Architectures Logicielles

Dr. Ghazala HCINI

Université de Gabès

2025-2026

Plan

Plan

Concepts de base

Génie logiciel

- Le Génie Logiciel (souvent appelé Software Engineering) est une discipline d'ingénierie industrielle. Son objectif est d'étudier et de codifier les méthodes de travail, les outils et les bonnes pratiques que les ingénieurs appliquent pour la conception et le développement de systèmes logiciels.
- Le Génie Logiciel se concentre sur l'établissement de procédures systématiques pour garantir que les systèmes complexes répondent aux attentes du client. L'objectif principal est de produire des logiciels qui possèdent des qualités essentielles telles qu'une haute fiabilité, des performances optimales, et un coût de maintenance réduit, tout en respectant les contraintes de budget et de délai de développement.

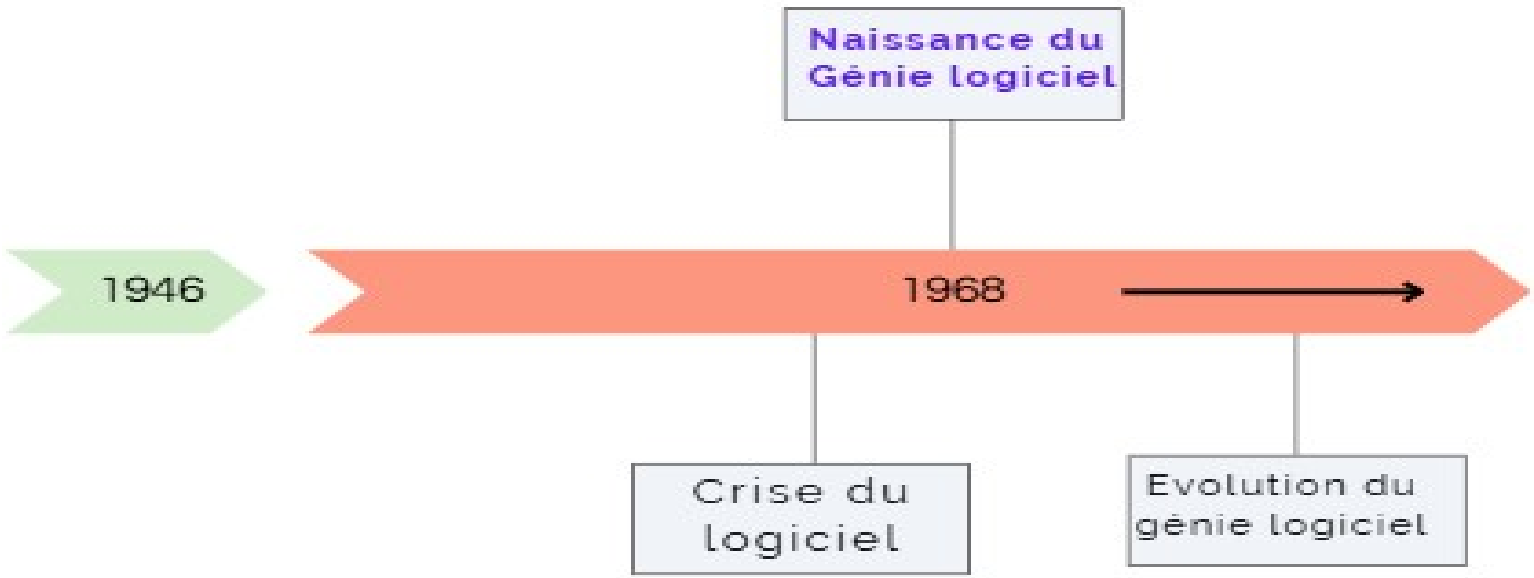
Concepts de base

Génie logiciel

Le champ du Génie Logiciel est vaste, couvrant de manière exhaustive les outils, techniques et méthodes utilisés par les équipes. Il organise l'intégralité des activités de développement et de maintenance, depuis la modélisation initiale jusqu'à la production des artefacts finaux et la gestion de l'ensemble du projet.

Un petit historique

Commençons par exposer brièvement l'origine et l'évolution du domaine de génie logiciel.



Un petit historique

De la crise du logiciel au GL

Problématique

- À la fin des années 60, avec l'avènement des machines de $3^{\text{ème}}$ génération est apparue la crise du logiciel.
- Les mécanismes utilisés pour les petits systèmes voyaient leur limite sur ces grosses applications.
- Les systèmes étaient peu fiable, difficiles à maintenir et avec de faible performance.

Un petit historique

De la crise du logiciel au GL

Besoins

- De nouvelles techniques et méthodes pour concevoir et maîtriser ces nouveau logiciels devenaient indispensable.
- Nécessité de passer d'une démarche artisanale à une discipline d'ingénieur.

