



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

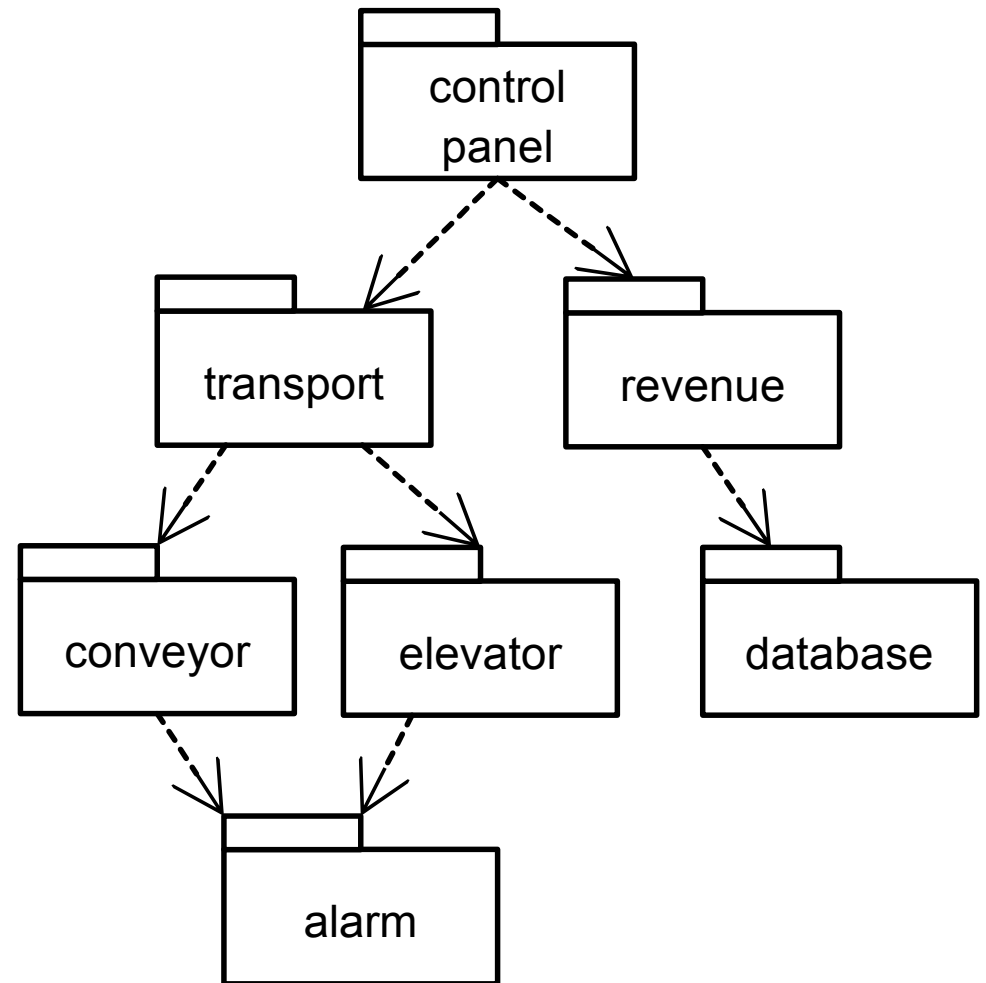
- Les paquets (*packages*) définissent un moyen d'organiser les sources et de structurer la conception.
 - Les paquets sont affectés à un ingénieur ou à une équipe d'ingénieurs de développement.
- Les enjeux de la structuration en paquets :
 - réduire la complexité selon le principe « diviser pour régner »,
 - diminuer le temps de compilation,
 - simplifier la construction de la distribution,
 - améliorer 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 (en c++ dossier),
 - espace de noms (en c++ namespace),
 - sécurité des classes (public ou package-private) (en c++ pas d'équivalent).
- Les classes d'un paquet sont souvent compilées ensemble en bibliothèque :
 - dll, .lib, .so, .a, .jar
- Rappel : en Java les paquets se nomment en rapport à un nom de domaine Internet :
 - 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.
- Dépendances
 - 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 maintenue 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 fusion manuelle.
 - Test : quand on veut tester un paquet A, il faut le lier à tous les paquets afférents.
- Les paquets présentent les mêmes challenges que les classes : ouverture / fermeture, maintenabilité, réutilisabilité, testabilité.

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 avant les classes ou les classes avant les paquets ?
- Pour répondre à ces questions, il y a plusieurs principes qui gouvernent la création, les relations et l'usage d'un paquet.
 - Ces principes se rapportent aux deux qualités attendues d'une conception:
 - ▶ forte cohésion,
 - ▶ faible couplage.

