



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

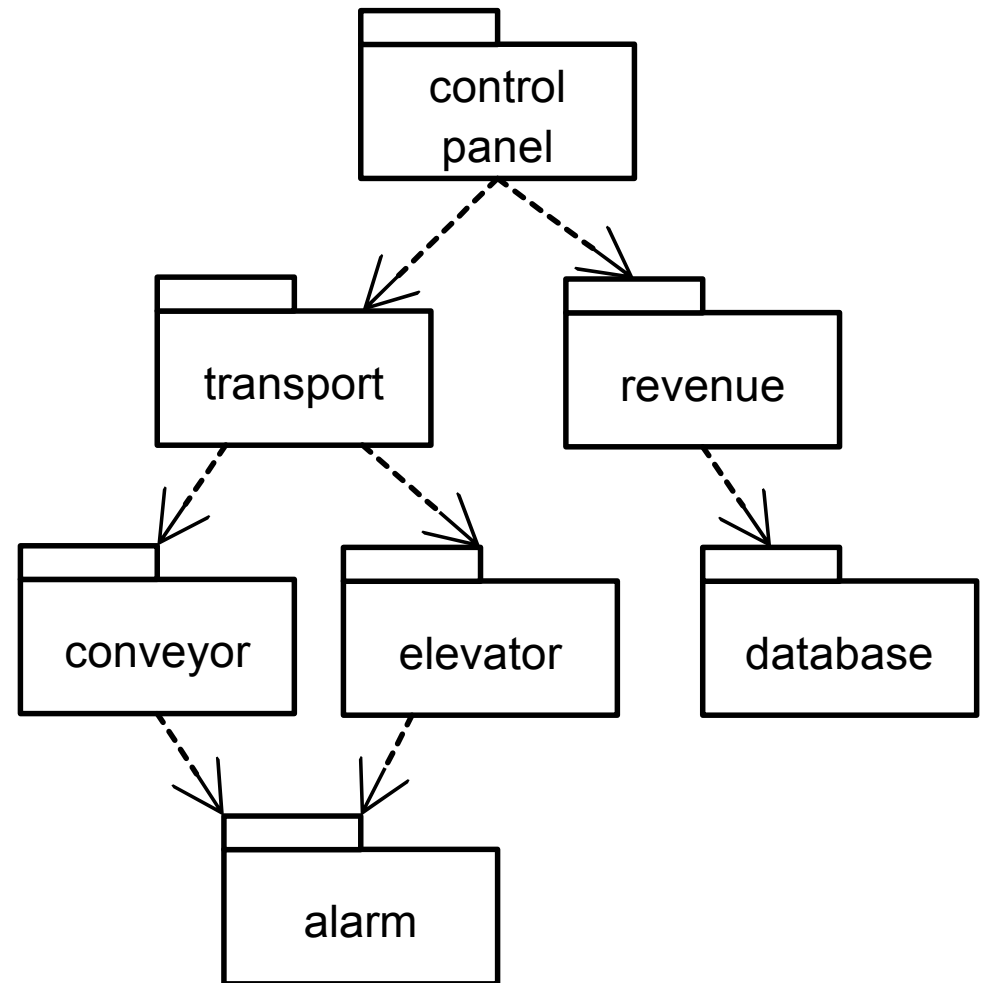
- Les paquets (*packages*) définissent un moyen d'organiser les sources et de structurer la conception.
 - Les paquets sont affectés à un ingénieur ou à une équipe d'ingénieurs de développement.
 - Les paquets qui sont hautement interdépendantes tendent à être rigides, non réutilisables et difficiles à maintenir. En même temps, les liens sont obligatoires pour pouvoir collaborer à la réalisation des fonctionnalités du logiciel.
- 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 (dossier en Java et C++),
 - espace de noms (classe en Java, namespace en C++),
 - sécurité des classes (public ou package en Java, 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 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 sont les relations 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 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 au début du projet ou en cours de projet ?
- Pour répondre à ces questions, il y a plusieurs principes qui gouvernent la création 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 de classes à part entière (numéro de version, archive). Le mainteneur du paquet doit être conscient de ses obligations envers les utilisateurs du paquet.
- Exemples de structuration en paquet. Mettre dans un même paquet :
 - ▶ Les classes Calendar, Date, Time.
 - ▶ Les classes Point, Line, Polygones.
 - ▶ Les classes permettant visualiser des statistiques sur des données sous différentes représentations graphiques (chart1D, chart2D, chart3D).
- Motivations
 - Éviter d'être dépendant des évolutions des paquets efférents.
 - Laisser la possibilité d'utiliser une version antérieure des paquets efférents.