Un petit historique

- Pour sortir de la crise, en **1968** sous le parrainage de l'OTAN (Organisation du Traité de l'Atlantique Nord), un groupe de chercheurs et de praticiens crée le génie logiciel (anglais, software engineering)
- Le génie logiciel est défini comme un domaine qui couvre les méthodes, la modélisation, les techniques, les outils, les activités, les biens livrables et la gestion de projets relatifs au développement et à la maintenance du logiciel.
- En s'appuyant sur une approche méthodique, l'équipe de développement accroit sa productivité et réalise des logiciels de qualité. De plus, il devient plus facile aux gestionnaires de « prédire » l'échéancier puisque les étapes sont connues.

Un petit historique

Depuis 1968, on assiste à une prolifération et à une évolution continue :

- des méthodes d'analyse et des notations pour spécifier les fonctionnalités d'un logiciel ;
- des techniques de conception et des notations pour exprimer la solution ;
- des techniques et des langages de programmation ;
- des procédures de validation et de vérification pour assurer la qualité du logiciel ;
- des procédures de maintenance.

Génie logiciel

Malgré les avancées, le domaine de la recherche en Génie Logiciel est demeuré hautement pertinent, conduisant à l'émergence de nouvelles approches fondamentales dans les années 90, notamment :

- Le développement des Systèmes Distribués,
- L'essor des Modèles de Composants (visant la réutilisation),
- L'établissement d'une norme de modélisation unique à travers UML (Unified Modeling Language),
- L'Architecture Logicielle.

Nouveaux thèmes de recherche

De nombreux thèmes de recherche se sont alors développés et ont contribué à l'élaboration de ce nouveaux domaine :

- La spécification et analyse des besoins
- La conception et la modélisation
- Les langages de programmation et les environnements de développement
- La validation et le test
- L'ingénierie inverse et la maintenance
- La gestion de projets

Architecture Logicielle

L'architecture logicielle est un domaine du GL qui a reçu une attention particulière ces dernières années. C'est au cours de la décennie 1970–80 que les grands principes architecturaux furent élaborés.
La conception d'une bonne architecture logicielle peut amener à un produit :

- Qui répond aux besoins des clients
- Qui peut être modifié facilement pour rajouter de nouvelles fonctionnalités.

Architecture Logicielle

L'architecture logicielle consiste à :

- Décrire l'organisation générale d'un système et sa décomposition en sous-systèmes ou composants
- Déterminer les interfaces entre les sous-systèmes
- Décrire les interactions et le flot de contrôle entre les sous-systèmes
- Décrire également les composants utilisés pour implanter les fonctionnalités des sous-systèmes.

Pourquoi développer une architecture logicielle ?

- Pour permettre à tous de mieux comprendre le système
- Pour permettre aux développeurs de travailler sur des parties individuelles du système en isolation
- Pour préparer les extensions du système.

Utilité d'une architecture logicielle

- **Compréhension** : facilite la compréhension des grands systèmes complexes en donnant une vue de haut-niveau de leur structure et de leurs contraintes. Les motivations des choix de conception sont ainsi mis en évidence
- **Réutilisation** : favorise l'identification des éléments réutilisables, parties de conception
- **Évolution** : met en évidence les points où un système peut être modifié et étendu.
- **Analyse** : offre une base pour l'analyse plus approfondie de la conception du logiciel, analyse de la cohérence, test de conformité, analyse des dépendances,
- **Gestion** : la gestion générale.

Caractéristiques architecturales

Performance

Représente la performance par rapport à la quantité de ressources utilisées dans des conditions données.

Question :

Citez des exemples de “performance ”

- **Comportement temporel** : temps de réponse, débit
- **Utilisation des ressources** : quantité et types de ressources utilisées
- **Capacité** : limites maximales

Caractéristiques architecturales

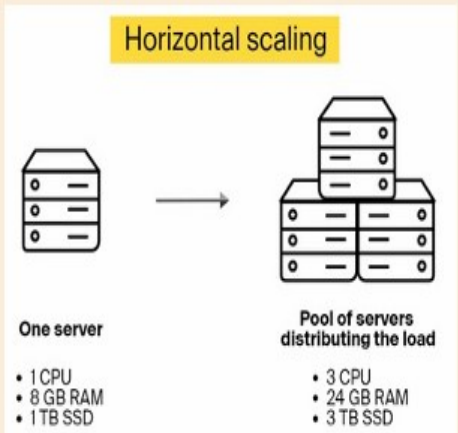
Scalabilité

La scalabilité consiste à gérer un grand nombre d'utilisateur sans avoir une dégradation sérieuse des performance.