Six principes de conception en paquets

- Augmenter la cohésion
 - Principe 1. Équivalence livraison / réutilisation
 - Principe 2. Fermeture commune
 - Principe 3. Réutilisation commune
- Réduire le couplage
 - Principe 4. Dépendances acycliques
 - Principe 5. Relation dépendance / stabilité
 - Principe 6. Stabilité des abstractions

Principe 1. Équivalence réutilisation / livraison

■ Définition

- Les paquets doivent être créés avec des classes réutilisables.
- Soit toutes les classes à l'intérieur d'un paquet sont réutilisables soit aucune d'entre elles.

■ Objectif

- *Point de vue de la réutilisation.*
- Les paquets doivent être gérés comme des bibliothèques à part entière (p. ex. numéro de version).
 - ▶ p. ex. une bibliothèque de classes pour visualiser des données avec différentes représentations graphiques.

■ Motivations

- Éviter d'être dépendant des évolutions des paquets efférents.
- Laisser la possibilité d'utiliser une version antérieure des paquets efférents (cf. le « syndrome du lendemain matin » de Robert Martin).

Définition de la réutilisabilité

■ Réutilisabilité

- La recopie de code n'est pas de la réutilisation. Le code copié devient du code normal.
- Un paquet a la qualité de la réutilisabilité si et seulement si le réutilisateur n'a pas besoin de regarder le code pour le réutiliser (autre que l'interface publique).
- Le paquet doit être réutilisé comme s'il s'agissait d'une archive compilée (eg. .jar, .dll).

Principe 2. Fermeture commune

- Définition
 - Les classes impactées par les mêmes changements doivent être placées dans un même paquet.
- Objectif
 - *Point de vue de la maintenance.*
 - Un paquet ne doit pas avoir plus d'une raison de changer.
- Motivation
 - Réduire l'impact des changements et donc réduire les coûts d'évolution et de maintenance.

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.
- 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

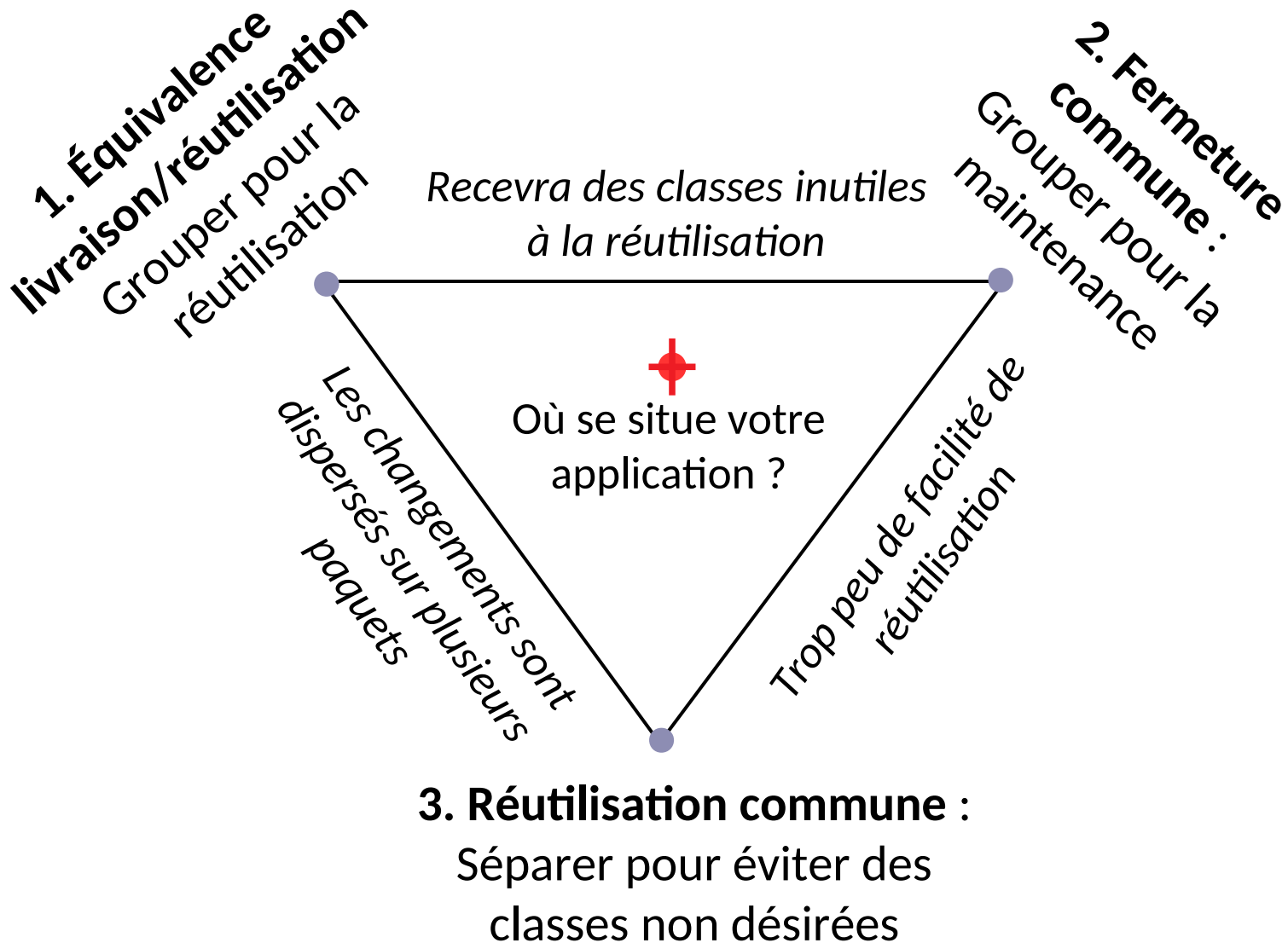
- Définition
 - Réutiliser une classe d'un paquet, c'est réutiliser le paquet entier.
 - Si vous réutilisez une des classes dans un paquet, vous les réutiliser toutes.
- Objectif
 - *Point de vue de la réutilisation.*
 - Les classes qui ont tendance à être utilisées ensemble appartiennent au même paquet.
 - ▶ p. ex. un 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.