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.
- Exemple de structuration en paquet :
 - Les classes CellDatabase (base de données) et CellEntity (table) devraient aller dans le même 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

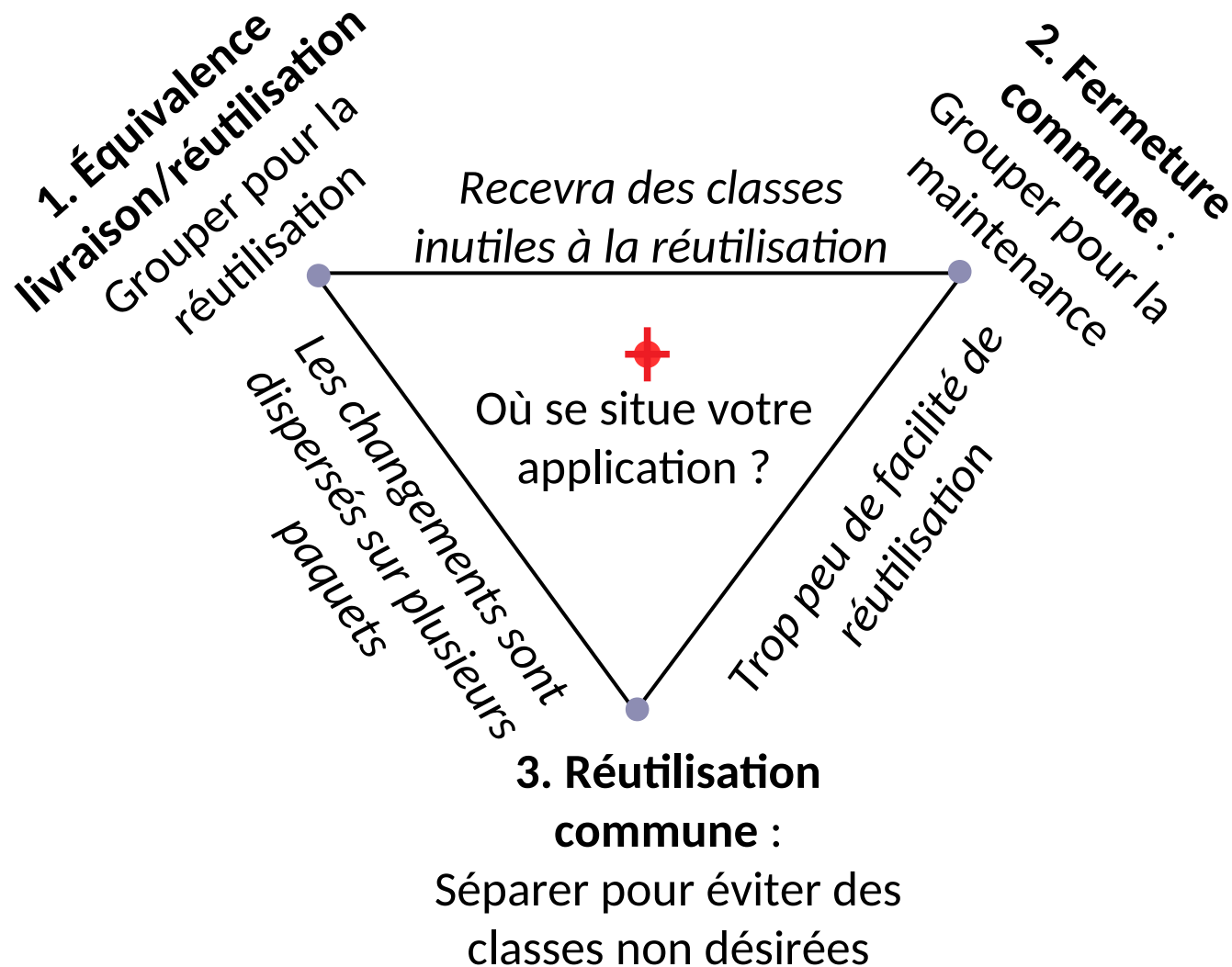
- Définition
 - Réutiliser une classe d'un paquet, c'est réutiliser le paquet entier.
 - Si vous réutilisez une des classes dans un paquet, vous les réutiliser toutes.
- Objectif
 - *Point de vue de la réutilisation.*
 - Les classes qui ont tendance à être utilisées ensemble appartiennent au même paquet.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