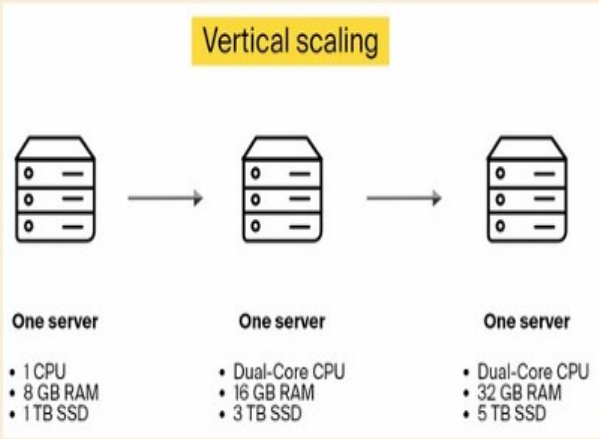
Question :

Comment pourrait-on faire ?

Ajouter nœuds supplémentaires (scale out)



Soit en renforçant le matériel (scale up)



Caractéristiques architecturales

Elasticité

L'élasticité est le degré auquel un système est capable de s'adapter aux demandes en approvisionnant et désapprovisionnant des ressources de manière automatique, de telle façon à ce que les ressources fournies soient conformes à la demande du système.
L'élasticité permet de saisir les aspects essentiels de l'adaptation, à savoir :

- La vitesse : correspond au temps nécessaire pour scale up.
- La précision : est défini comme l'écart absolu entre la quantité actuelle de ressources allouées et la demande réelle de ressources.

Caractéristiques architecturales

Interopérabilité

Est la capacité d'un système d'entreprise (ou de tout système informatique général) à utiliser les informations et les fonctionnalités d'un autre système.

- L'interopérabilité est la clé de la construction de systèmes d'entreprise avec des services mixtes.

Caractéristiques architecturales

Couplage faible, Forte cohésion

Concevoir des composants autonomes, indépendants et dotés d'un objectif unique et bien défini.

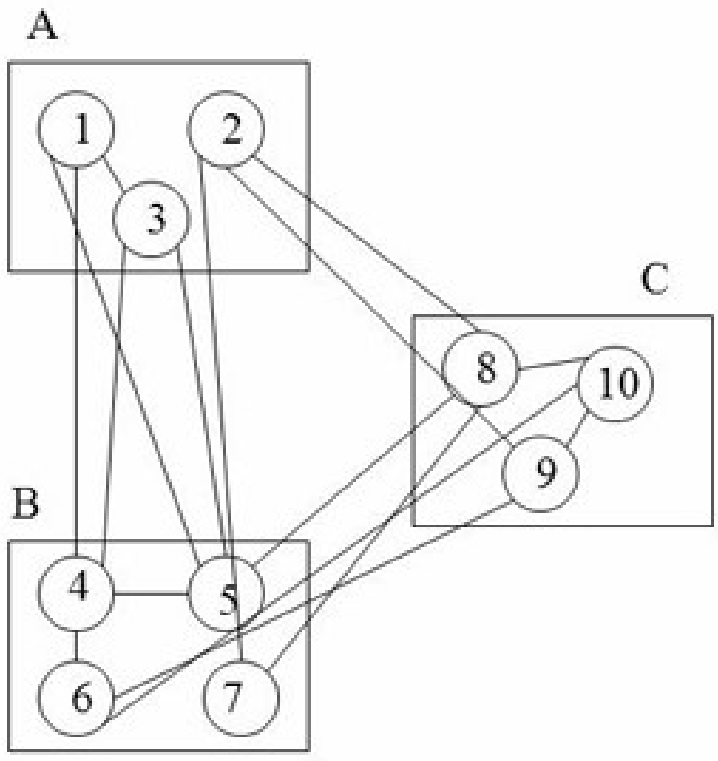
Question :

Que cela peut-il signifier (couplage et cohesion) ?

- Couplage faible : Les modules doivent être aussi indépendants que possible des autres modules, de sorte que les modifications apportées au module n'aient pas d'impact important sur les autres modules.
- Forte cohésion : La cohésion fait référence à la manière dont les éléments d'un module s'intègrent les uns aux autres. Les codes apparentés doivent être proches les uns des autres afin d'assurer une grande cohésion.

