ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

05
Chapitre

Principes de conception en paquets

2I1AC3 : Génie logiciel et Patrons de conception

Régis Clouard, ENSICAEN - GREYC

« 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

Objectifs de la conception en paquet

- Pourquoi ne pas mettre tous les fichiers dans un seul paquet ?

- Les enjeux de la structuration en paquets :
  - Apporter une vue concise de la conception (auto-documentation)
  - Améliorer la développabilité
  - Diminuer le temps de compilation
  - Augmenter la testabilité
  - Favoriser la réutilisation
- Ces enjeux deviennent critiques à mesure que la taille du logiciel augmente

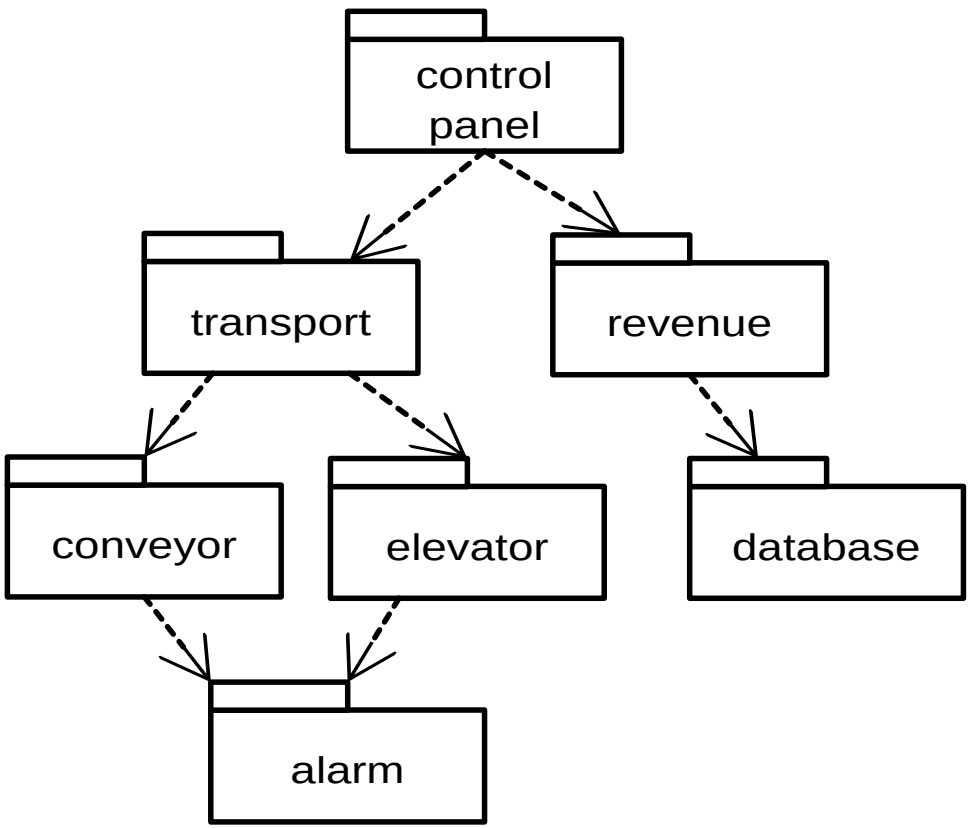
Qu'est qu'un paquet ?

- 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, public en C++)
- Depuis Java 9, la notion de paquet est renforcée avec l'introduction des modules
  - Un module est un ensemble de paquets conçus pour être réutilisés ensemble
    - Comme les classes, certains paquets sont publics d'autres privés

- Rappel : en Java les paquets se nomment à partir d'un nom de domaine Internet à l'envers
  - *fr.ensicaen.ecole.projet.paquet*

Dépendance entre paquets

- La dépendance signifie que certaines classes d'un paquet ont besoin de classes d'un autre paquet pour fonctionner.
- Une dépendance naît d'une relation entre classes :
  - Héritage
  - Implémentation d'interface
  - Association
  - Utilisation
- Liens entre paquets
  - import en Java
  - include en C++



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, 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. Il n'est pas rare que la compilation complète d'un logiciel dure plusieurs heures.
  - **Intégration** : quand deux développeurs travaillent sur un même paquet, l'intégration peut conduire à des conflits qui doivent être réglés manuellement.

