Chapitre 5 : Principes de conception en paquets

« J'ai toujours rêvé d'un ordinateur qui soit aussi facile à utiliser qu'un téléphone. Mon rêve s'est réalisé : je ne sais plus comment utiliser mon téléphone. » ***Bjarne Stroustrup***

1. Objectifs de la conception en paquet

Les paquets (*packages* en anglais) définissent un moyen d'organiser les sources et de structurer la conception. Mais, les enjeux de la structuration en paquets vont bien au-delà :

- Réduire la complexité selon le principe « diviser pour régner ».
- Améliorer la développabilité. Les paquets sont affectés à un ingénieur ou à une équipe d'ingénieurs de développement qui peuvent ainsi travailler en parallèle.
- Diminuer le temps de compilation. Le temps de compilation complet peut durer plusieurs minutes, ce qui freine le développement s'il faut attendre plusieurs minutes avant de voir l'effet d'un changement dans le code.
- Simplifier la construction de la distribution.
- Améliorer la testabilité.
- Favoriser la réutilisation. Les paquets qui sont hautement interdépendants tendent à être rigides, non réutilisables et difficiles à maintenir.

Ces enjeux deviennent critiques à mesure que la taille du logiciel augmente.

1.1. Qu'est qu'un paquet ?

Il y a plusieurs dimensions à la notion de paquet en UML :

- groupe de classes (dossier en Java et C++),
- espace de noms (package en Java, namespace en C++),
- sécurité des classes (public ou package en Java, uniquement public en C++).

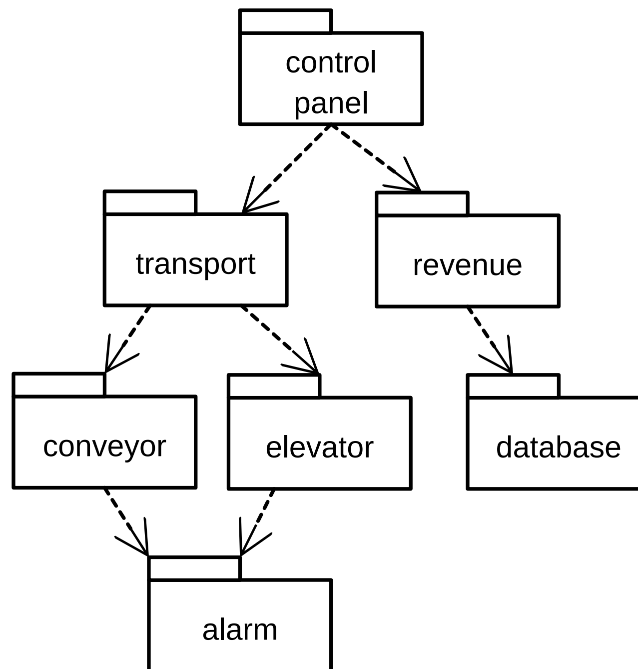
Les classes d'un paquet sont souvent compilées ensemble en bibliothèque : .jar, .dll, .lib, .so, .a.

Rappel : en Java les paquets se nomment par rapport à un nom de domaine Internet (à l'envers), par exemple : `fr.ensicaen.ecole.projet.paquet`. Un paquet définit aussi un espace de noms, ce qui permet de recopier un paquet d'un autre projet directement sans conflit avec l'existant.

1.2. Dépendance entre paquets

La dépendance signifie que certaines classes d'un paquet ont besoin de classes d'un autre paquet pour fonctionner. Une dépendance est la résultante d'une relation entre classes de paquets différents : héritage, implémentation d'interface, association ou simple utilisation.

Les liens entre paquets se traduisent par `import` en Java ou `#include` en C++.



1.3. Challenges de la conception en paquets

Les dépendances entre les paquets peuvent constituer des freins à la conception.

- Développement : quand un paquet A dépend d'un paquet B maintenu par une autre équipe, les évolutions du paquet B impactent le paquet A.
- Compilation : quand un paquet A dépend d'un autre paquet B, le paquet A doit être recompilé à chaque fois que le paquet B est modifié.
- Intégration : quand deux développeurs travaillent sur un même paquet, l'intégration nécessite une résolution manuelle des conflits pouvant être fastidieuse.

Les paquets présentent les mêmes challenges que les classes : développabilité, maintenabilité, réutilisabilité et testabilité.

