# Java M2 TTT

Valérie Gautard

# 1. De la programmation séquentielle à la programmation orientée objet

Chacun veut que ses programmes soient rapides, fiables, ergonomiques, lisibles… Comment faire ?
La POO tente d'apporter une réponse à cette question.

Definissons les facteurs de qualité logicielle

## *1.1. La qualité logicielle*

### *1.1.1. Facteurs internes et facteurs externes*

On distingue principalement deux catégories de qualités :

- Les facteurs externes de la qualité : ce sont les qualités dont la présence ou l'absence dans un logiciel peuvent être détectées par les utilisateurs du produit.
  Exemple : facilité d'emploi, robustesse…
- Les facteurs internes de qualité : Ce sont les qualités perceptibles seulement par des informaticiens

Seuls comptent les facteurs externes mais les facteurs internes sont déterminants pour l'obtention des facteurs externes.

### *1.1.2. Quelques définitions des facteurs externe de qualité*

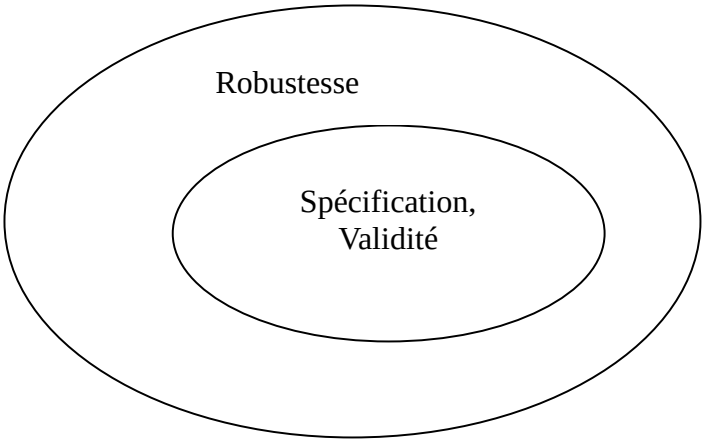
#### La validité

Définition

Cette notion est souvent difficile a atteindre, notamment parce que les spécifications sont difficiles à formaliser rigoureusement.

#### La robustesse

Définition




## Extensibilité

Définition

Pour améliorer l'extensibilité, deux principes sont essentiels :

- Simplicité de conception
- Décentralisation : plus les modules d'une architecture sont autonomes, plus il est probable qu'une modification simple n'affecte qu'un seul module ou un nombre restreint de modules

## La réutilisabilité

La réutilisabilité est l'aptitude d'un logiciel à être réutilisé en tout ou partie pour de nouvelles applications.

## La compatibilité

Définition

Exemple :
Variété de formats de fichiers incompatibles dans de nombreux OS
-> Un programme ne peut utiliser les résultats d'un autre programme que si les formats de fichiers sont compatibles…

## Autres qualités

Les qualités abordées jusqu'à présent sont celles qui bénéficient le plus de la POO mais il existe toutefois d'autres aspects.

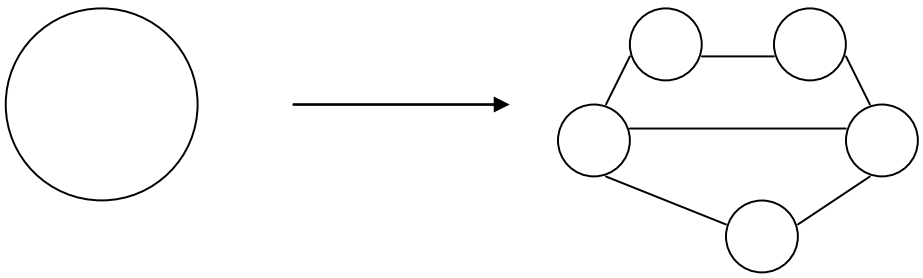
- 
- 
- 
- L'intégrité : C'est l'aptitude des logiciels à protéger leurs différentes composantes (prog, données, doc…) contre des accès ou des modifications non autorisées
- La facilité d'utilisation

## 1.2. La modularité

Pour tenter de satisfaire les qualités que l'on vient de citer, un mot vient à l'esprit : la production de logiciel modulaire (notamment pour l'extensibilité, la réutilisabilité et la compatibilité)