Un critère d'exclusion de classes

- En fait, ce principe nous en dit plus sur quelles classes ne doivent pas être ensemble :
 - 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.
 - C'est le principe de ségrégation des interfaces appliqué aux paquets.

Conflits d'intérêts entre principes

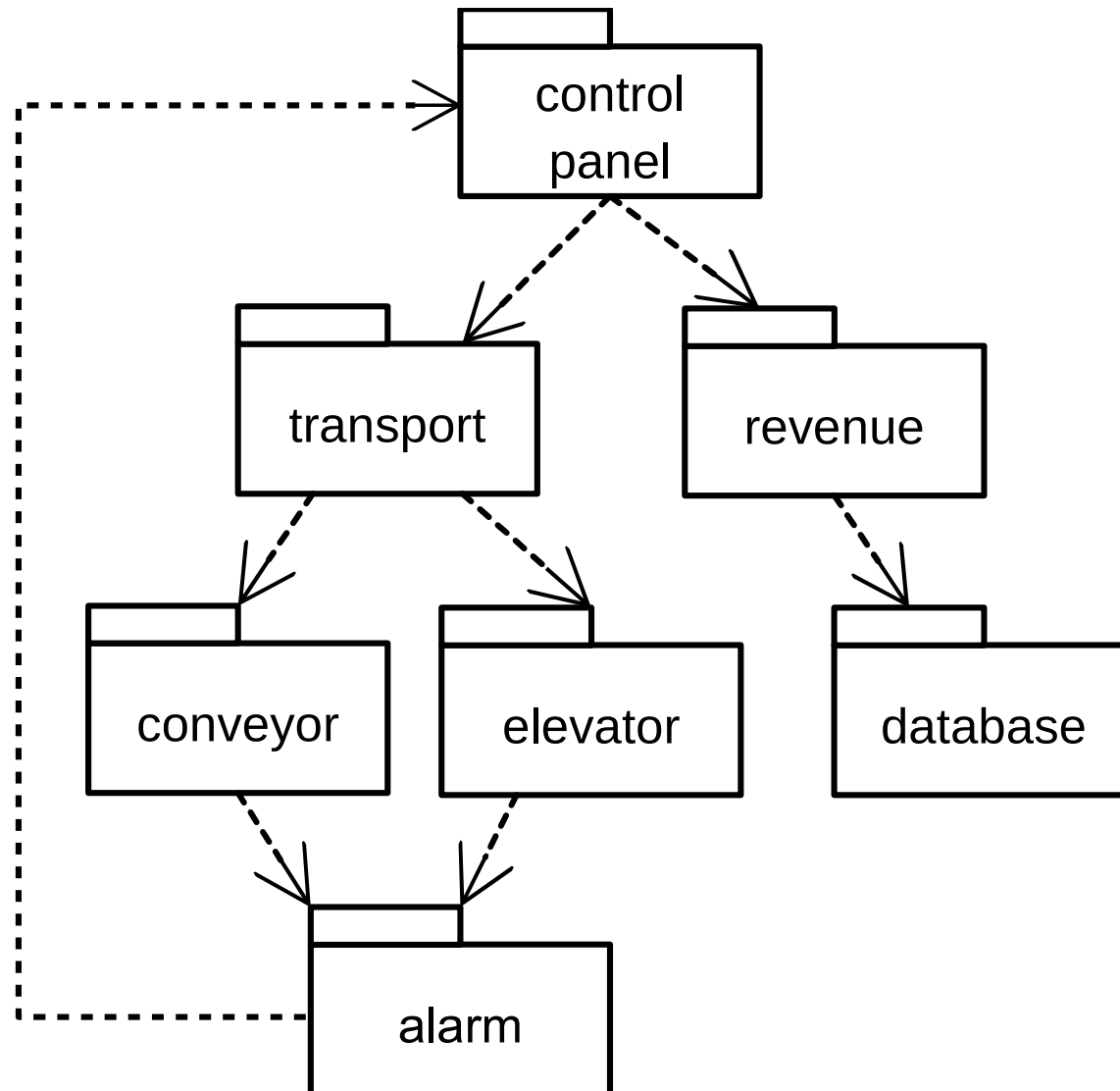
- Une application doit choisir ses priorités dans l'application de ces principes qui peuvent être contradictoires.