1.4. La conception en paquets en questions

Questions :

- Quel est le meilleur critère de partitionnement ?
- Quels principes utiliser pour identifier les paquets ?
- Est-ce que les paquets doivent être définis au début du projet ou en cours de projet ?

Pour répondre à ces questions, on peut s'appuyer sur 6 principes qui gouvernent la composition et l'organisation d'un paquet, 3 principes pour la composition intra-paquet et 3 principes pour l'organisation inter-paquets.

2. Trois principes de composition d'un paquet

Ces 3 principes visent à répondre à la question que faut-il mettre dans un paquet ?

- Principe 1. Équivalence livraison / réutilisation
- Principe 2. Fermeture commune

- Principe 3. Réutilisation commune

2.1. Principe 1. Équivalence réutilisation / livraison[1]

Le point de vue est celui de la développabilité.
Définition

- **Un paquet doit être conçu comme un ensemble normalisé de classes qui offrent des services à d'autres paquets. Les paquets doivent être pensés comme des sous-projets à part entière avec, pourquoi pas, un numéro de version.**

Objectifs

- Augmenter l'étanchéité entre les parties du logiciel. Cela permet d'affecter les développements à des équipes de développement différentes.
- Laisser la possibilité d'utiliser une version antérieure des paquets efférents et ainsi éviter qu'un code ne soit dépendant des évolutions des paquets importés.

Exemples de structuration en paquets selon ce principe

- Un paquet avec les classes `Calendar`, `Date`, `Time`.
- Un paquet avec les classes `Point`, `Line`, `Polygon`.
- Un paquet avec les classes `Chart1D`, `Chart2D`, `Chart3D`, `Histogram1D`.

2.2. Principe 2. Fermeture commune

Le point de vue est celui de la maintenance.
Définition

- **Les classes impactées par les mêmes changements doivent être placées dans un même paquet.**

Objectif

- Un paquet ne doit pas avoir plus d'une raison de changer.

Motivation

- Réduire la zone d'impact des changements et donc réduire les coûts d'évolution et de maintenance.

Exemple de structuration en paquets selon ce principe

- Les classes `CellDatabase` (passerelle vers la base de données) et `CellEntity` (une table de la base de données) devraient aller dans le même paquet.

2.2.1. Liens avec les principes SOLID

Ce principe est le principe de responsabilité unique appliqué aux paquets. Un changement qui affecte un paquet affecte également toutes les classes de ce paquet mais aucun autre paquet.
Il est aussi étroitement lié au principe d'ouverture-fermeture. Puisque 100 % d'ouverture n'est pas possible, il faut mettre les classes impactées par un même changement dans le même paquet.

2.3. Principe 3. Réutilisation commune

Le point de vue est celui de la réutilisabilité.

1 Il s'agit ici d'une interprétation personnelle du principe présenté par Robert Martin dans son ouvrage « Agile Software Development, Principles, Patterns, and Practices » *Prentice Hall, 2000*, et qui est plus axée sur la réutilisabilté que l'originale.

Définition

- **Réutiliser une classe d'un paquet, c'est réutiliser le paquet entier.**
- **Si vous réutilisez une des classes dans un paquet, vous les réutiliser toutes.**

Objectif

- Les classes qui ont tendance à être utilisées ensemble appartiennent au même paquet.

Exemple de structuration en paquets selon ce principe

- Mettre ensemble la classe conteneur et son itérateur.