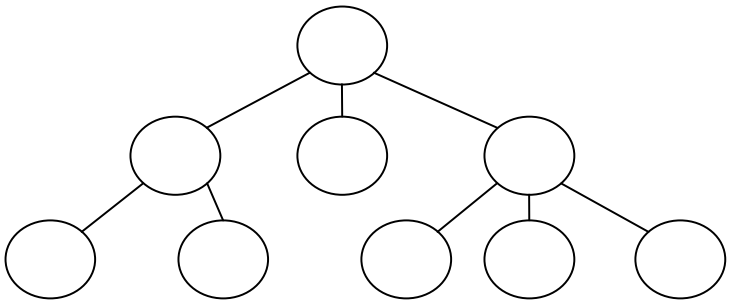
### 1.2.1. Cinq critères

#### 1.2.1.1. La décomposabilité

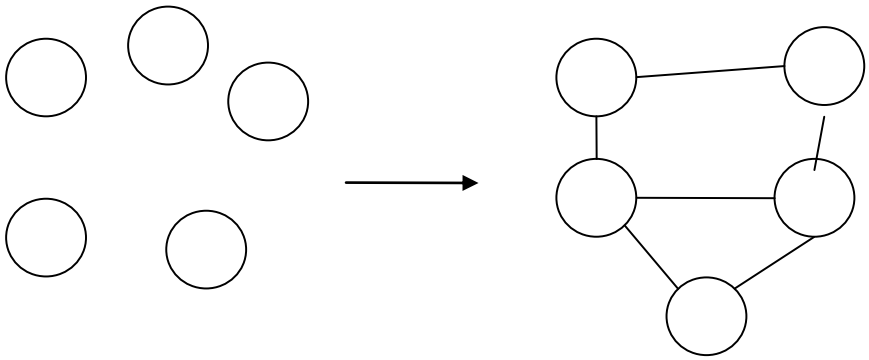


Cette méthode doit aider à réduire la complexité d'un problème.

Exemple : la conception descendante
Cette méthode consiste à commencer par une description abstraite du système et à l'affiner ensuite. Chaque sous-système est décomposé en système plus simple.



### 1.2.1.2. La composabilité modulaire



On utilise des éléments existants pour construire de nouveaux systèmes. On transforme ainsi le processus de conception du logiciel en un jeux de construction où les programmes seraient construits à partir d'éléments standards existants.

Exemple :
- les bibliothèques de sous-programme
- le langage de commande Unix avec |

#### 1.2.1.3. La compréhensibilité modulaire

Ce critère est important notamment pour la maintenance

Contre-exemple : la dépendance séquentielle :
Si un ensemble de modules a été conçu de façon à ne fonctionner correctement que si ces modules s'exécutent dans un certain ordre alors ceux-ci seront difficiles à comprendre individuellement.

#### 1.2.1.4. La continuité modulaire

Une méthode de conception répond au critere de continuité modulaire si une patite modification de la specification du probleme amene à modifier que tres peu de modules d'un système pr

#### 1.2.1.5. La protection modulaire

Exemple une méthode qui impose que chaque module qui lit des données est responsable de la vérification de leur validité répond au critère de protection modulaire.

### *1.2.2.Cinq principes*

Nous allons à présent exposer 5 principes devant être respectés pour obtenir une « bonne » modularité

#### 1.2.2.1. Unité modulaire linguistique

Les modules doivent correspondre à des unités syntaxiques du langage

Exemple : langage de programmation, langage de conception, de spécification

Ce principe est une conséquence des critères :

- décomposable : pour diviser le développement d'un système en tache séparées, chacune des taches doit correspondre à une unité syntaxique claire.
- composabilité : comment combiner autre chose que des unités closes ?
- protection : On peut contrôler l'étendue des erreurs que si les modules sont délimités syntaxiquement

Ce principe impose de choisir un langage approprié au type de programmation mise en œuvre. POO : java, C++

#### 1.2.2.2. Peu d'interface

Ce principe résulte des critères de :

- continuité
- protection : s'il y a trop de relation entre les modules alors tout changement ou toute erreur peut se propager
- composabilité : pour qu'un module soit utilisable dans un nouvel environnement, il ne doit pas dépendre d'un trop grand nombre de module

#### 1.2.2.3. Petite interface (couplage faible)

Principe :

Ce principe résulte des critères de continuité et de protection

Contre-exemple : COMMON en fortran ou chaque unité a accès à toutes les données => toutes les unités sont couplées de façon étroites !

#### 1.2.2.4. Interface explicite

Ces méthodes exigent que les modules se parlent peu, que leur conversation soit limitée à peu de mots mais elles imposent à présent que les conversations soient publiques !

Chaque fois que deux modules A et B communiquent, cela doit ressortir clairement du texte de A, de B

#### 1.2.2.5. Masquage de l'information

Principe :

Critère de continuité : si un module change d'une façon qui affecte ses éléments privés mais non son interface publique alors les autres modules ne seront pas affectés
Ce principe est également lié aux exigences de décomposabilité, de composabilité et de compréhensibilité pour développer séparément les modules d'un système d'un système, pour combiner différents modules existant, pour comprendre des modules individuels. Il est indispensable de connaitre parfaitement ce que chacun peut ou ne peut pas attendre des autres.