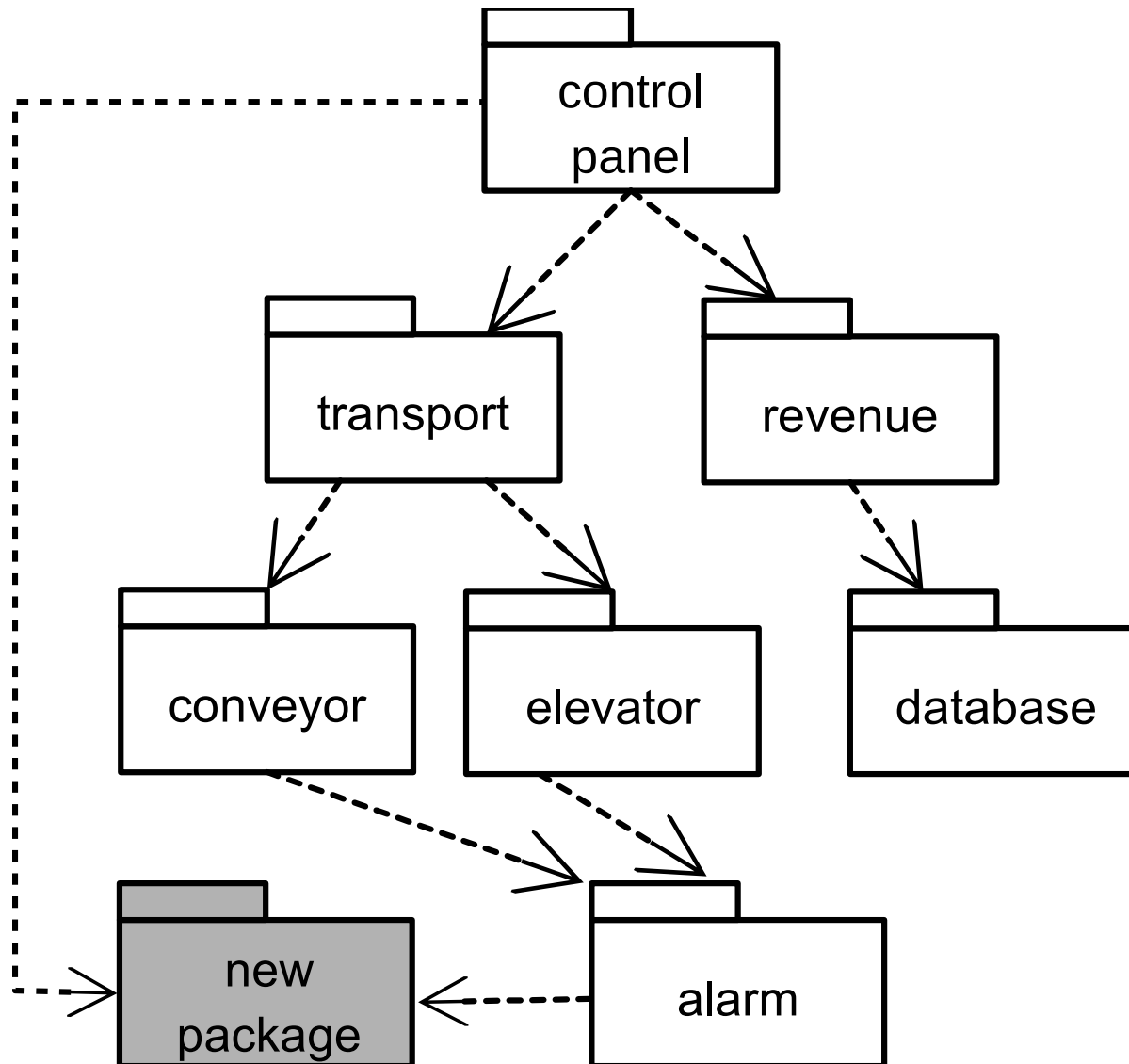
Principe 4. Dépendances acycliques

- 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.

