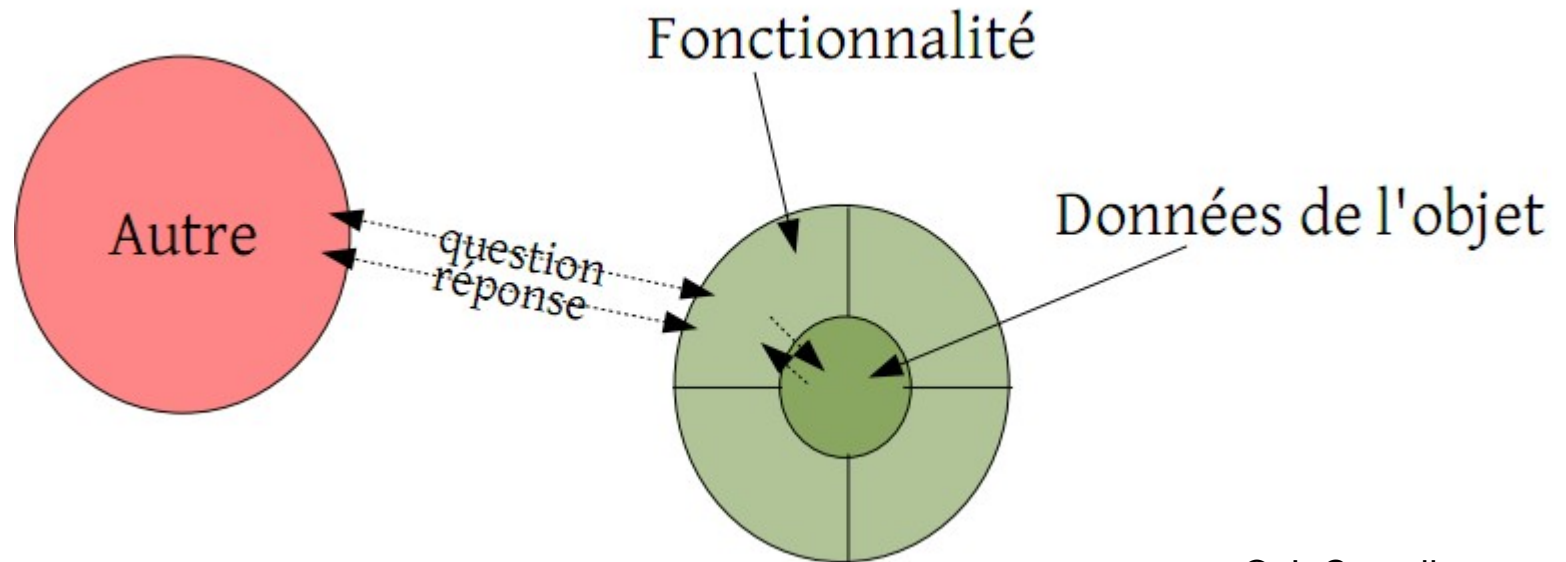
- Objets
- Hiérarchie de classes, lien entre classes
- Modules

Le but principal de la programmation objet
est d'aider à la conception
ET
la maintenance de logiciels

POO : Objet

■ Concept Objet :

- + Un module d'un programme doit souvent interagir avec d'autres, éventuellement sans « scrupules »
- + Un objet protège son intégrité des autres objets,
 - il n'expose que des fonctionnalités



© J. Cervelle

POO : Objet

- champs = état
- fonctionnalités = comportement = méthodes
 - Paradigme : envoi de message
- → **Encapsulation**
 - Cohérence après construction, cohérence
 - Attention aux setter !
 - Attention aux getter et aux membres mutables

POO : encapsulation

- Encapsulation : la base
 - + Séparer intérieur / extérieur
 - + Responsabiliser
 - + Contractualiser
 - + → On ne se préoccupe pas de l'implémentation de ce qu'on utilise
- Encapsulation = je ne sais pas *COMMENT* tu fais
 - + Je ne peux pas coder une logique qui dépend de comment ça marche !
 - + Cercle vertueux de solidité

POO : encapsulation

- Conséquences vertueuses
 - + Découplage "interface" / implémentation
 - + Un objet → une responsabilité
 - + Il est interdit de connaître l'état interne d'un objet
 - ➔ les méthodes garantissent que l'objet est dans un état valide
 - ➔ Jamais le "monde" extérieur !