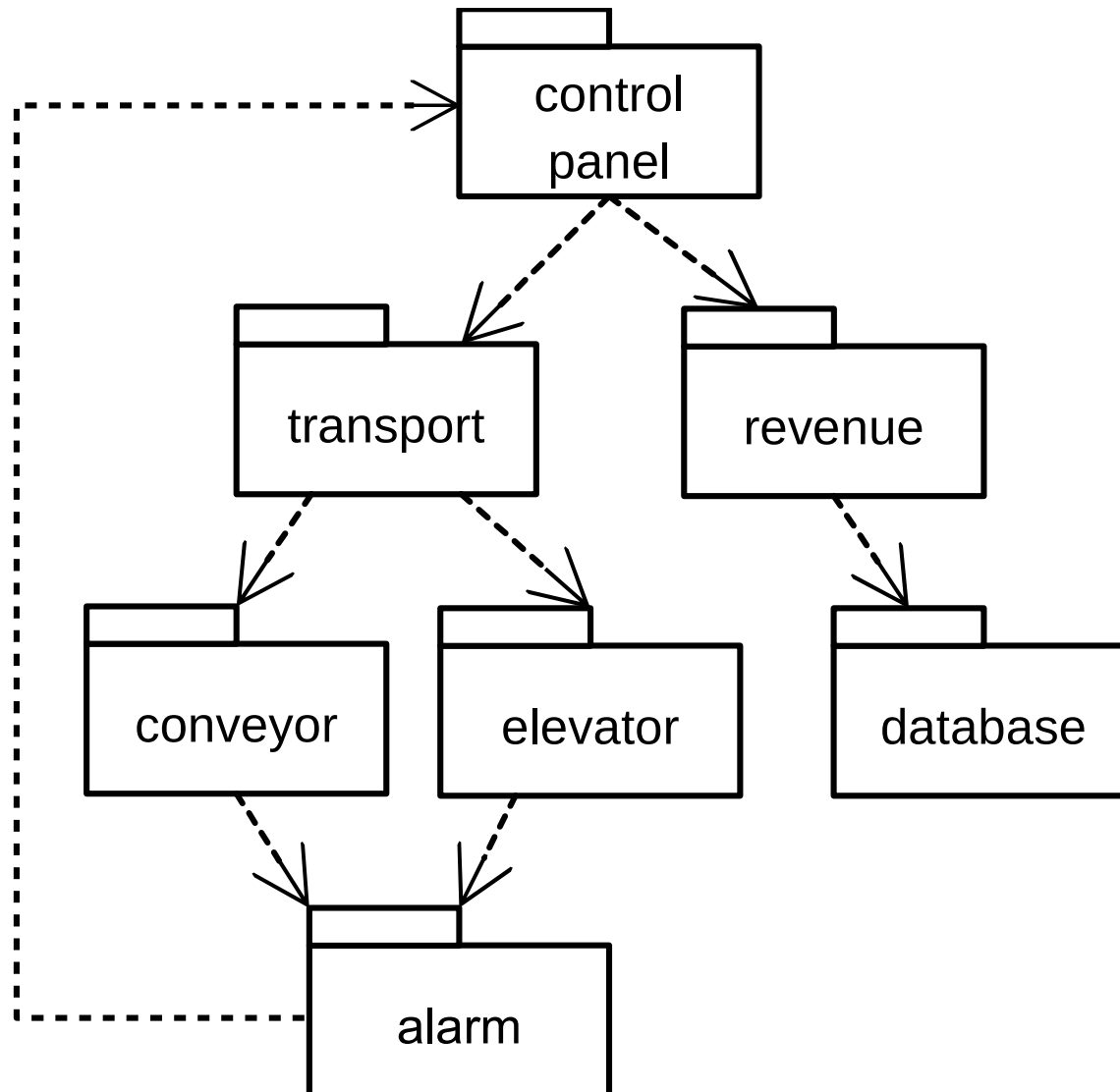
- Ces 3 principes ne sont pas forcément conciliables pour toutes les applications. Une application doit choisir ses priorités dans l'utilisation de ces principes qui peuvent être contradictoires.