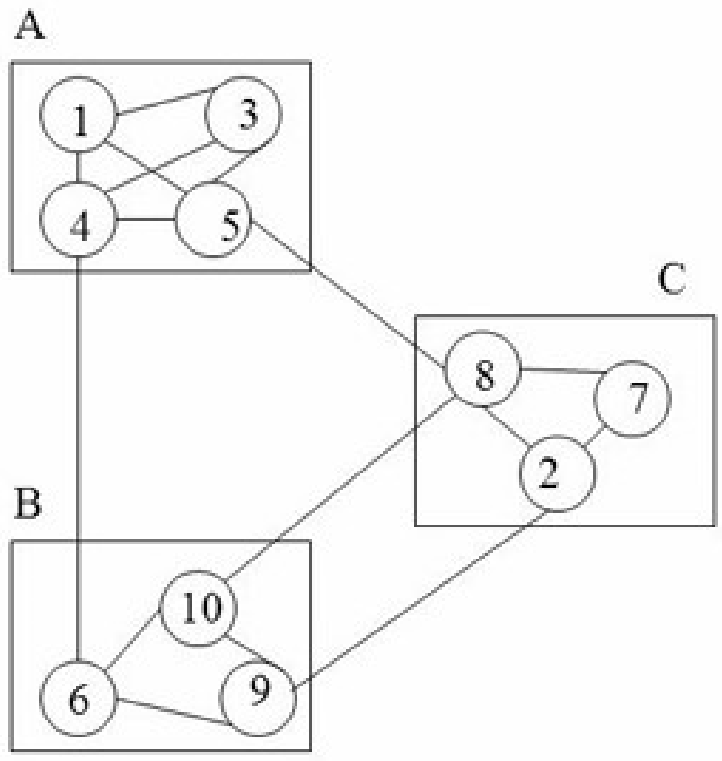
Caractéristiques architecturales

Couplage faible, Forte cohésion



Mauvaise modularisation

Faible cohésion, couplage élevé

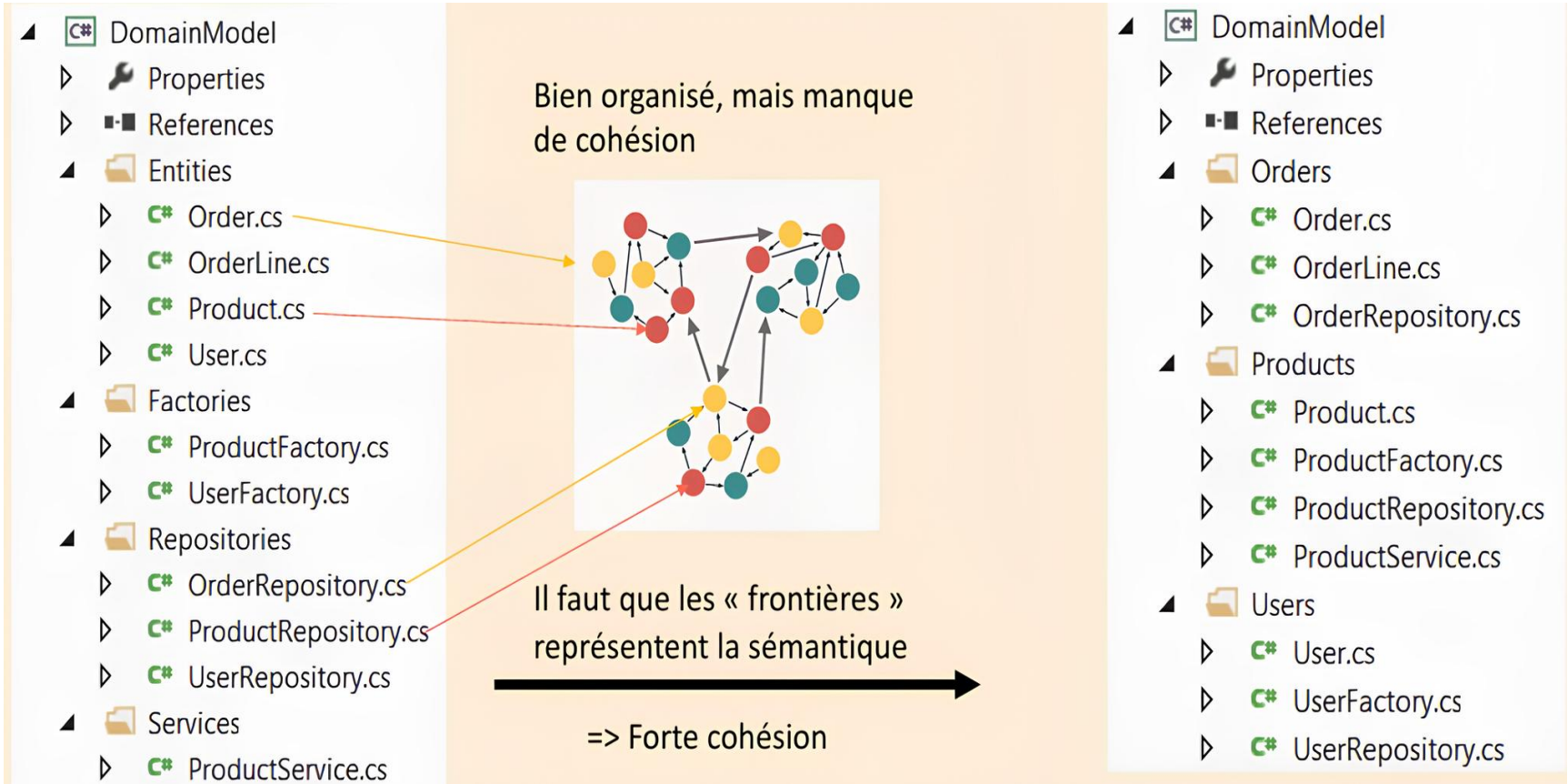


Bonne modularisation

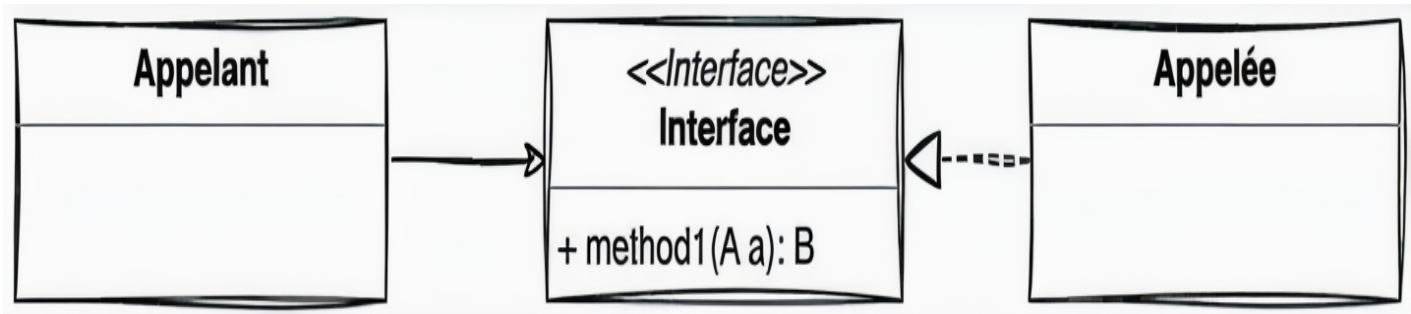
Haute cohésion, faible couplage

Caractéristiques architecturales

Couplage faible, Forte cohésion (package)

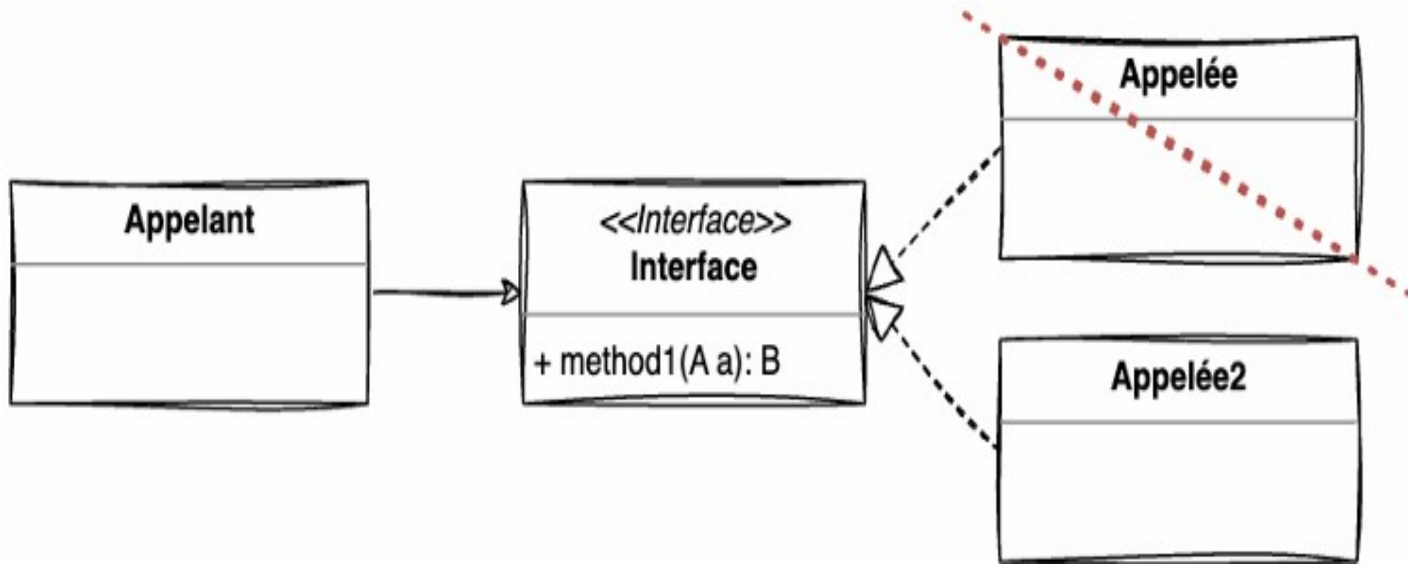


Interface



- Elle définit les méthodes (i.g. service) qui vont pouvoir être appelées
- Elle définit les informations que doit fournir l'appelant (i.g paramètre de la méthode)
- Elle définit les informations que l'appelant va obtenir (i.g type de retour).

Changer l'implémentation



Note : Cycle **Build** → **Test** → **Deploy**

Changer l'implémentation

Interface

- L'appelant utilise une interface qui définit une méthode commune.
- L'interface agit comme un contrat que toutes les classes concrètes doivent respecter.
- Les classes concrètes (Appelée, Appelée2) implémentent cette interface en fournissant des comportements spécifiques.
- L'appelant ne dépend que de l'interface, pas des classes concrètes, ce qui permet de découpler la logique métier des détails d'implémentation.