POO : Classe

- Réutilisation
 - + Héritage
 - + Polymorphisme

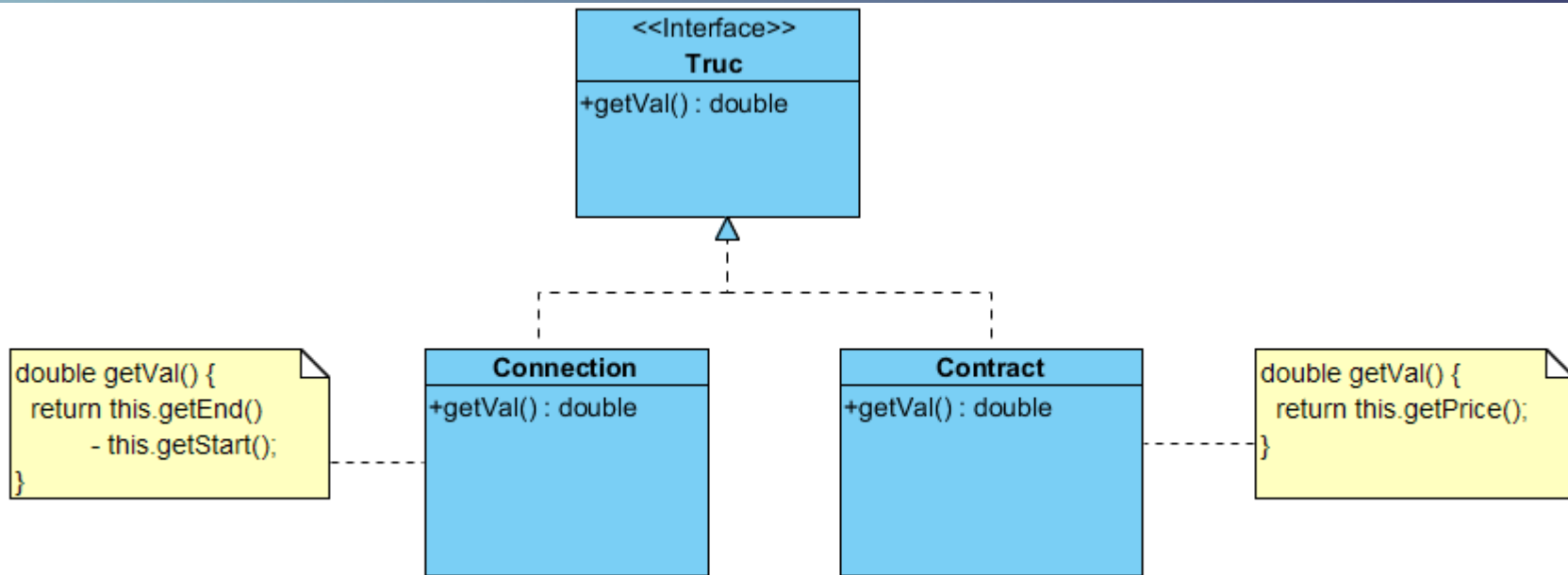
Bonnes pratiques que vous connaissez déjà ?