## 1.3. La route vers les objets

### 1.3.1. Objectif réutilisabilité

**La répétition en programmation**
La programmation est une activité très répétitive.

Exemple : tri, recherche, lecture, écriture dans BDD, fichiers etc…

Pourquoi cette répétition :

**Les approches basiques de la réutilisabilité** :

- réutilisation des codes sources : UNIX, Windows et plus généralement des environnements
- réutilisation du personnel
- réutilisation des conceptions : ouvrage de génie logiciel

**5 conditions**

- Variations des types : écrire du code applicable à différentes natures de type
- Variations des structures de données et d'algorithme : généralisation de la condition précédente
- Regrouper les opérations ayant des points communs
  Exemple : regrouper les procédures de création, d'insertion…

## 1.3.2. Les différents styles de programmation

Style applicatif (programmation séquentielle)

- Style de programmation fondé sur l'évaluation d'expressions ou le résultat ne depend que de la valeur des arguments
- Donne des programmes courts, faciles à comprendre
- usage intensif de la recursivite

Style impératif

- Fondé sur l'exécution d'instructions modifiant l'état de la mémoire.
- Utilisation de structure de contrôle et de structure de données

Routines

Définition

Exemple : les procédures, les fonctions

La décomposition de systèmes logiciels en routines est obtenue par la méthode de décomposition fonctionnelle descendante
Mais :

- chaque problème doit être spécifiable de façon simple ie définissable par un petit ensemble de paramètres.
- les problèmes individuels doivent être clairement distincts les uns des autres.

Cette approche ne permet pas d'exploiter les points communs qui peuvent exister

- Aucune structure de données ne doit être complexe.

## 1.3.3. Objets/fonctions

Un ensemble logiciel est un ensemble de mécanisme permettant d'effectuer certaines actions sur certaines données.

Question :
Doit-on structurer un logiciel autour des données ou des fonctions ?

Une méthode de conception satisfait au principe de continuité si elle conduit à des architectures qui n'ont pas besoin de fortes modifications pour tout changement mineur des spécifications.

Prenons l'exemple d'un logiciel de gestion de paie. Utilisé initialement pour produire des bulletins de paie en fonction de la présence, il est ensuite utilisé pour produire des statistiques, des informations fiscales etc…

En fait les données changent peu mais les fonctions évoluent

Conclusion :
Lorsqu'un système évolue, ses taches peuvent changer radicalement mais les données que le système manipule varient moins

Utilisation des données
L'intérêt d'une architecture basée sur les objets provient :

- La compatibilité : il est difficile de combiner des actions sans prendre en compte les structures de données concernées
- La réutilisabilité : il est difficile de construire des composants réutilisables s'ils matérialisent des actions seules et s'ils ignorent les données
- La continuité : au cours du temps, les structures de données restent l'élément le plus stable du système.



## 1.3.4. La conception par objets

**Une première définition**

Slogan de la conception objet :
Ne commencez pas par demander ce que fait le système mais demandez ce sur quoi il le fait !

**Description des objets**
Classe

Définition :
On appelle classe un ensemble de structure de données ayant des propriétés communes

Type abstrait

Définition

Définition
Les fonctions énumèrent les services disponibles sur les occurrences du type.

Il existe plusieurs catégories de fonctions :
- fonction constructeur, destructeur
- fonction transformateur
- fonction de consultation

Une définition précise
La conception par objets est la construction de systèmes logiciels prenant la forme de collections structurées d'implémentation et de types de données abstraites.

Surcharge et généricité

On appelle surcharge la possibilité d'attacher plusieurs significations à un même nom qui apparait dans un programme

exemple : operateur + pour des entiers, réels, vecteurs…

Définition

Un module n'est pas directement utilisable, c'est plutôt un canevas de module

Le plus souvent les paramètres, appelés paramètres génériques, représentent des types. Les modules réels, appelé occurrences du module générique sont obtenus en fournissant les types effectifs (paramètres génériques effectifs) qui correspondent à chacun des paramètres formels

La surcharge et la généricité offrent deux possibilités symétriques :
- la surcharge permet d'écrire le même code en utilisant plusieurs implémentations différentes d'une structure de données
- la généricité permet d'écrire le même code pour toutes les occurrences de la même implémentation d'une structure de données qui s'applique à différents types d'objets

La généricité résout le problème :
- variations des types
- variations dans les structures de données ou d'algorithme
- permet de réaliser une opération sans connaitre son implémentation

Paquetages

Avantage :

- un paquetage peut rassembler des éléments ayant une signification communes mais de nature informatique très différente : types, variable, routines…

Mais on a besoin d'autres techniques :

- pour libérer les utilisateurs du choix des représentations
- pour exploiter les points communs à l'intérieur des groupes d'implémentation

## 1.3.5. 7 pas vers les objets

Niveau 1 (structure modulaire fondée sur les objets)
Les systèmes sont découpés en modules sur la base de leur structure de données

Niveau2 abstraction des données
Les objets doivent être décrits comme des implémentations de type de données abstraites

Niveau 3 : Gestion automatique de la mémoire
Les objets inutilisés doivent être desalloués par le système sous-jacent, sans intervention du programmeur

Niveau 4 : classes
Tout type est un module et tout module est un type

Niveau5 : héritage
Une classe peut être définie comme une extension ou une restriction d'une autre classe

Niveau 6 : polymorphisme
Une entité de programme doit pouvoir faire référence à des objets de plusieurs classes et une opération doit pouvoir avoir des versions différentes dans des classes différentes

Niveau 7 : héritage multiple