Principe 4. Dépendances acycliques

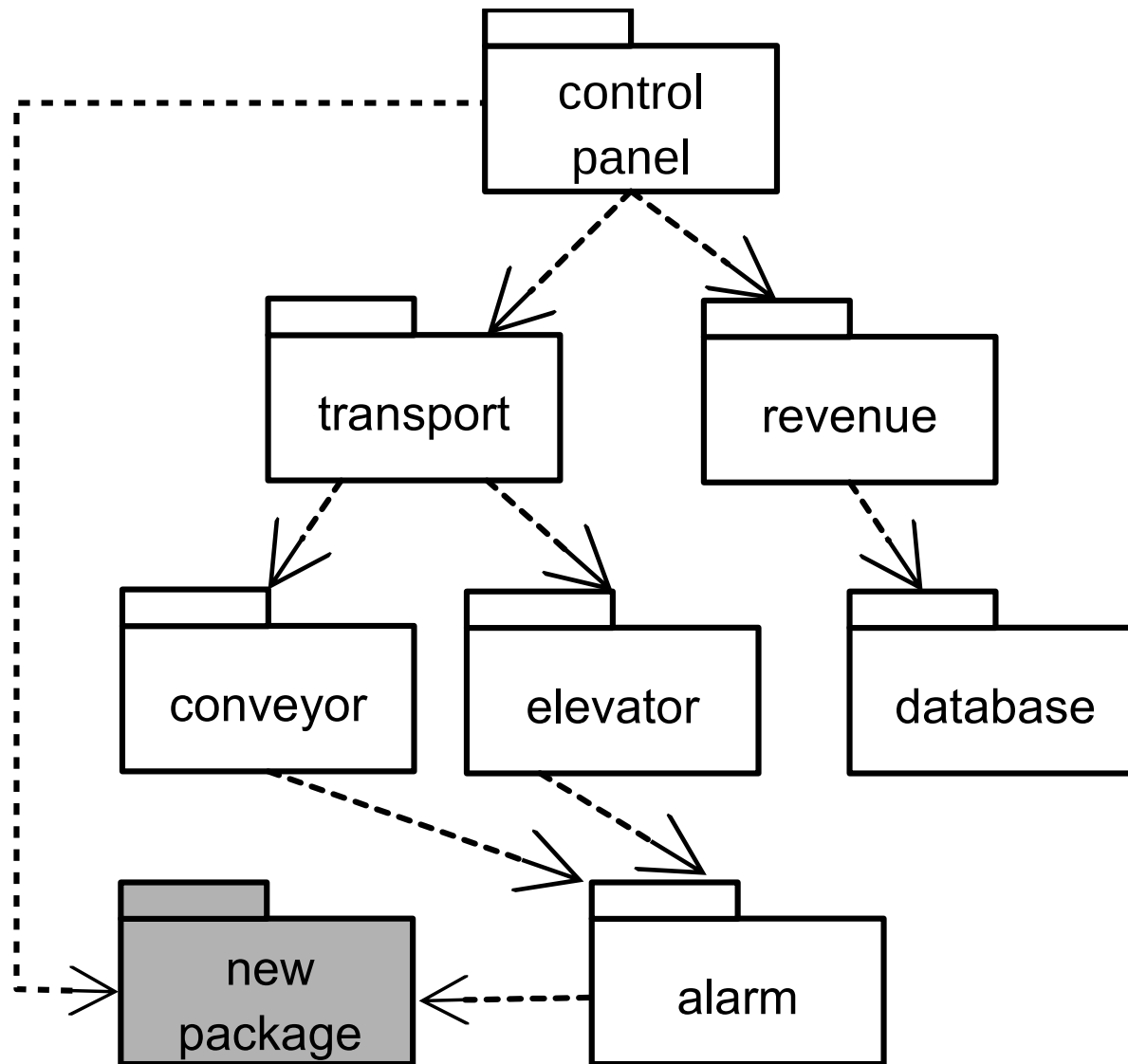
- Définition
 - Les dépendances entre paquets doivent former un graphe direct acyclique.
- Objectif
 - Supprimer les dépendances circulaires entre les paquets.
- Motivations
 - Augmenter la réutilisabilité.
 - Réduire les interférences entre les équipes de développement.
 - Permettre la testabilité.