- Une responsabilité → un objet
- I/A/C sans abus

POO : Héritage vs Délégation

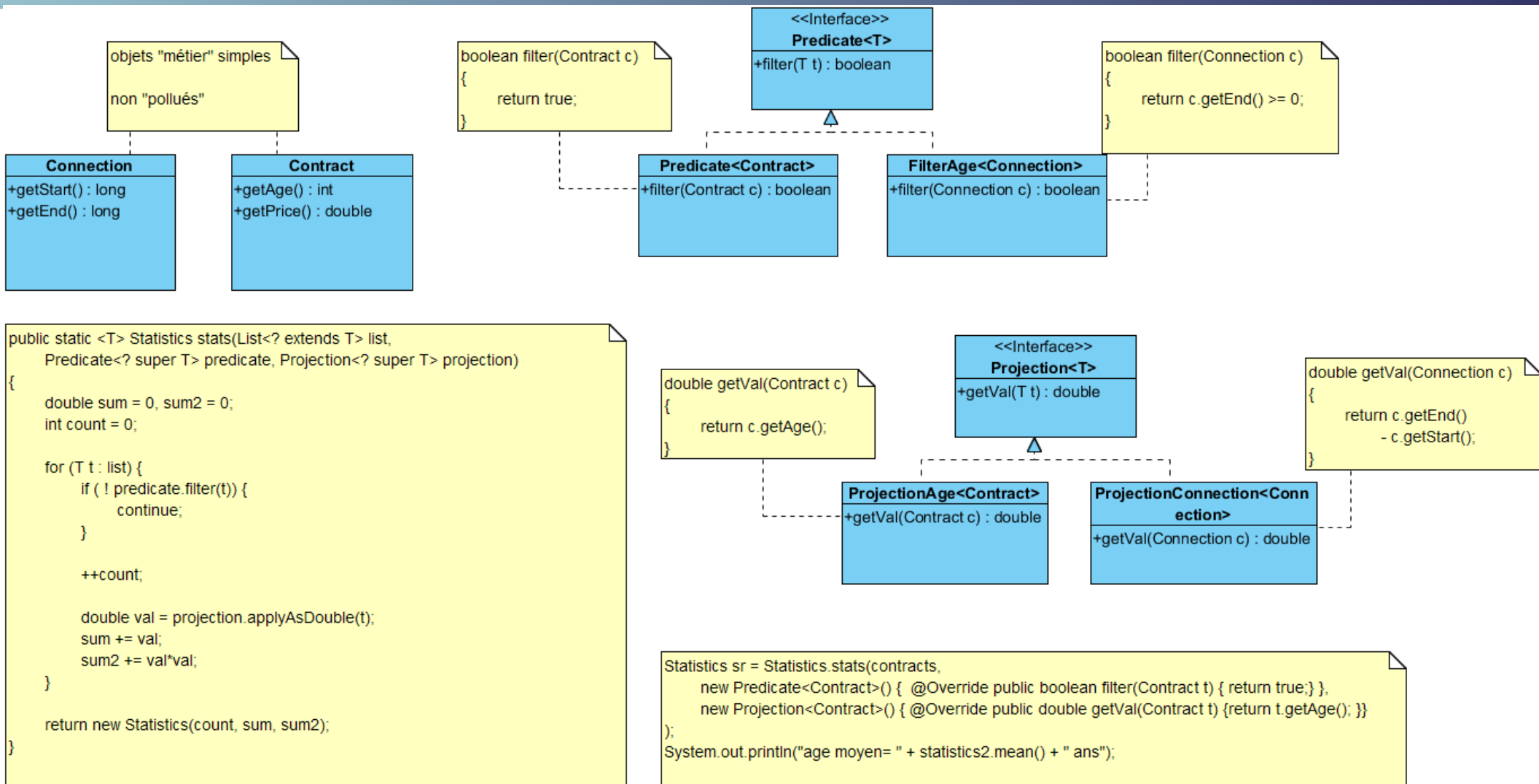
- Réutilisation
 - + "sur-vendue" depuis 30 ans !
- Héritage multiple
 - + Limite heureusement incontournable avec java (7)
- + Héritage = EST UN
- + Délégation = A UN

TD1 : avec héritage...



```
... stats(List<Truc> list)
{
    double sum = 0, sum2 = 0;
    int count = 0;
    for (Truc t : list) {
        {
            double val = t.getVal();
            sum += val;
            sum2 += val*val;
            count ++;
        }
    }
    double mean = sum / count;
    double mean2 = sum2 / count;
    double stdDev = Math.sqrt(mean2 - mean*mean);
    ...
}
```

TD1 : avec délégation...



POO : principe de base

- Préférer la composition à l'héritage
 - + L'héritage a été mis en avant pour la réutilisation
 - Trop !
 - + Souvent, l'héritage est rigide et la composition est souple
 - + Flexibilité
 - liens entre objets vs liens entre classes/interfaces
 - dynamique vs static

Souvent, cela implique d'avoir "plus" d'objets.
Comment les initialiser ?
Et comment initialiser les liens entre eux ?

Principe de substitution de Liskov (LSP)

- « Si une propriété P est vraie pour une instance x d'un type T, alors cette propriété P doit rester vraie pour tout instance y d'un sous-type de T »
- Implications :
 - Le contrat défini par la classe de base (pour chacune de ses méthodes) doit être respecté par les classe dérivées
 - L'appelant n'a pas à connaître le type exact de la classe qu'il manipule : n'importe quelle classe dérivée peut être *substituée* à la classe qu'il utilise
- → Principe de base du polymorphisme :
 - Si on substitue une classe par une autre dérivée de la même hiérarchie: comportement (bien sûr) différent mais *conforme*.