Motivation

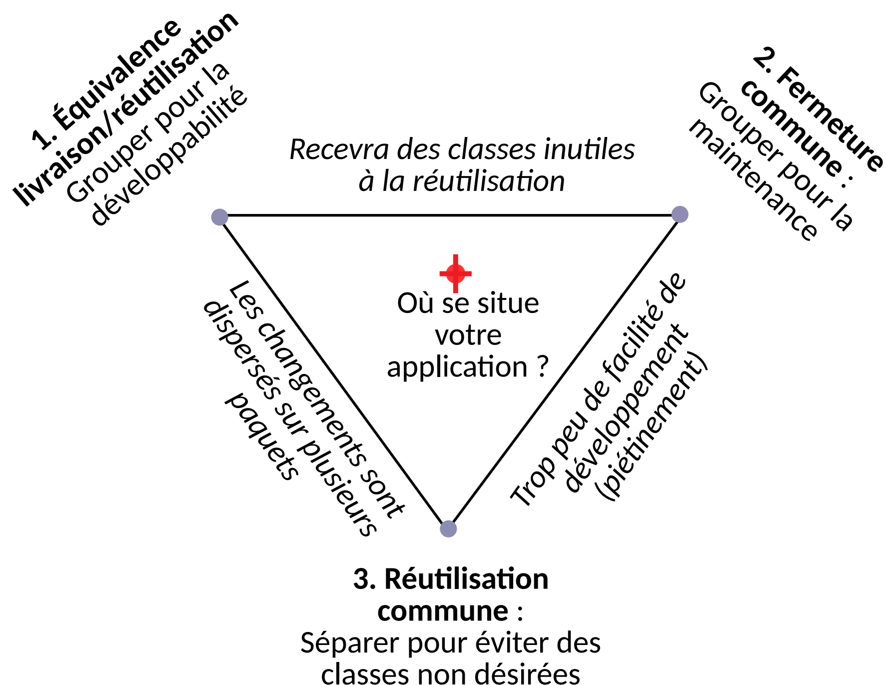
- Réutiliser une classe d'un paquet force à dépendre de tout le paquet. Si l'on place 2 classes totalement indépendantes dans un même paquet, on oblige les utilisateurs d'une classe à dépendre de l'autre classe alors que c'est inutile et coûteux.

2.3.1. Lien avec les principes SOLID

C'est le principe de ségrégation des interfaces appliqué aux paquets. Les classes qui ne sont pas liées les unes aux autres par des relations de classes ne devraient pas être dans le même paquet.

2.4. Conflits d'intérêts entre ces 3 principes

Ces principes peuvent se révéler contradictoires entre eux. Nous devons constamment choisir entre ces trois principes pour construire nos paquets. La figure ci-dessous illustre ces conflits.



3. Trois principes d'organisation entre paquets

Ces 3 principes visent à répondre à la question quelles relations doit-on garder entre les paquets ?

- Principe 4. Dépendances acycliques
- Principe 5. Relation dépendance / stabilité

- Principe 6. Stabilité des abstractions

3.1. Principe 4. Dépendances acycliques

Définition

- **Les dépendances entre paquets doivent former un graphe direct acyclique.**

Objectif

- Supprimer les dépendances circulaires entre les paquets.

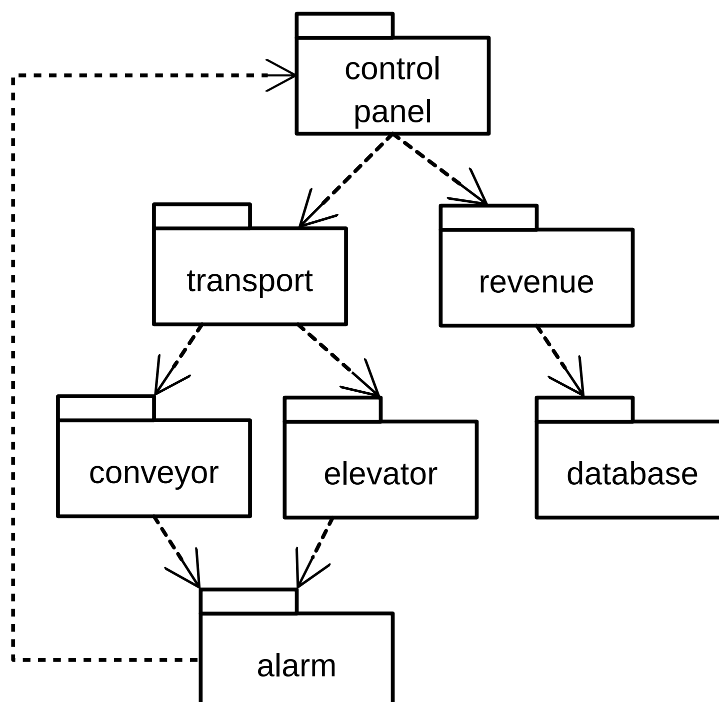
Motivations

- Augmenter la réutilisabilité.
- Réduire les interférences entre les équipes de développement.
- Permettre la testabilité.
- Réduire les temps de compilation.

Remarque : IntelliJ permet de voir le graphe de dépendance entre paquets et repérer les dépendances acycliques (voir le menu Analyze).