Le « syndrome du lendemain matin »

- Vous finissez votre journée et votre programme fonctionne. Vous partez avec le sentiment du devoir accompli
- Vous revenez le lendemain matin, et votre programme ne fonctionne plus
  - Quelqu'un est resté plus tard que vous et a enregistré des modifications qui rendent votre code inutilisable. En attendant son retour, vous ne pouvez plus travailler

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 au cours du projet ?
- Pour répondre à ces questions, on peut s'appuyer sur 6 principes qui gouvernent la composition et l'organisation des paquets (Robert C. Martin, *Agile Software Development Principles*, *Patterns and Practices*, 2003)

Trois principes de composition d'un paquet

- Que mettre dans un paquet ?
- 3 alternatives selon 3 points de vue :
  - Principe 1. Équivalence livraison / réutilisation
  - Principe 2. Fermeture commune
  - Principe 3. Réutilisation commune

Principe 1. Équivalence réutilisation / livraison

- Point de vue de la **développabilité**
- Définition
  - Un paquet est conçu comme un **ensemble normalisé de classes** qui offrent des services à d'autres paquets
  - Mettre dans un même paquet des classes de même préoccupation (ou issues d'une même sous-équipe de développement)
- Exemples de structuration en paquets
    - Un paquet avec les classes Calendar, Date, Time
    - Un paquet avec les classes Point, Line, Polygon
- Implémentation
  - Un paquet doit être pensé comme une **bibliothèque de classes** à part entière et indépendante (pourquoi pas versionnée) : une réponse au « syndrome du lendemain matin »

Principe 2. Fermeture commune

- Point de vue de la **maintenance**
- Définition
  - Les classes impactées par les **mêmes changements** doivent être placées dans un même paquet
  - Un paquet ne doit pas avoir plus d'une raison de changer
- Exemple de structuration en paquet :
  - Les classes CellDatabase (base de données) et CellEntity (table) devraient aller dans le même paquet
- Motivation
  - Réduire l'impact des changements et donc réduire les coûts d'évolution et de maintenance

Liens avec les principes SOLID

- C’est le principe SOLID de responsabilité unique appliqué aux paquets
  - Un changement qui affecte un paquet ne devrait n’affecte que des classes de ce paquet, et aucune classe d’autres paquets
- Ce principe 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