POO : polymorphisme

- « Program to interface, not implementation »
 - + Je ne sais pas *QUI* tu es
 - + Cercle vertueux « d'ignorance » / de généricité
 - + Je ne peux pas coder une logique qui dépend « qui fait quoi » !
 - ➔ anti-spaghetti

POO : principes de base

- Séparer ce qui varie de ce qui ne varie pas !
 - + Changement plus localisés
 - + Moins de risques
 - + Dépendances limitées
 - + Moins de risques d'ouverture
- Plus de flexibilité
- Moins de risques

POO : principes de base un bon début : S.O.L.I.D.

- Un bon point de départ, l'essentiel
 - + Single responsibility principle
 - + Open close principle
 - + Liskov principle
 - + Interface segregation principle
 - + Dependency inversion principle

POO : principes de base

Single responsibility principle

- Une classe = une et une seule responsabilité
- Objectifs :
 - + Limiter rigidité et fragilité
 - + Aider à la localité
 - + Les CRC peuvent aider

POO : principes de base

Open Close Principle

■ OCP

- + Fermeture : Le code a été écrit, testé, debuggé → on n'y touche plus
- + Ouverture : Le code est prévu pour être extensible

POO : principes de base

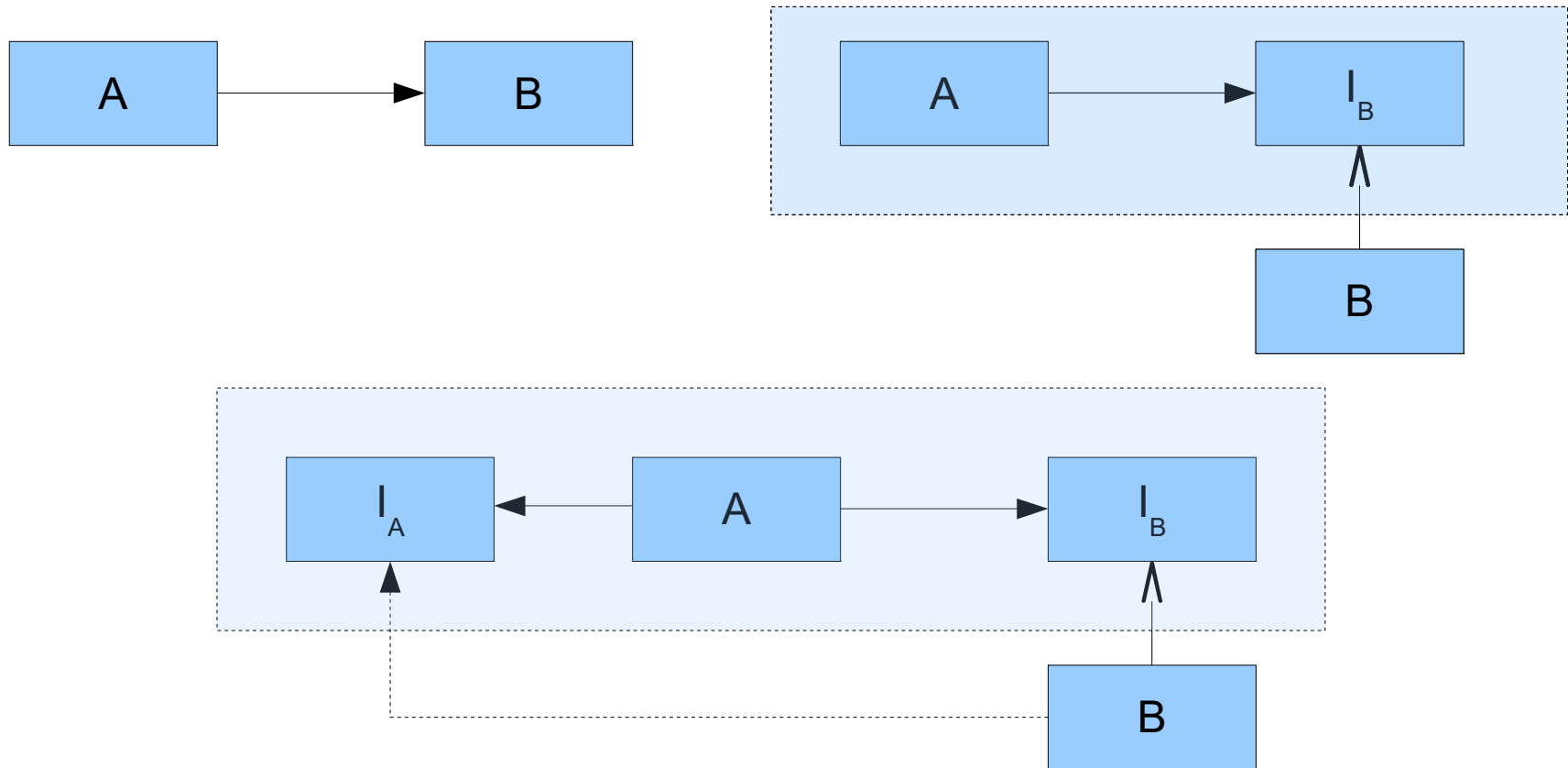
Interface segregation principle

- Interface segregation principle
 - + Un client doit avoir des interfaces avec uniquement ce dont il a besoin
 - + Incite à ne pas faire "extract interface" sans réfléchir
 - + Incite à avoir des interfaces petites
- + Peut amener à une multiplication excessive du nombre d'interfaces
 - à l'extrême : une interface avec une méthode
 - Nécessaire : expérience, pragmatisme et le bon sens !

POO : principes de base

Dependency inversion principle

- Inversion de dépendance : comment ?



POO : principes de base

- Inversion de dépendance
 1. Réduire les dépendances sur les classes concrètes
 2. Program to interface, not implementation »
 3. Les abstractions ne doivent pas dépendre de détails.
→ Les détails doivent dépendre d'abstractions.
 4. Ne dépendre QUE sur des abstractions, y compris pour les classes de bas niveau
- Extensibilité
- Permet OCP

POO : principes de base

- Des règles apparemment strictes ...
 - + Des interfaces cohérentes et pas trop grosses
 - Sinon, on découpe
 - + Des objets qui interagissent avec peu d'objets
 - Sinon, il manque des « chefs » ou des intermédiaires
 - + Ne pas parler directement aux amis de ses amis !