3.1.1. Les cycles ruinent l'harmonie

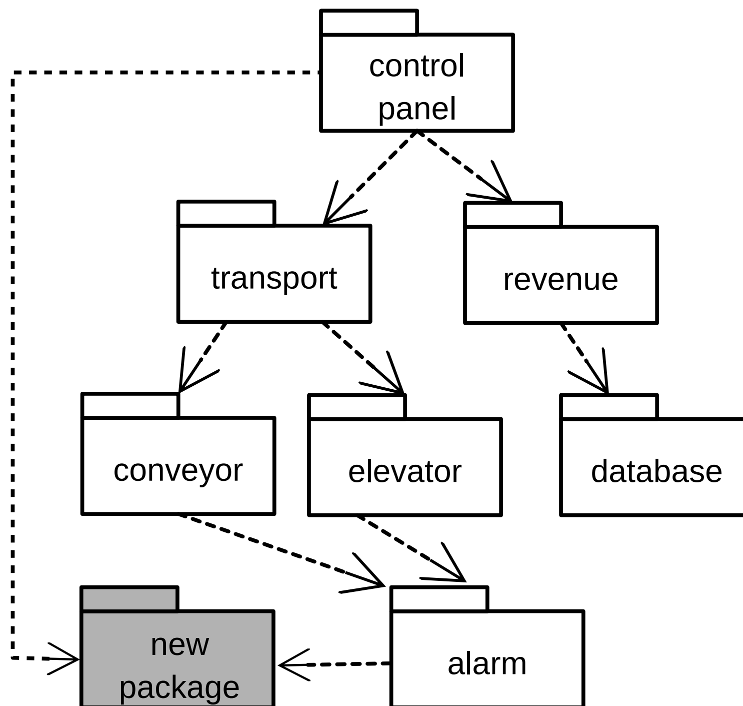
Dans l'exemple dans la figure ci-dessous, si on modifie le paquet `transport`, cela impacte les paquets `conveyor` et `elevator`, qui impactent eux-mêmes le paquet `alarm`, qui impacte les paquets `control_panel` et `revenue` et `database`. Cela peut aussi s'illustrer avec le cas de la compilation. Si on modifie le paquet `transport`, on oblige à recompiler tout le projet ! En Python, ce projet n'est même pas compilable.



3.1.2. Solution 1 : Casser les cycles par ajout d'un paquet de dépendances communes

La première solution consiste à introduire un nouveau paquet avec les classes de `control_panel` et

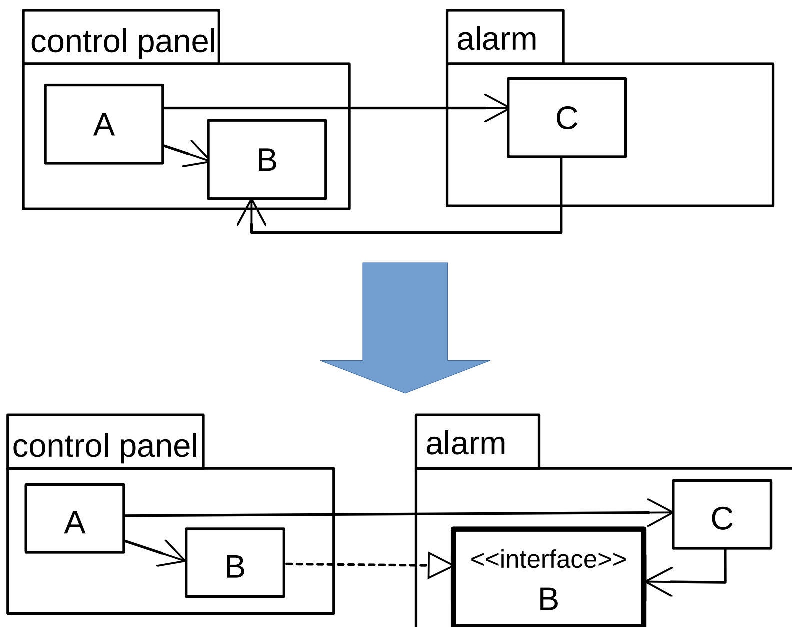
`alarm`.



Ce n'est pas toujours possible. On casse potentiellement la cohésion dans le paquet `control_panel`.

3.1.3. Solution 2 : Casser les cycles par inversion des dépendances

La seconde solution consiste à ajouter une interface dans `alarm` pour inverser la dépendance entre les deux paquets.



3.2. Principe 5. Relation dépendance / stabilité

Le principe des dépendances stables (SDP) est basé sur le constat que le code est volatil (ie, malléable) et

que les changements se répercutent sur un système le long des lignes de dépendance. Du point de vue de la volatilité, la dépendance est transitive.