Principe 3. Réutilisation commune

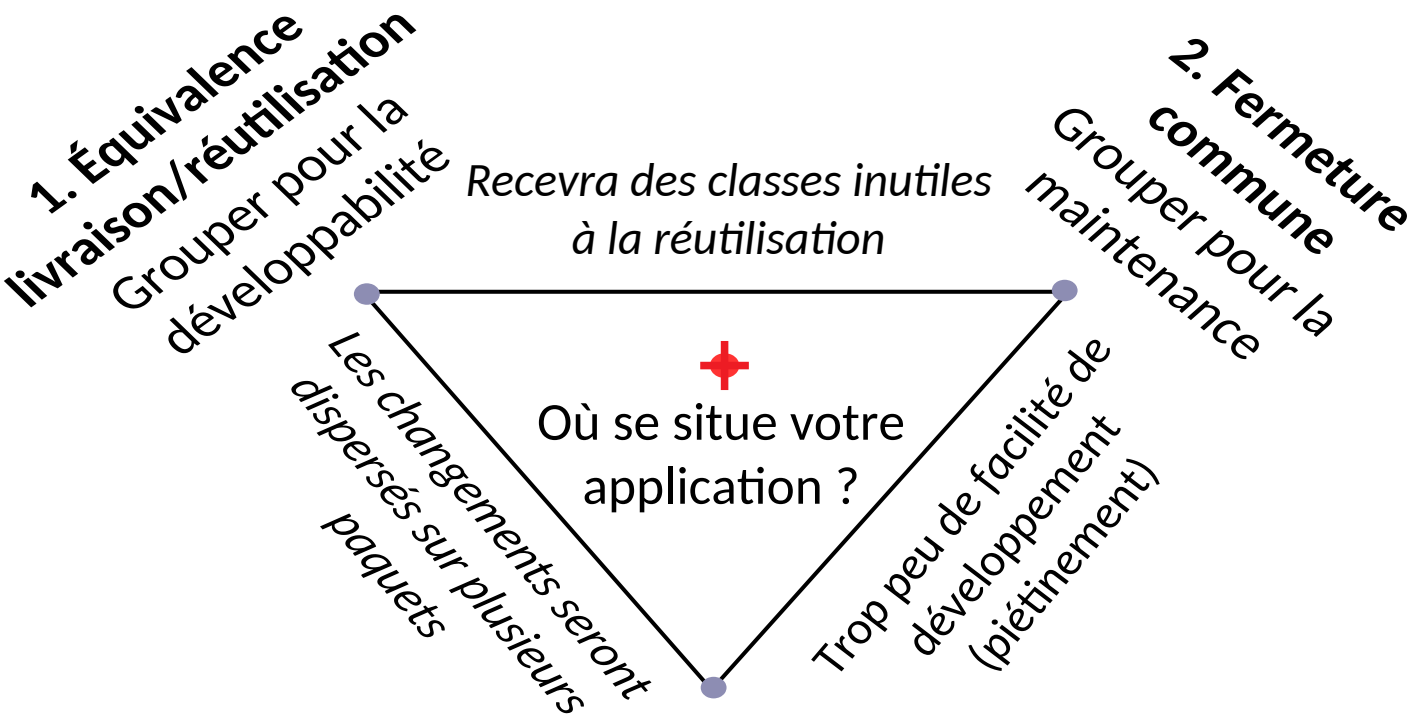
- Point de vue de la **réutilisation**
- Définition
  - **Réutiliser une classe d'un paquet, c'est réutiliser le paquet entier**
  - Les classes qui ont tendance à être utilisées ensemble appartiennent au même paquet
- Exemple de structuration en paquet :
    - Mettre dans un paquet la classe conteneur et celles de ses itérateurs
- Motivation
  - Réutiliser une classe d'un paquet force à dépendre de tout le paquet. Si 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.
- Ce principe nous dit plus quelles sont les classes qu'il faut écarter du paquet.

Lien avec les principes SOLID

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

Balance entre ces principes

- Ces principes peuvent se révéler contradictoires entre eux.
- Nous devons choisir entre ces trois principes pour construire chacun de nos paquets.