Les cycles ruinent l'harmonie



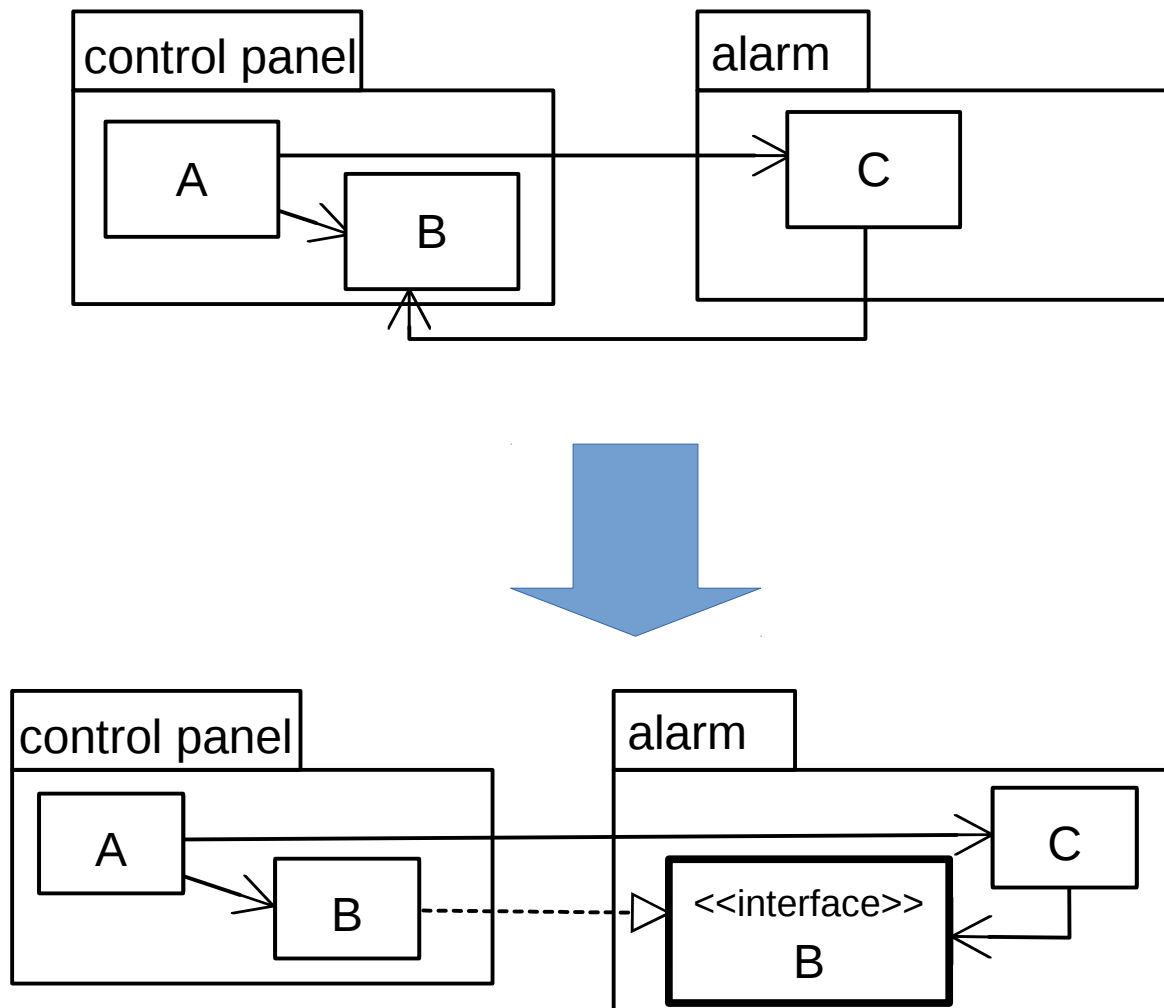
Casser les cycles par ajout d'un paquet de dépendances communes : solution 1

- Introduire un nouveau paquet avec les classes de « control panel » partagées par « alarm ».