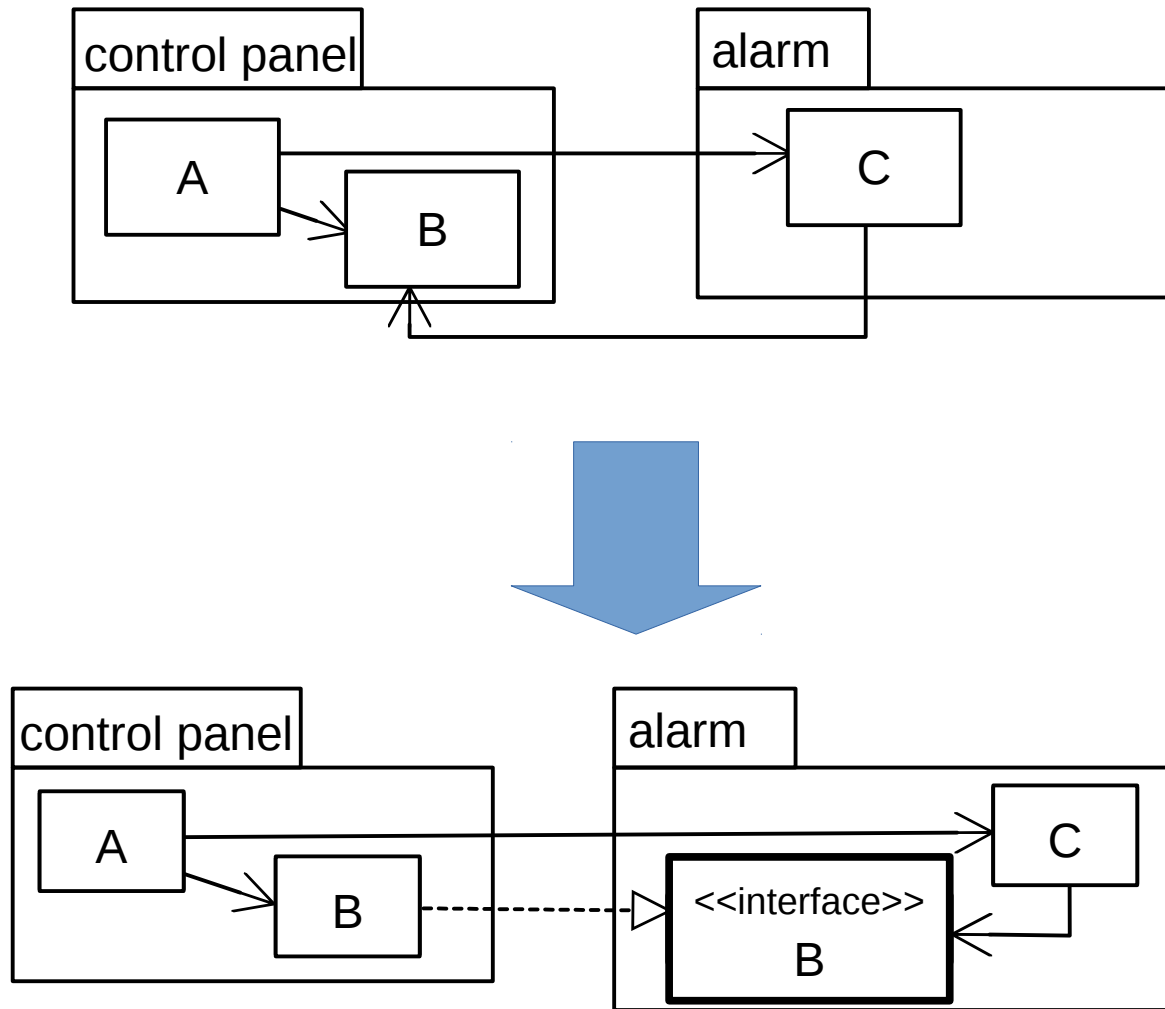
Les cycles ruinent l'harmonie



Casser les cycles par ajout d'un paquet de dépendances communes



Casser les cycles par inversion des dépendances



Principe 5. Relation dépendance / stabilité

■ 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.