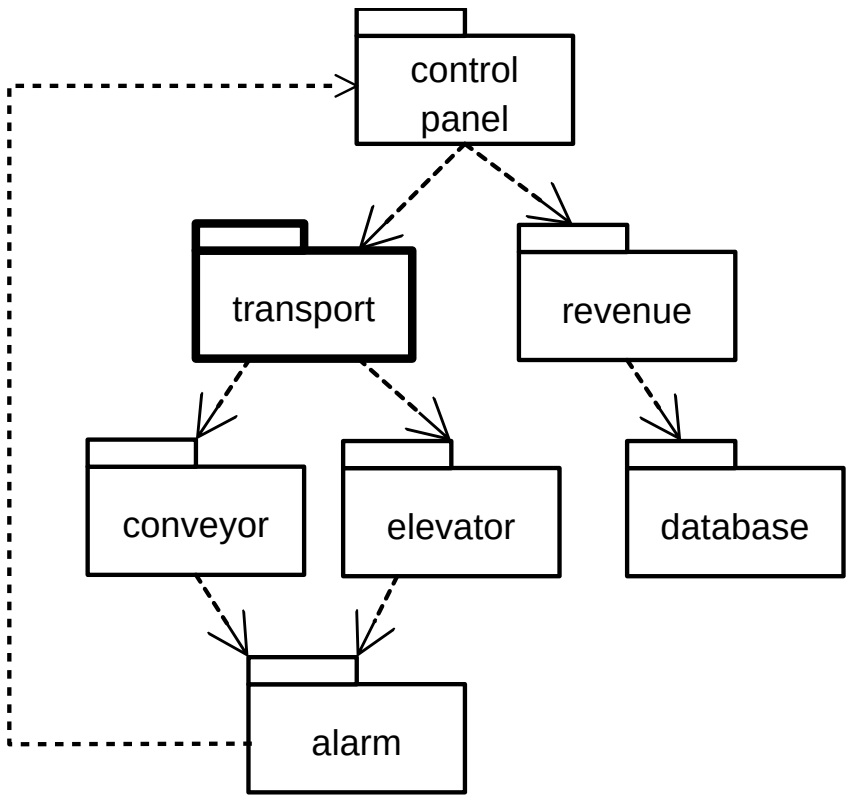


Trois principes d’organisation en paquets

- Quelle organisation entre paquets ?
- 3 principes à respecter :
  - Principe 4. Dépendances acycliques
  - Principe 5. Relation dépendance / stabilité
  - Principe 6. Stabilité des abstractions

Principe 4. Dépendances acycliques

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

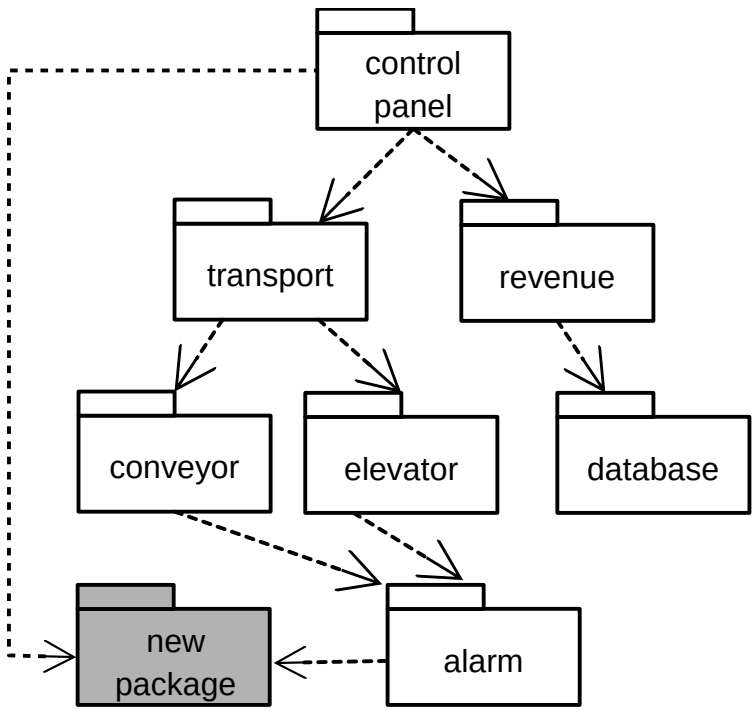


Dépendances acycliques

- Motivations
  - Augmenter la réutilisabilité.
  - Réduire les interférences entre les équipes de développement.
  - Permettre la testabilité.
- Remarque : IntelliJ permet de voir le graphe de dépendance entre paquets et repérer les dépendances acycliques (menu Analyze)
- Comment transformer un graphe cyclique en graphe acyclique ?