Définition

- **Un paquet ne doit dépendre que de paquets plus stables que lui.**
- Note : la **stabilité** d'un paquet s'entend comme la **difficulté à changer** le paquet. Donc, la phrase précédente devient : un paquet ne doit dépendre que de paquets plus difficiles à changer que lui.

Objectif

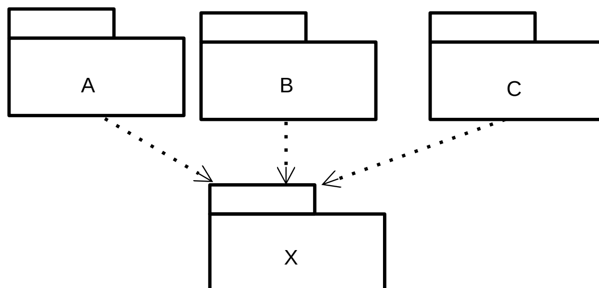
- Un paquet que l'on veut stable ne devrait pas dépendre de paquets qui sont susceptibles de changer au cours du cycle de vie. S'il le fait, les changements dans ses dépendances entraîneront des risques de corrompre le logiciel.
- Or, nous avons naturellement tendance à faire dépendre les abstractions sur les implémentations. Il est donc nécessaire d'inverser cette dépendance pour la faire aller dans le sens de la stabilité.

Motivation

- On cherche à limiter l'impact des changements les plus fréquents et maximiser la stabilité globale de l'application.

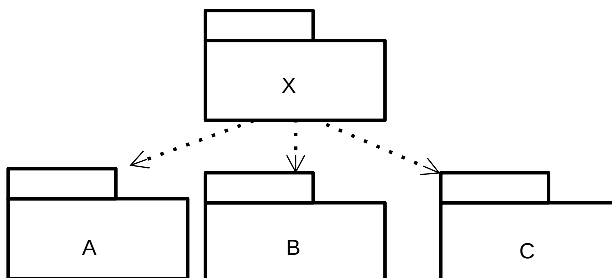
3.2.1. Paquet stable

Un paquet avec beaucoup de dépendances afférentes doit être très stable (ie, parce que difficile à changer) pour limiter l'impact des changements.



3.2.2. Paquet instable

Un paquet avec peu de dépendances afférentes peut être instable (ie, parce que facile à changer).



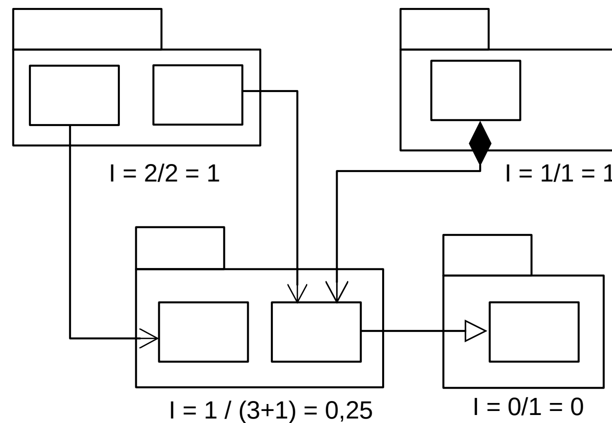
3.2.3. Une mesure d'instabilité

Instabilité I = Ce / (Ca + Ce)

- Ce = couplages efférents. Nombre de classes dans le paquet qui dépendent de classes en dehors

du paquet (*flèches sortantes*).

- Ca = couplages afférents. Nombre de classes en dehors du paquet qui dépendent de classes du paquet (*flèches entrantes*).



- Les valeurs de $I \in [0, 1]$
  - 0 : paquet stable
  - 1 : paquet instable

3.2.4. Utilisation de la mesure d'instabilité

Tous les paquets ne peuvent pas être stables. S'ils étaient tous stables le système ne serait plus évolutif. Le graphe des dépendances doit aller des packages instables (packages faciles à modifier) aux packages stables (packages difficiles à modifier). Autrement dit, la valeur d'instabilité d'un paquet doit être supérieure à la valeur d'instabilité des paquets dont il dépend. Pour inverser la direction de stabilité, on reprend les solutions des sections 3.1.2 et 3.1.3 :

- principe d'inversion des dépendances,
- création d'un paquet intermédiaire et déplacement des classes dont dépend la stabilité dans ce paquet.