Avantages

- La modification ou le remplacement d'une implémentation peut être faite sans impacter le code de l'appelant, ce qui facilite la maintenance et l'évolution continue du logiciel.
- Ce modèle est parfaitement adapté au cycle **Build** $\rightarrow$ **Test** $\rightarrow$ **Deploy**.

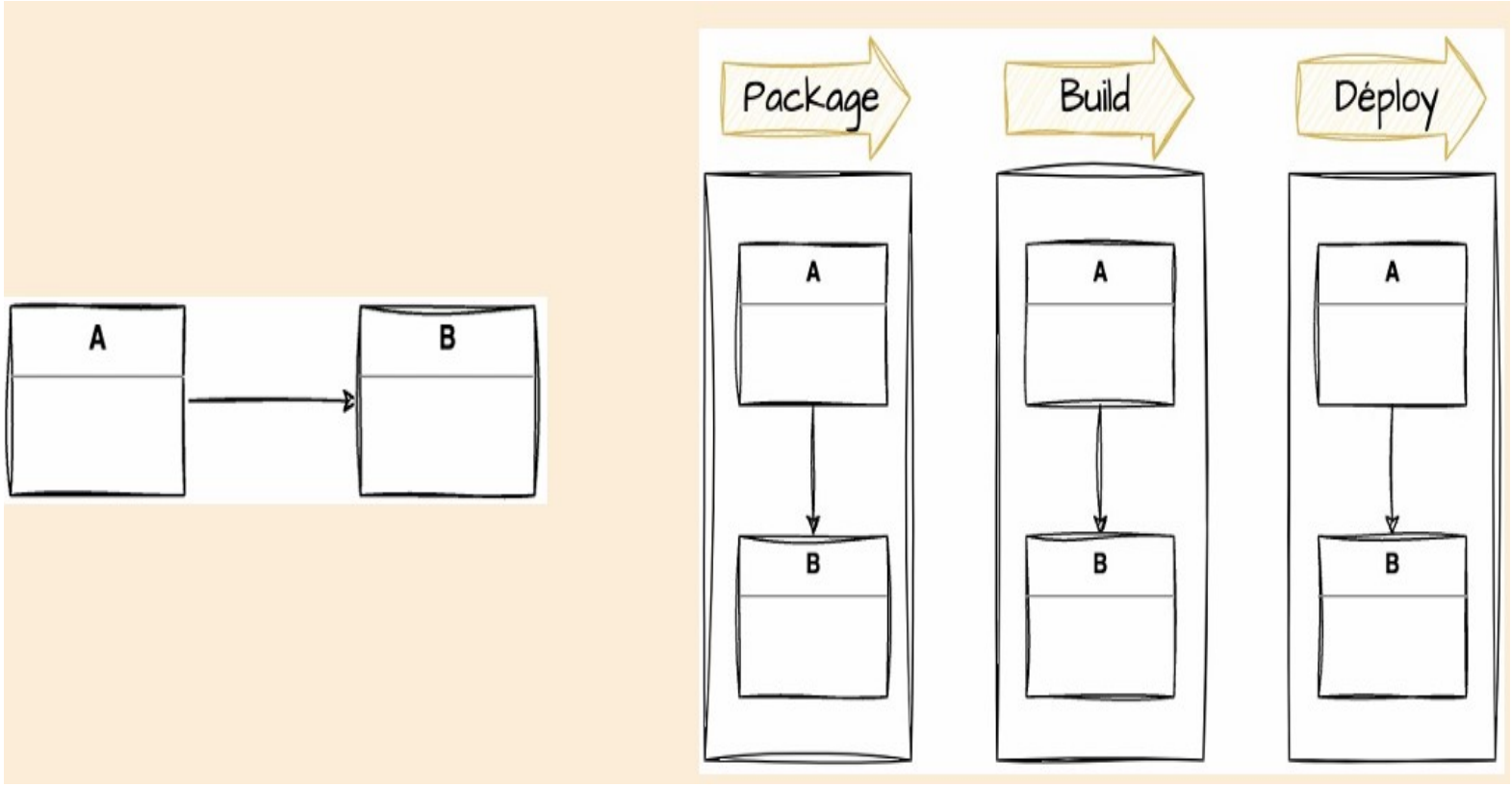
Concepts fondamentaux

Inversion de dépendance

Question :