Casser les cycles : solution 1
Ajouter un paquet de dépendances communes

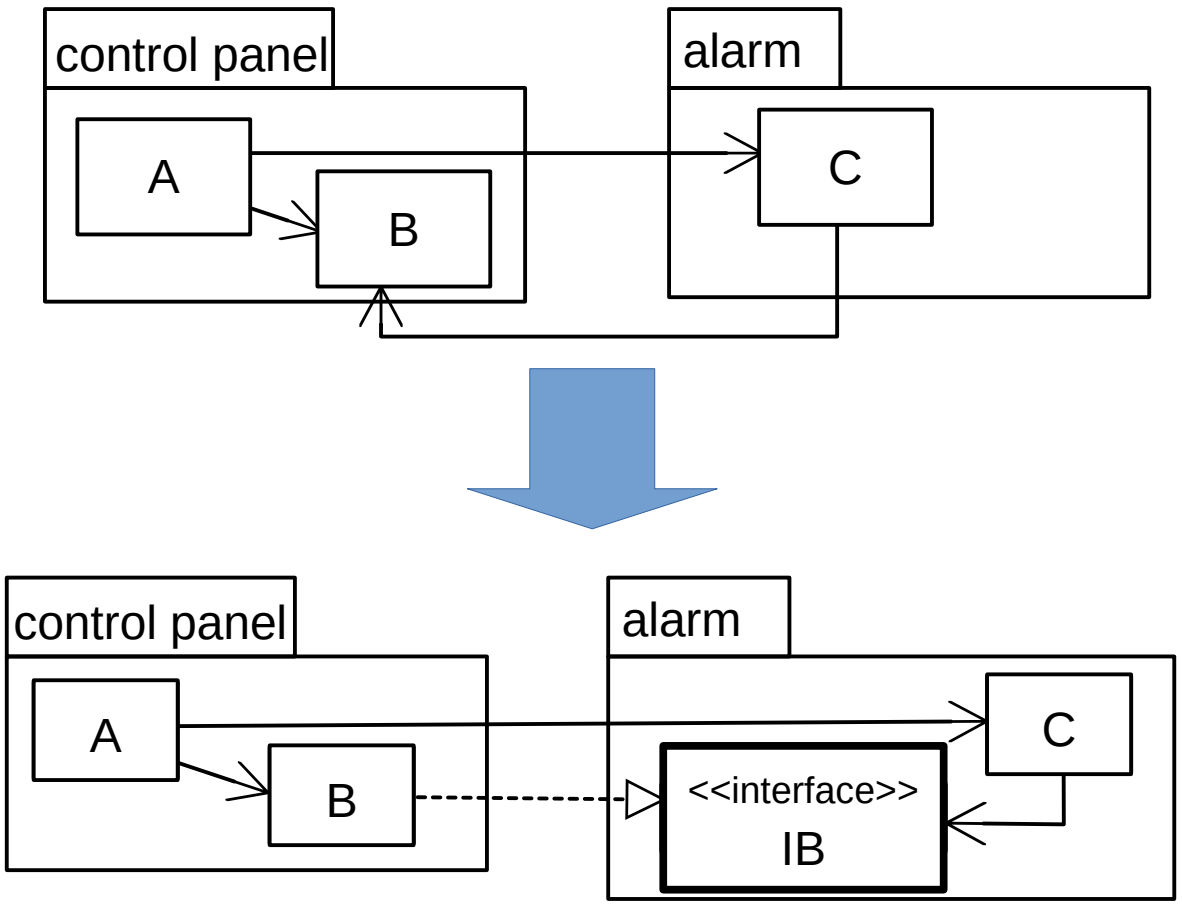
- Introduire un nouveau paquet avec les classes de `control_panel` qui sont liées à des classes dans `alarm.`



- Ce n'est pas toujours possible. On casse potentiellement la cohésion dans les paquets `control_panel` ou `alarm.`

Casser les cycles : solution2
Inverser les dépendances

- Ajouter une interface dans `alarm` pour inverser la dépendance entre les deux paquets.