■ Objectif

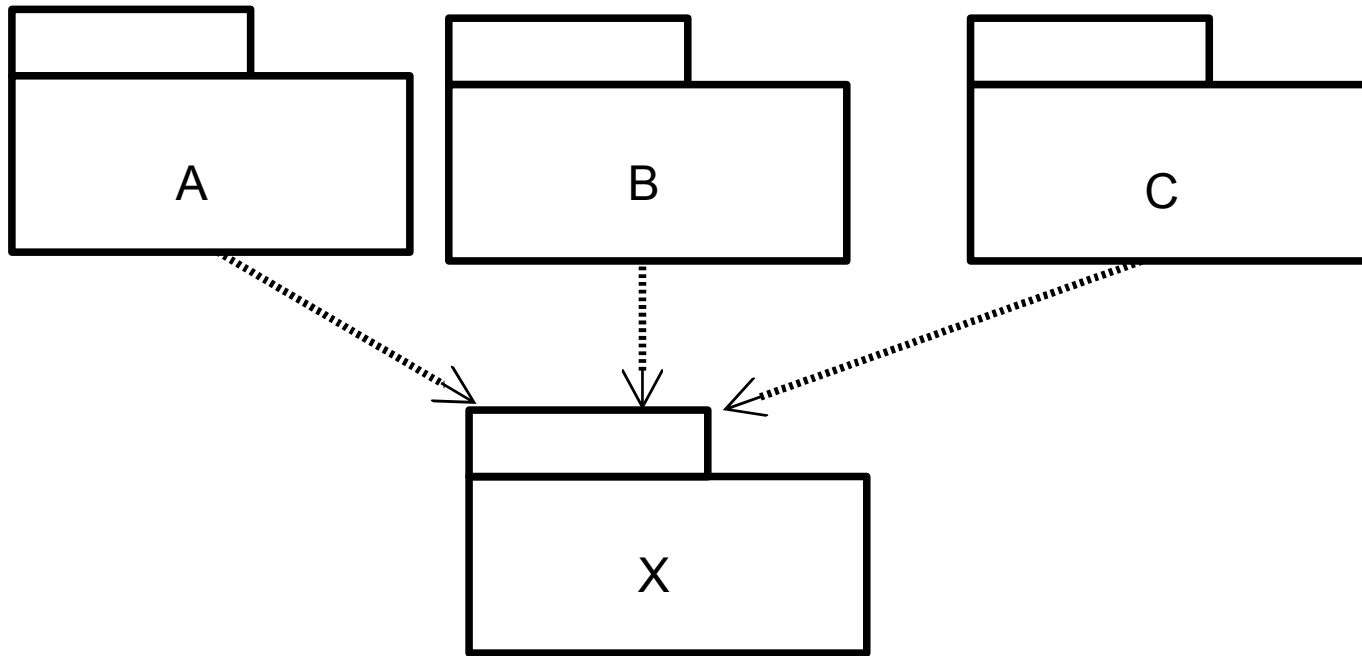
- Les paquets que nous voulons évolutifs ne devraient pas être dépendants de paquets difficiles à changer.
- La dépendance va dans le sens de la stabilité.

■ Motivation

- Limiter l'impact des changements les plus fréquents et maximiser la stabilité globale de l'application.