Casser les cycles par inversion des dépendances : solution 2

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

■ Objectif

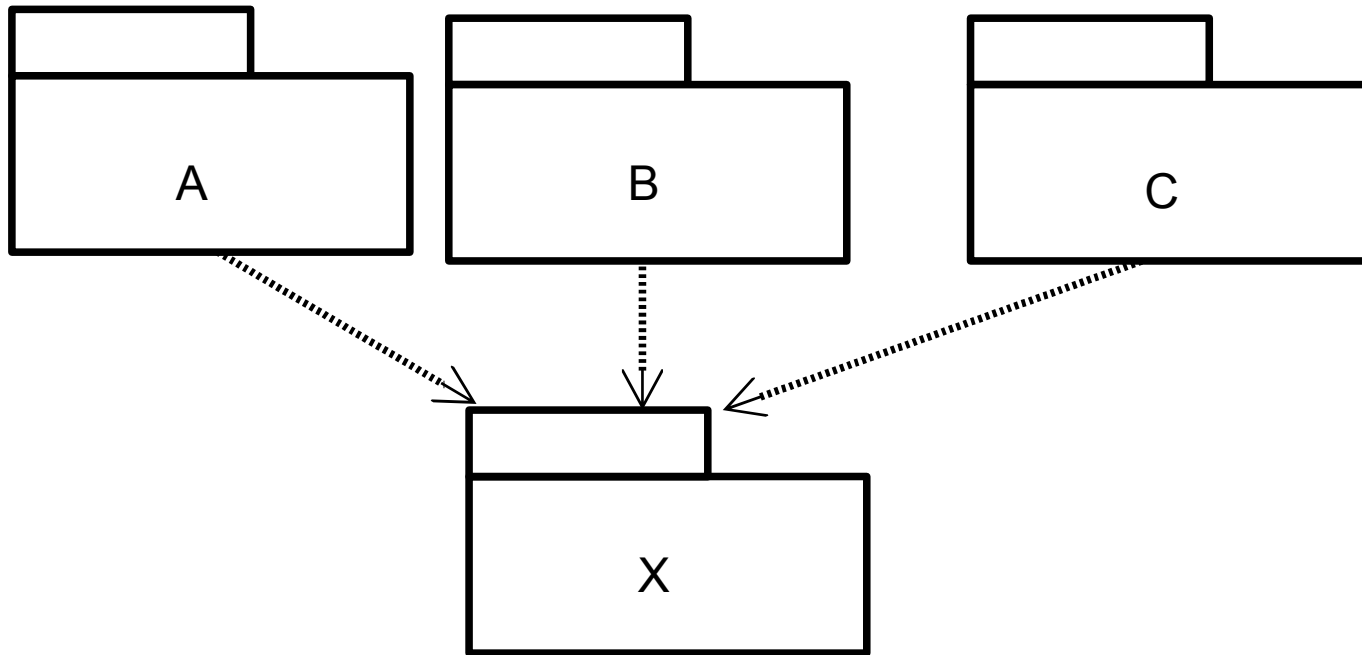
- Un paquet que l'on veut stable ne devrait pas dépendre de paquets qui sont susceptibles de changer au cours du cycle de vie. S'il le fait, les changements dans ses dépendances entraîneront des risques de corrompre le logiciel.
- La dépendance va dans le sens de la stabilité.