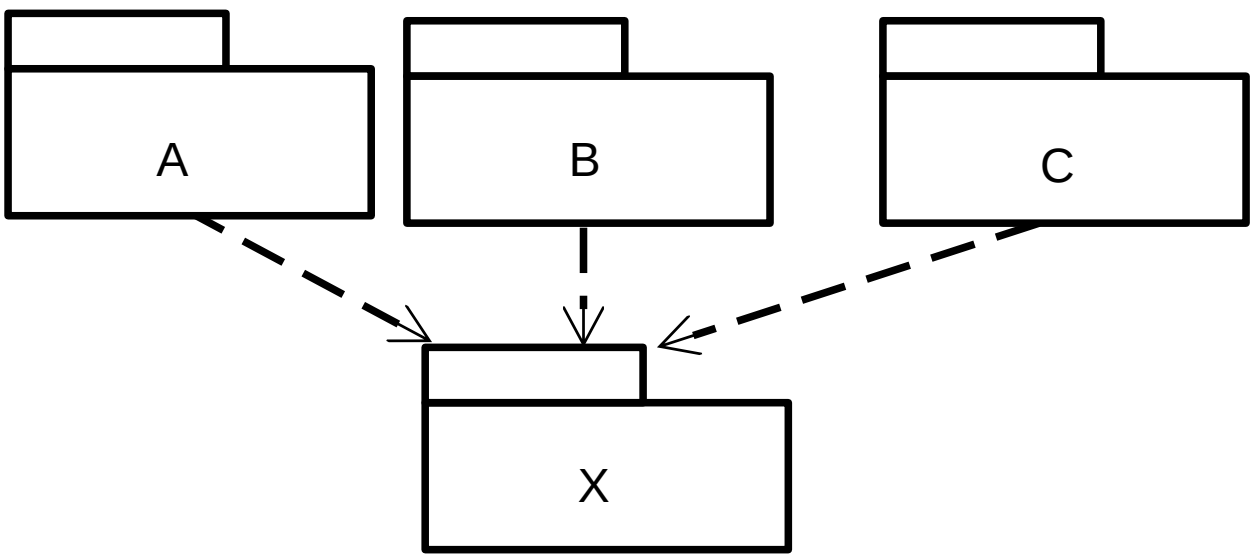


Principe 5. Relation dépendance / stabilité

- Définition
  - **Un paquet ne doit dépendre que de paquets plus stables que lui**
    - Note : La stabilité d'un paquet se réfère à la **difficulté à changer** le paquet. Plus un paquet est difficile à changer (parce qu'il est utilisé par plusieurs autres paquets), plus il doit être stable.