3.3. Principe 6. Stabilité des abstractions

Définition

- **Le degré d'abstraction d'un package doit correspondre à son degré de stabilité.**
- **Les paquets les plus stables doivent être les plus abstraits.**
- **Les paquets instables doivent être concrets.**

Objectif

- Un paquet stable devrait être abstrait de sorte que sa stabilité ne l'empêche pas d'être étendu.
- Un paquet instable peut être concret, car son instabilité permet à son code interne d'être facilement changé.
- Un paquet doit être aussi abstrait qu'il est stable.

Motivation

- Les classes abstraites portent la logique de l'application. Elles forment ainsi l'architecture de l'application.

3.3.1. Une mesure d’abstraction

Degré d’abstraction A = Na / N

- Na = nombre de classes abstraites et d’interfaces.
- N = nombre total de classes.

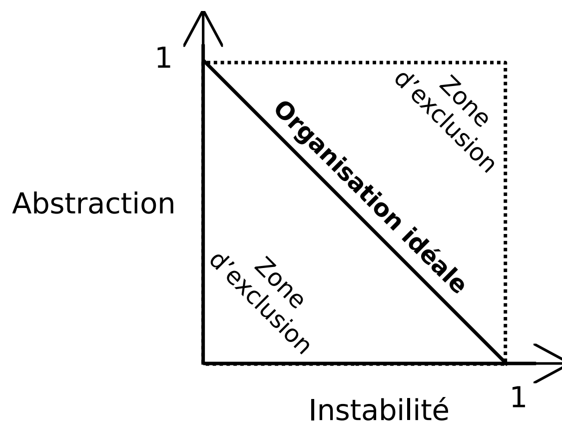
Les valeurs de A ∈ [0, 1]

- 0 : pas de classe abstraite dans le paquet.
- 1 : que des classes abstraites dans le paquet.

3.3.2. Relation instabilité / abstraction

Mesure de qualité d’un paquet :

- Calcul de la distance à l’organisation idéale entre (I)nstabilité (I) et (A)bstraction
  - Paquet (I = 0, A = 0) : non souhaitable.
  - Paquet (I = 1, A = 1) : inutile.
  - Organisation idéale : la droite d’équation A = 1 – I.
  - **Mesure :** distance à l’organisation idéale d = |A + I – 1|, d ∈ [0,1].



4. Méthode de conception en paquets

Est-ce que les paquets doivent être définis au début du projet ou en cours du projet ?

- Réponse : L’organisation en paquet d’un projet ne peut qu’être conçue au fur et mesure de l’avancée du projet. Les dépendances entre paquets croissent et évoluent avec l’application. Il faut souvent choisir entre développabilité, réutilisabilité et maintenabilité.

Démarche

- Les premiers paquets sont inspirés de l’architecture. Puis, les 6 principes précédents sont appliqués dès que se posent les questions de développabilité, réutilisabilité et maintenance.

Conséquence

- En fin de projet, le diagramme de paquets a très peu à voir avec la description de la fonction de l’application ni même de l’architecture. Il forme plutôt une **carte de construction de l’application**.

5. Conclusion

5.1. Conception

Lors de l’inclusion de classes dans un paquet, nous devons choisir entre développabilité, réutilisabilité et maintenabilité. Ce choix conduit à des remises en cause périodiques de la conception en paquets. Le partitionnement idéal des classes en paquets ne peut donc pas être anticipé avant d’avoir défini les classes et leurs relations. Il n’y a aucune métrique pour calculer automatiquement la cohésion d’un paquet. Les mesures de stabilité et d’abstraction sont utilisées pour produire une organisation à faible couplage.

5.2. Gestion de la granularité des paquets

Il faut décomposer l’application en paquets pour gérer correctement les versions et permettre une réelle réutilisation. On regroupera dans un même paquet les classes qui sont utilisées ensemble et qui sont impactées par les mêmes changements.

5.3. Gestion de la stabilité de l’application

Les paquets doivent être organisés en un arbre de dépendances.

- Placer les paquets les plus stables à la base de l’arbre.
- Mettre des interfaces entre les paquets dans le sens de la stabilité comme des pare-feux contre les changements.
- Placer les interfaces dans les paquets les plus stables.

6. Lectures

Martin, Robert C., "Principles of OOD". « https://web.archive.org/web/20220121055640/butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod ».

Martin, Robert C. (1996). "Granularity". C++ Report. SIGS Publications Group. Nov-Dec 1996.

Martin, Robert C. (2002). Agile Software Development, Principles, Patterns, and Practices. Prentice Hall.