■ Motivation

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

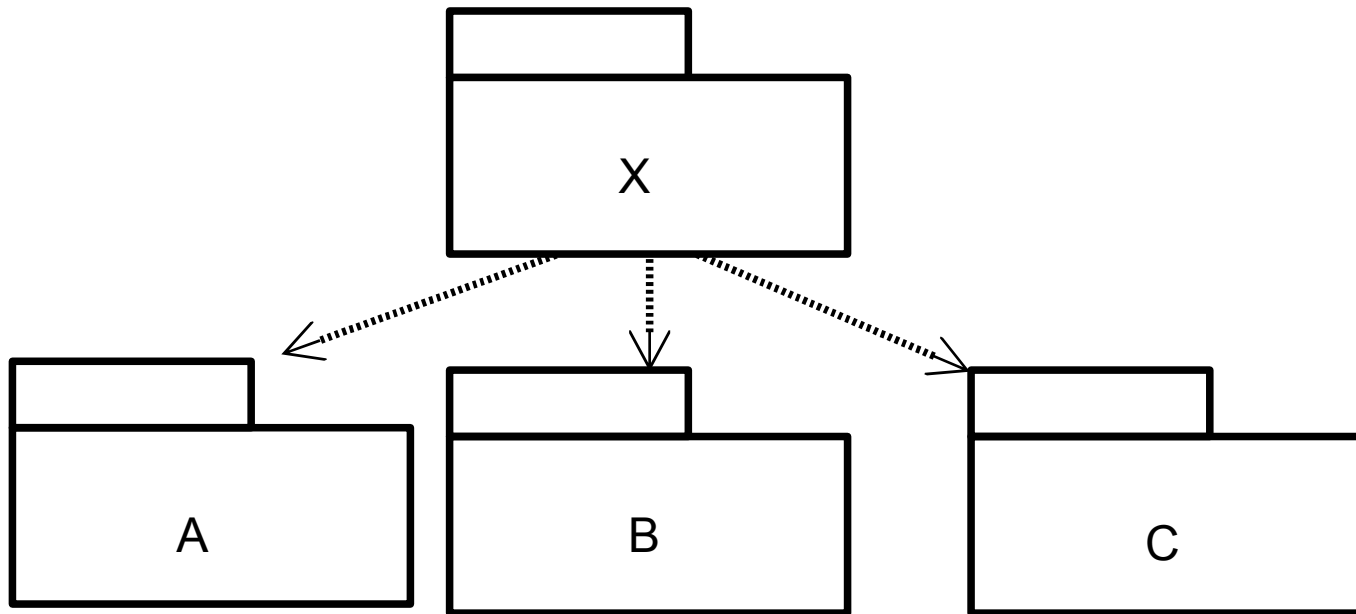
Paquet stable

- Un paquet avec beaucoup de dépendances afférentes doit être très stable (ie, parce que difficile à changer).