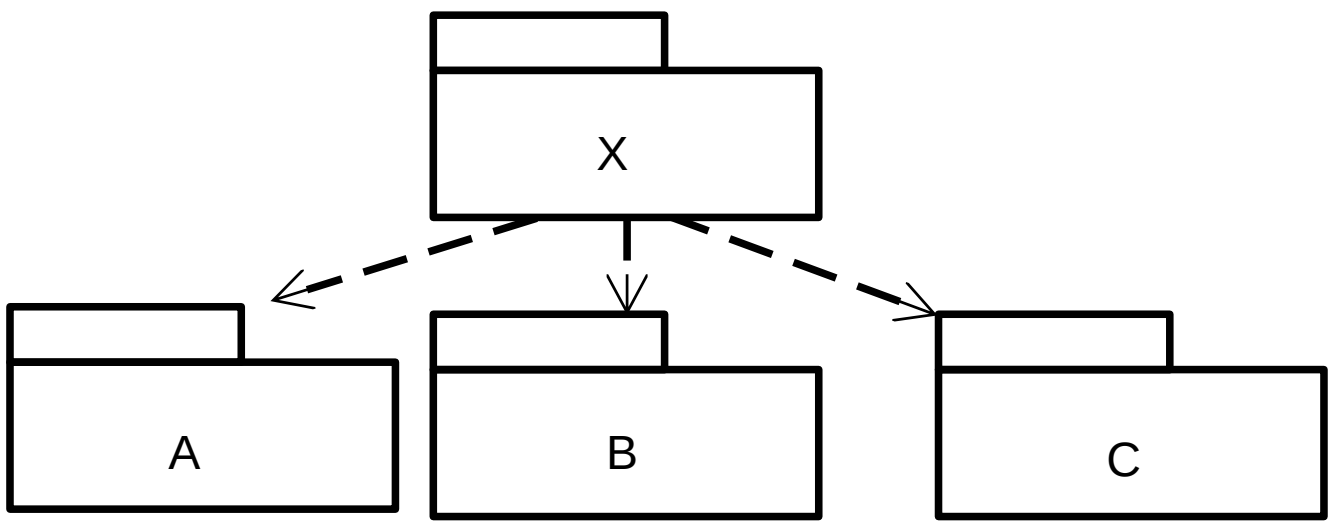
Paquet stable

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



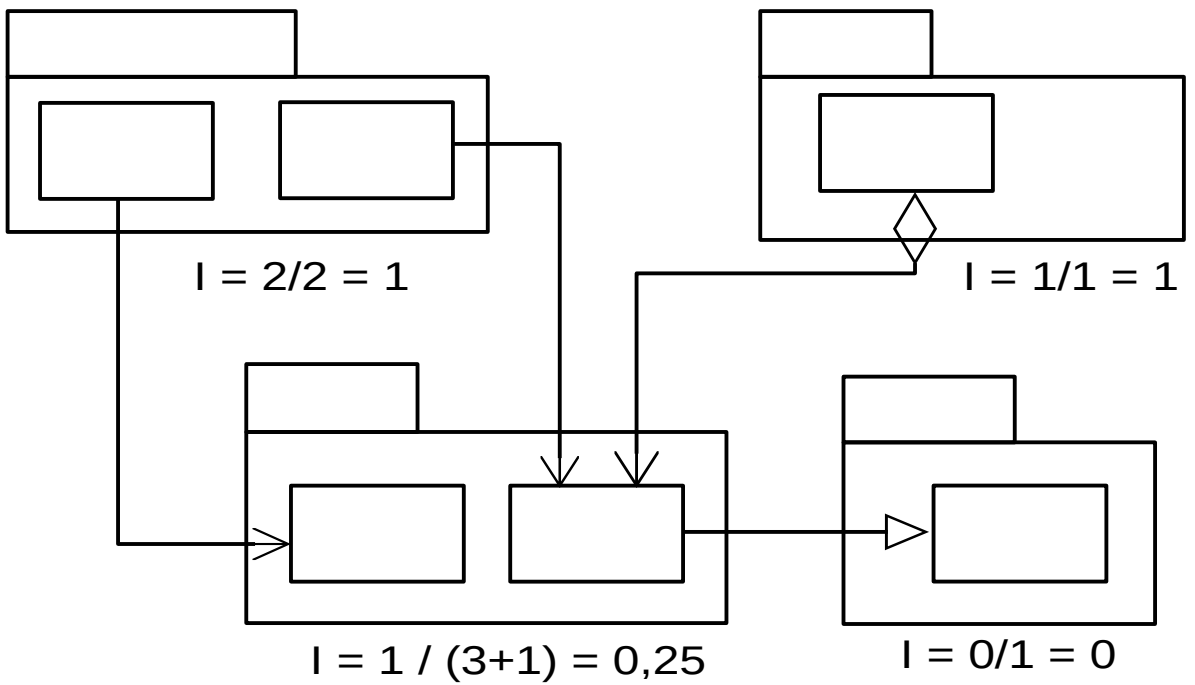
Paquet instable

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



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*).



- Valeurs dans [0, 1]
  - 0 : paquet stable
  - 1 : paquet instable

Utilisation de la mesure d'instabilité

- Le graphe des dépendances doit aller des packages instables (packages faciles à modifier) vers les packages stables (packages difficiles à modifier).
  - Tous les paquets ne peuvent pas être stables. S'ils sont tous stables le système ne serait plus évolutif.
  - Mais, la valeur d'instabilité d'un paquet doit être supérieure à la valeur d'instabilité des paquets dont il dépend.
- Pour résoudre le problème de la direction de stabilité, 2 solutions :
  1) Créer un paquet intermédiaire et déplacer les classes dont dépend la stabilité dans le paquet.
  2) Inverser les dépendances.