 - On fait transmettre les messages

Mais à appliquer avec pragmatisme !

Modules

- + Définitions
- + Principe d'ouverture/fermeture
- + Interfaces
- + Fermeture commune
- + Stabilité et abstraction
- + Dépendances acycliques
- + Livraison
- + Ré-ouvrir ?

Modules

- **Définition** : Un module est une unité de code
- De toute taille
- À forte cohésion
 - + classe, fichier
 - + paquetage
 - + bibliothèque

Module client

- Quand deux modules interagissent, celui qui **utilise l'autre** est appelé module client
 - ✦ Ex Voiture/Conducteur
- Le conducteur utilise l'interface de la voiture et pas l'inverse
- Ex Administrateur/Administré
- Les deux modules font des demandes croisées. Les deux modules sont clients l'un de l'autre.

Module Ouvert

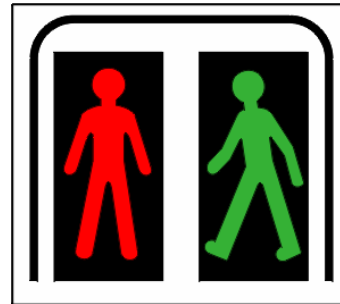
- Un module dans lequel il est encore possible de réaliser des modifications
- Ajout d'attributs, changement de code, ajout de classe publique...
- C'est ce que demande le concepteur d'un module qui veut pouvoir : le corriger, l'étendre, l'adapter

Module Fermé

- Compilable, validé, testé (+ versionné)
- **Un module utilisable par d'autres** (ce qu'ils veulent)
- La valeur d'un module est plus grande s'il est fermé
 - + avancement
 - + debugging
 - + réutilisation

Un rêve : Ouvert ET Fermé

- On aimerait à avoir des paquetages ouverts et fermés
 - + sécurisé
 - + extensible



Principe d'ouverture/fermeture

- On cherche à avoir des modules ouverts/fermés
- Cette stratégie s'applique quelle que soit le type de module (classe, paquetage, .c)

Modules → Paquetages

- Unité de programmation plus efficace que les classes pour l'organisation du travail et pour définir l'architecture logicielle
 - + Conception
 - + Cohésion
 - + Documentation
 - + Livraison
- Les paquetages permettent :
 - + Ouverture/fermeture réaliste
 - + Bonne lisibilité
 - + ensemble des classes publiques/protégées et leur doc
 - + Réutilisabilité



De l'art d'écrire des Modules

- Le travail d'étude se différencie du travail de production
 - + il s'applique à quelque chose de Neuf.
 - + Ainsi le risque d'erreur y est beaucoup plus prégnant.
Toutes les méthodes cherchent à réduire le coût de la correction d'erreur avec une stratégie qui consiste à faire cette détection au plus tôt.
- L'art d'écrire les modules a pour objectif de réduire le nombre d'erreurs.