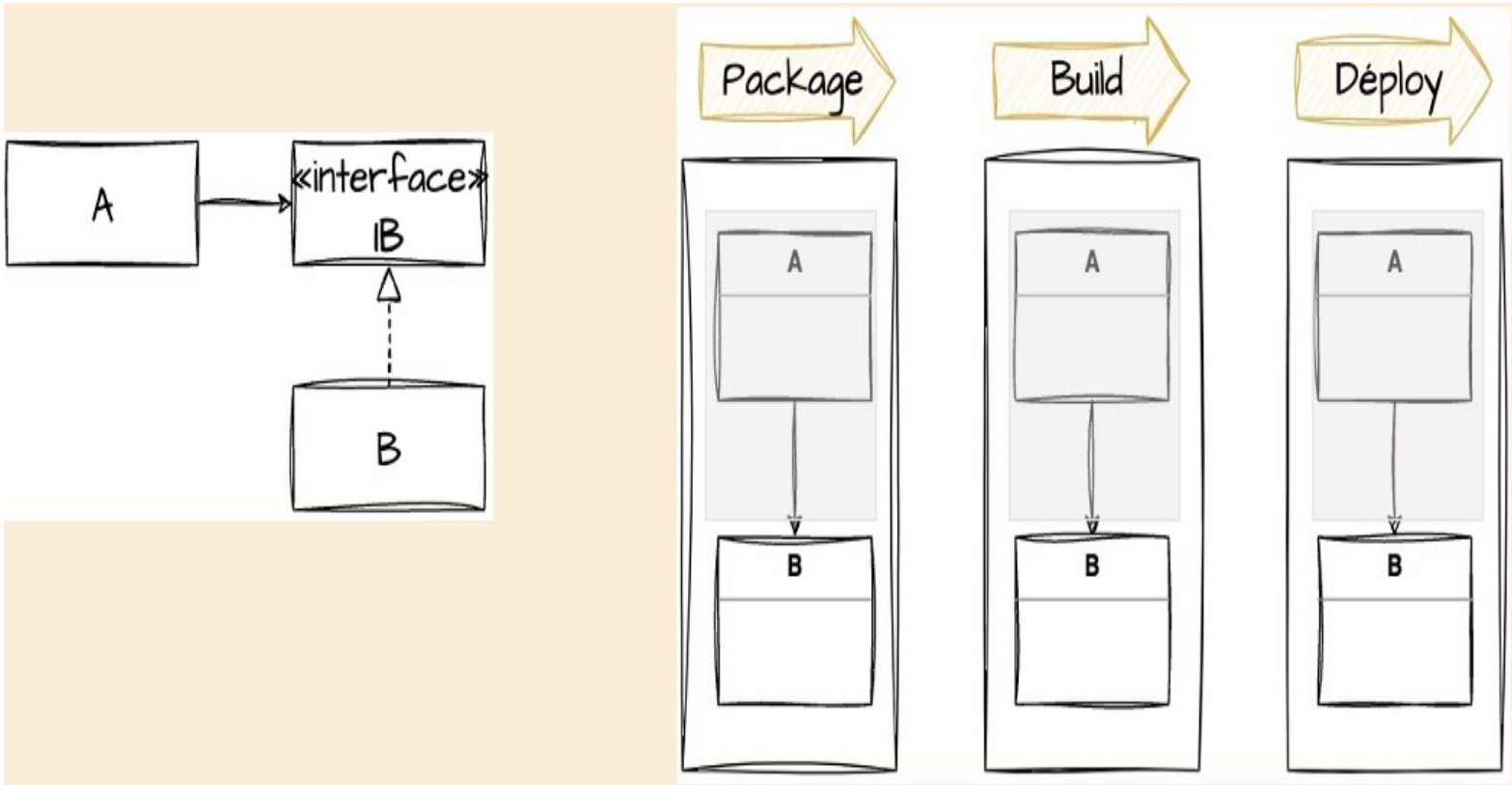
Que pouvez me dire de ce principe ?