Principe 6. Stabilité des abstractions

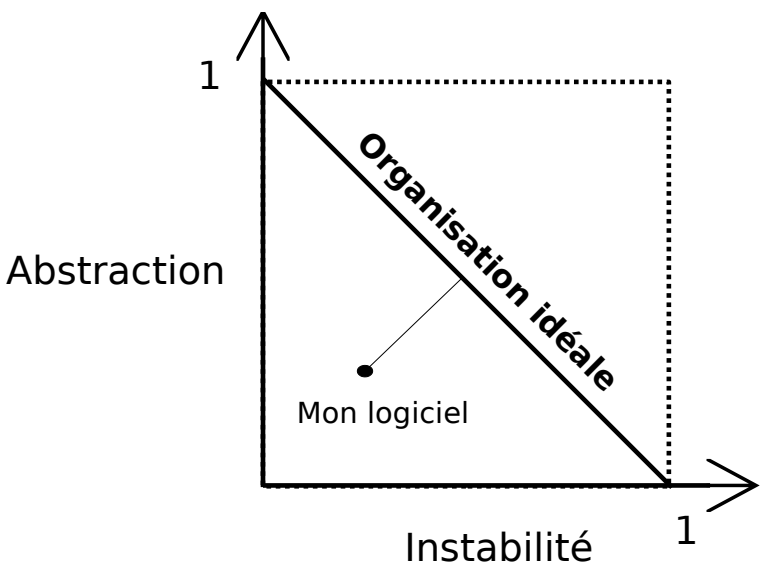
- Définition
  - **Le degré d'abstraction d'un package doit correspondre à son degré de stabilité.**
  - **Les paquets les plus stables doivent être les plus abstraits.**
  - **Les paquets instables doivent être concrets.**
- Motivation
  - Limiter l'impact des changements les plus fréquents

Une mesure d'abstraction

- Degré d'abstraction A = Na / N
  - Na : nombre de classes abstraites et d'interfaces
  - N : nombre total de classes
- Valeurs dans [0, 1]
  - 0 : pas de classes abstraites dans le paquet
  - 1 : que des classes abstraites dans le paquet

- Mesure de qualité d'un paquet :

  Distance à l'organisation idéale = |A + I - 1|
  - Paquet (I = 0, A = 0) : non souhaitable
  - Paquet (I = 1, A = 1) : inutile



Utilisation de la mesures des abstractions

- 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
- Motivation
  - Les classes abstraites portent la logique de l’application. Elles forment ainsi l’architecture de l'application (aka Framework)

Lien avec les principes SOLID

- Les deux principes précédents :
  - Principe 6. Stabilité des abstractions
  - Principe 5. Relation dépendance / stabilité

  forment le principe SOLI(D) d'inversion des dépendances appliqué aux paquets
  - Vous ne devez dépendre que de paquets qui ne changeront probablement pas
  - Les paquets abstraits ne doivent pas dépendre de paquets concrets

Méthode de conception en paquets

- Est-ce que les paquets doivent être définis au début du projet ou au 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 modifier ses priorités entre développabilité, réutilisabilité et maintenabilité
- Démarche
  - Les premiers paquets sont souvent inspirés de l'architecture choisie
  - Puis, les principes 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 l'architecture
  - Le diagramme de paquets forme plutôt une **carte de construction de l'application**

Démonstration d’une restructuration en paquets à partir des principes

Conclusion

- 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.

Conclusion

- Gestion de la granularité des paquets
  - Décomposer l'application en paquets pour gérer correctement les versions et permettre une réelle réutilisation.
  - Regrouper dans un même paquet les classes qui sont utilisées ensemble et qui sont impactées par les mêmes changements.
- Gestion de la stabilité de l'application
  - Organiser les modules 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.