Paquet instable

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



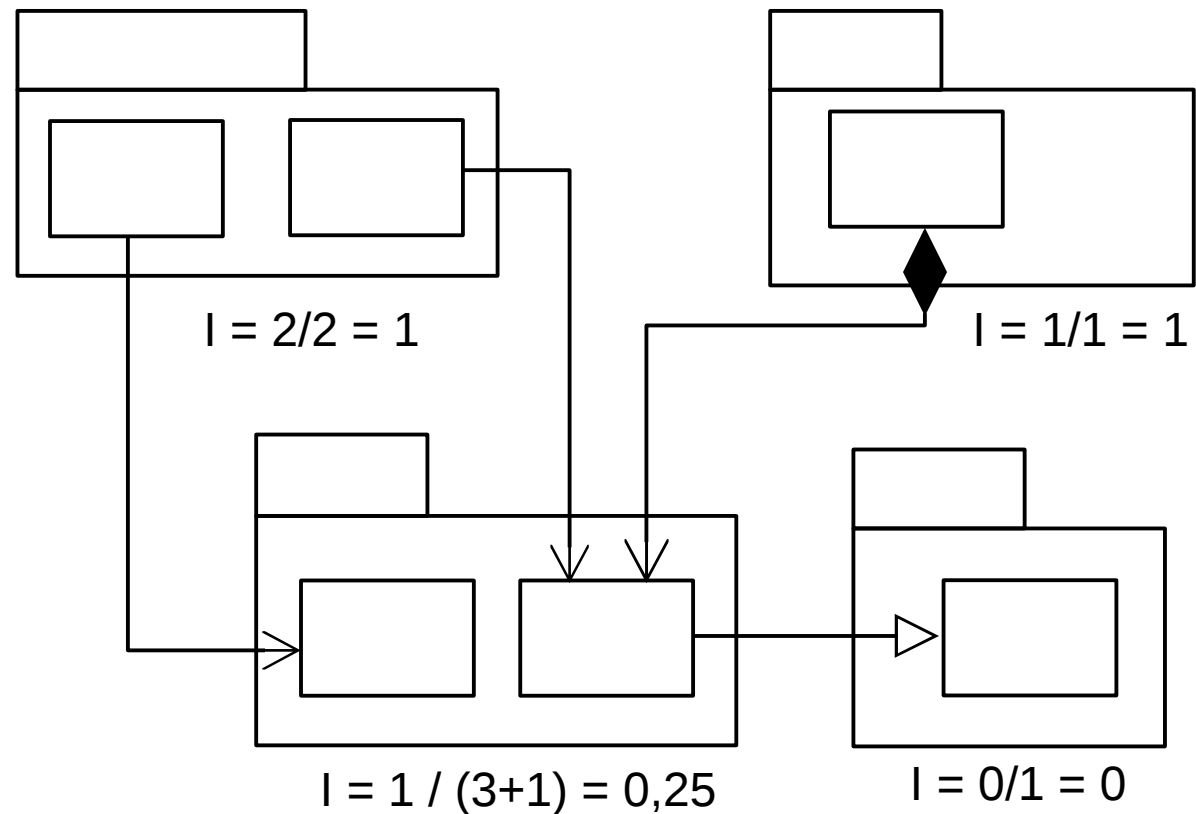
Une mesure d'instabilité

■ 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

- 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 l'architecture de l'application.

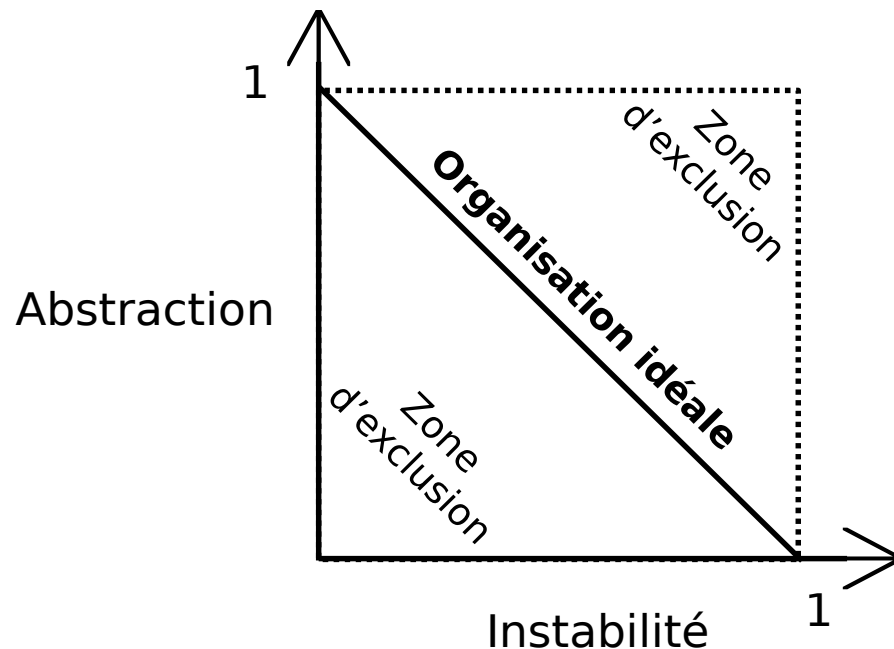
Le degré d'abstraction d'un package doit correspondre à son degré de stabilité.

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 $A = 1 - I$.
 - ▶ **Mesure = distance à l'organisation idéale = $|A + I - 1| \in [0,1]$.**
 - Paquet ($I = 0, A = 0$) : non souhaitable.
 - Paquet ($I = 1, A = 1$) : inutile.



- 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 réutilisabilité, développabilité et maintenabilité.
- Démarche
 - Les premiers paquets sont inspirés de l'architecture.
 - 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 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 des remises en cause périodique de la conception en paquets.
- Le partitionnement idéal des classes en paquets ne peut 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.

- 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.