Modules

Encapsulation - Interface

- Cacher le fonctionnement interne d'un module
- On cherche à obtenir des interfaces, au sens de la chimie
 - + Qui sépare l'intérieur du module de l'extérieur, sans possibilité d'échanges
 - + Les changements internes au module n'ont pas d'influence sur le code extérieur
 - + ni les corrections, ni les bugs ne doivent se propager d'un module à l'autre
 - + En particulier, les changements d'implémentation d'un module ne doivent pas avoir d'incidence sur les modules qui l'utilisent
- Concepts importants : contrat et responsabilité

Interface

- L'interface est la partie publique d'un module
- Un module client utilise l'interface
- Si l'interface change, les modules clients doivent changer

Ce doit être la partie stable d'un module

Interface

- **Une bonne partie du travail de conception est la définition des interfaces entre les différents modules**
- Il est important de bien définir l'interface
 - + Syntaxique : prototype
 - + Sémantique :
 - + fonctionnel
 - + contrat complet (prérequis, cas d'erreur, effet de bord, ...)

Interfaces

- Le choix d'une interface est une activité primordiale qui doit être identifiée et maîtrisée en tant que telle
- C'est un savoir faire spécifique
- La qualité des interfaces est un préalable à la qualité du logiciel
- Objectifs pour une interface
 - + réduire les risques d'ouverture
 - + assurer la compréhensibilité
 - + assurer la réutilisabilité
 - + continuité

Petites interfaces

- Les interfaces pléthoriques vont à l'encontre de la compréhension du module
- Une interface importante n'aide pas à la réutilisabilité du module
- Si l'interface est grande, le module et la maintenance sont complexes

Peu d'interfaces

- Il faut réduire le nombre d'interfaces utilisées par un module :
 - + un module doit être le client d'un minimum d'autres
- + de fournisseurs => + de chances d'avoir à changer
- + de fournisseurs => + de complexité

Interfaces explicites

- Les modules doivent avoir une interface explicite où rien n'est caché ou tordu
- La documentation du module devrait pouvoir se limiter à l'interface

Fermeture commune

- Les entités qui dépendent d'un même concept doivent être placées dans le même module
- Si le concept change, les classes concernées sont toutes dans le même paquetage
- **L'unité de fermeture est le module**
- Changer un concept force la réouverture, il est important qu'un nombre minimal de modules soit rouverts

Stabilité

- La stabilité est l'inverse du risque de changement
- Plus le module est client d'autre modules, moins il est stable
- Mesure mathématique : hauteur dans le graphe de dépendance

Valeur de la stabilité

- Plus un module est fermé donc stable, plus il a de valeur pour les développeurs
- Plus les modules stables (qui ont donc beaucoup de clients directs et indirects) sont effectivement stable, plus l'application est stable dans son ensemble

Stabilité et abstraction

- Plus un module est abstrait, plus il est facile de le garder stable
- Pour fabriquer un paquetage stable, il faut identifier les bonnes abstractions
 - + ne rien sous-entendre sur l'implémentation
 - + *plus c'est abstrait, plus c'est stable*

Stabilité

- Identifier et concevoir les paquetages qui doivent être stables est une tâche importante
- Penser que l'on peut agrandir son paquetage avec de nouvelles interfaces sans trop de danger
- *Un package ne doit dépendre que de packages plus stables que lui.*
 - + *Sinon refactoring !*

Dépendances acycliques

- Un module doit toujours dépendre de modules plus stables que lui
- En respectant cette règle, **le graphe de dépendances ne doit pas être cyclique**
- En cas de cycle, il faut faire de *refactoring* pour éliminer le cycle

Livraison

- Un module livré est un paquet fourni aux clients
- Livraison = publication = mise-en-forme
- livré : numéro de version
release.version.revision
- Seulement une fois livré, c'est réutilisable

Livraison / Configuration

- Penser à spécifier dans votre paquetage livré la configuration cible
 - + version de logiciels et bibliothèques utilisés
 - + configuration hardware
 - + Les contraintes de configuration

Livraison => Réutilisation

- **On ne peut réutiliser qu'un paquetage livré**
- On réutilise un paquetage complet et non une classe
- Le code est la propriété de l'auteur, c'est lui qui corrige les bugs
- *« Je ne veux pas le savoir du moment que ça marche, je ne cherche pas à savoir pourquoi »*
 - ➔ ... sauf au coin café, après avoir épuisé les sujets de discussions usuels