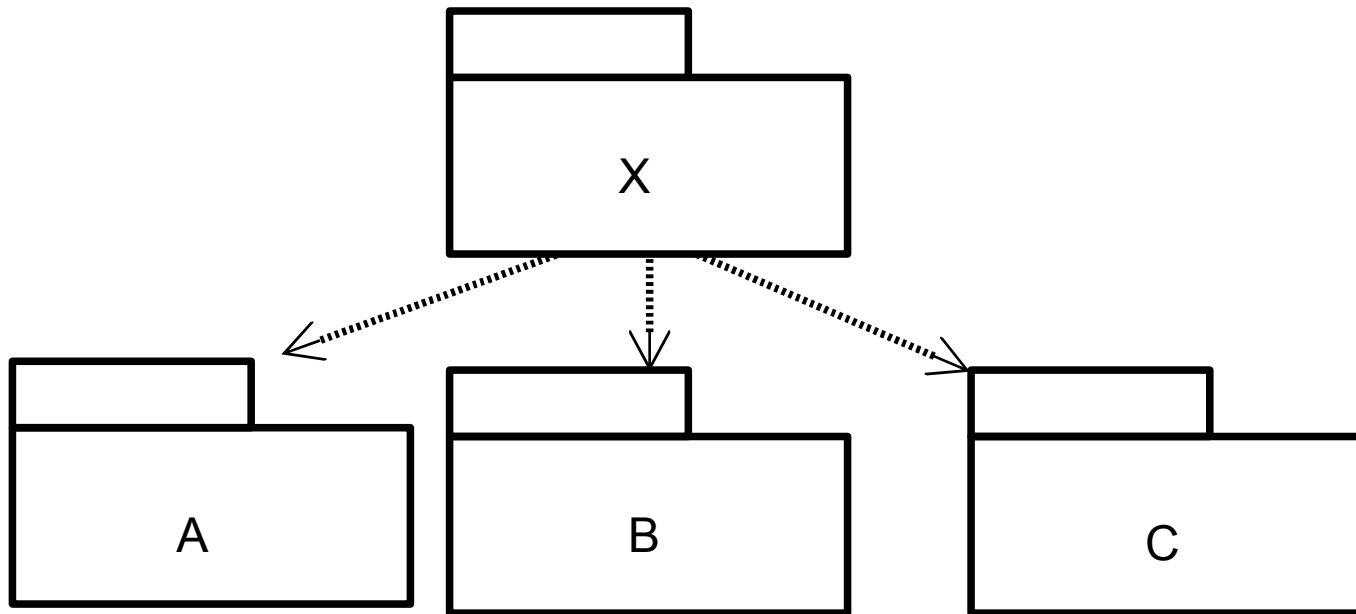
Paquet stable

- Un paquet avec beaucoup de dépendances afférentes est très stable (ie, difficile à changer).



Paquet instable

- Un paquet avec peu de dépendances afférentes est très instable (ie, facile à changer).



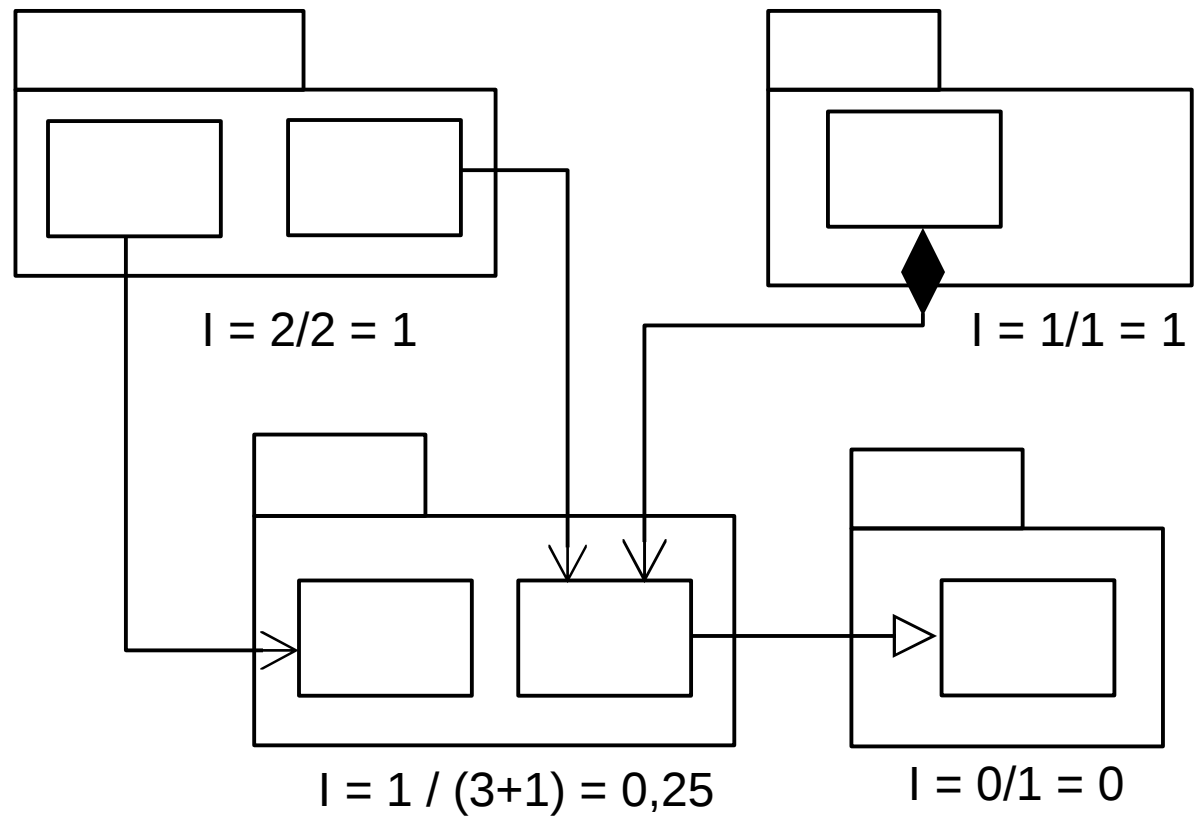
Une mesure d'instabilité

■ Instabilité $I = Ce / (Ca + Ce)$

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

■ Valeurs dans $[0, 1]$

- 0 : paquet stable
- 1 : paquet instable



Utilisation de la mesure d'instabilité

- La valeur d'instabilité d'un paquet doit être supérieure à la valeur d'instabilité des paquets dont il dépend.
- Tous les paquets ne peuvent pas être stables. S'ils sont tous stables le système ne serait plus évolutif.
- Pour résoudre le problème de la direction de stabilité :
 - Principe d'inversion des dépendances.
 - Création d'un paquet intermédiaire et déplacer les classes dont dépend la stabilité dans le paquet.

Principe 6. Stabilité des abstractions

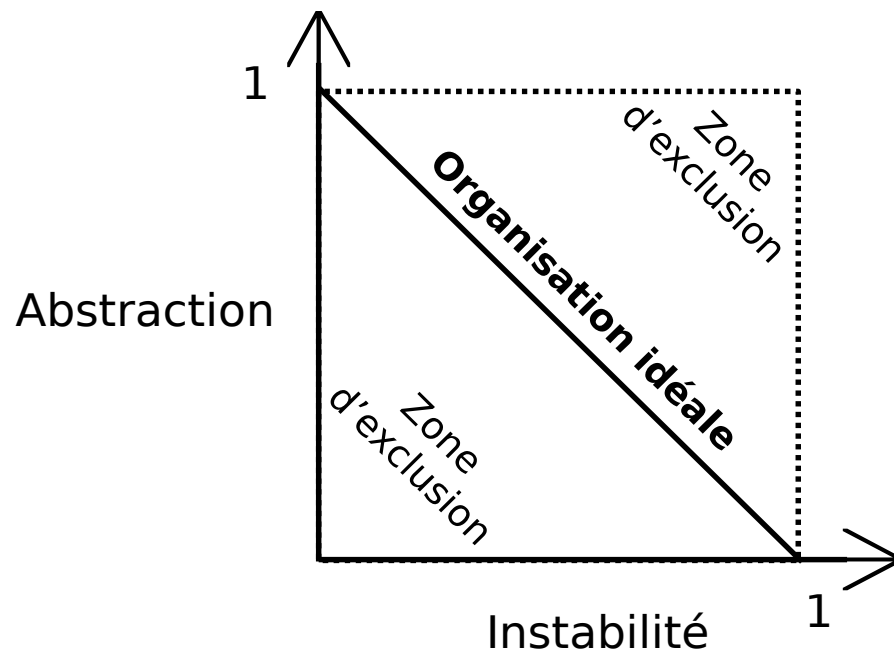
- Définition
 - Les paquets les plus stables doivent être les plus abstraits.
 - Les paquets instables doivent être concrets.
- Objectif
 - Un paquet stable devrait être abstrait de sorte que sa stabilité ne l'empêche pas d'être étendu.
 - Un paquet instable devrait être concret car son instabilité permet 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 métier. Elles forment ainsi l'architecture de l'application.

Une mesure d'abstraction

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

Relation instabilité / abstraction

- Mesure de qualité d'un paquet :
 - Calcul de la distance à l'organisation idéale entre instabilité (I) et abstraction (A):
 - ▶ Organisation idéale : la droite $A = 1 - I$.
 - ▶ **Distance à l'organisation idéale** = $|A + I - 1| \in [0,1]$.
 - ▶ Paquet ($I = 0, A = 0$) : non souhaitable.
 - ▶ Paquet ($I = 1, A = 1$) : inutile.



Méthode de conception en paquets

- 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 réutilisabilité, développabilité et maintenabilité.
- Démarche personnelle
 - Les premiers paquets sont inspirés de l'architecture.
 - Puis, les principes sont appliqués dès que se posent les questions de développabilité, réutilisabilité et maintenance.
- Finalement, 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**.

■ Conception

- En choisissant d'inclure des classes dans un paquet, nous devons choisir entre réutilisabilité, développabilité et maintenabilité.
- Ce choix conduit à de sans cesse remises en cause de la conception en paquets.
- Le partitionnement des classes en paquets ne peut pas intervenir 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.

- 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 (en supprimant donc tout cycle dans le graphe des 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.