Sans Inversion de dépendance



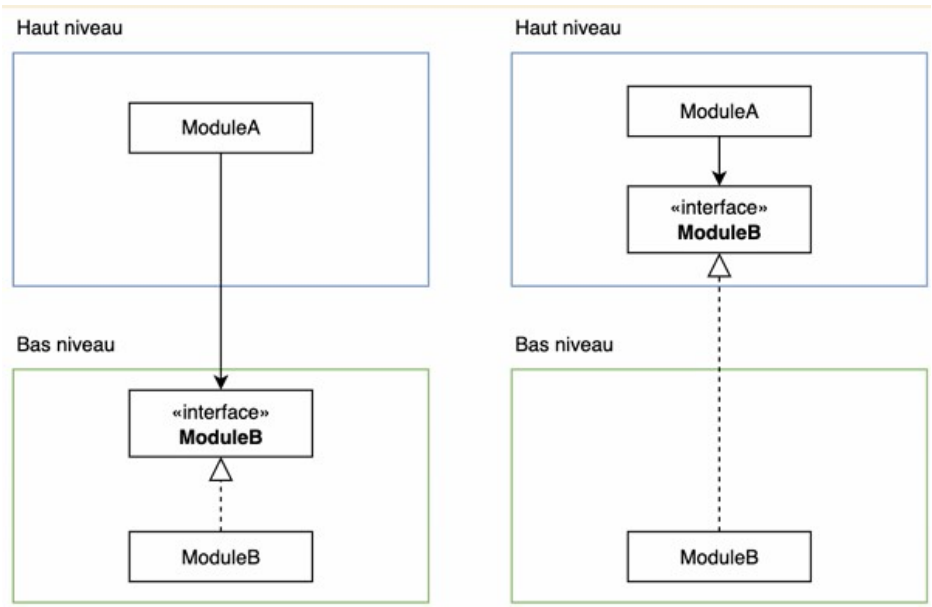
Inversion de dépendance

Avec Inversion de dépendance



- A ne dépend plus de B mais de son interface IB
- Donc si B est modifiée alors seulement B sera packagée, build et déployée chez le client.

Inversion de dépendance



À droite :

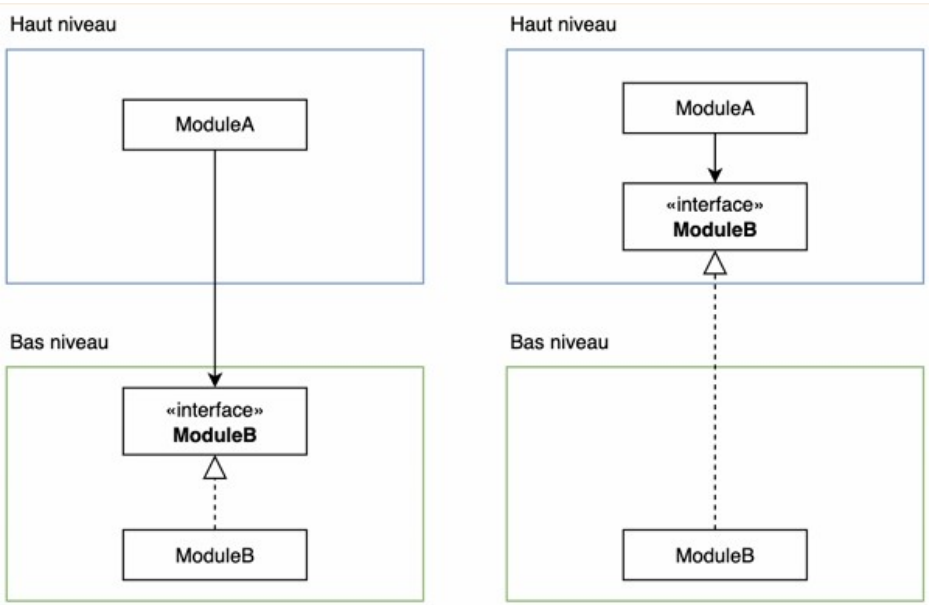
- C'est le module de bas niveau qui définit comment on interagit avec lui
- $\Rightarrow$ c'est le module de bas niveau qui dirige

À gauche :

- On change « la phrase »
- Le module de haut niveau dit les fonctionnalités requises par chaque implémentation
- $\Rightarrow$ c'est le module de haut niveau qui dirige

Inversion de dépendance

Amélioration ; MAIS . . .

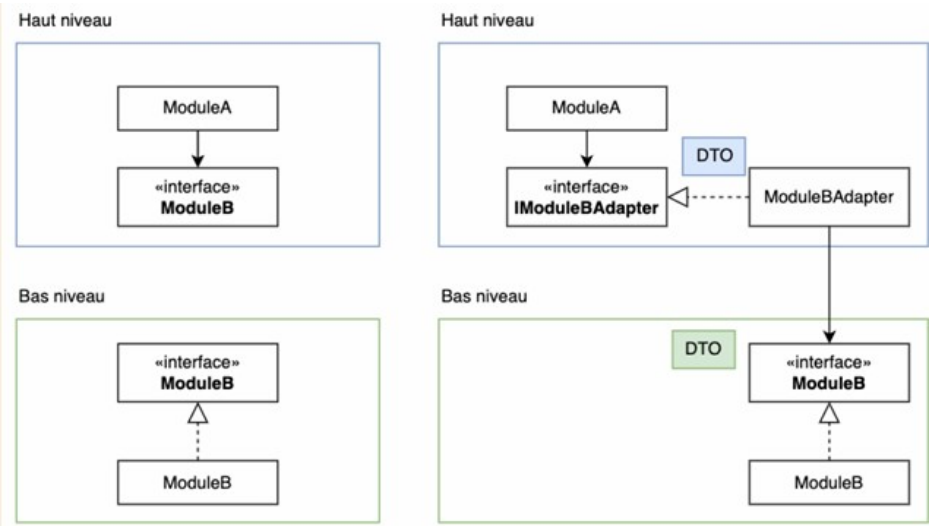


Avec cette conception :

- Il est impossible de réutiliser le module de bas niveau dans un autre contexte
- En effet, c'est le module de haut niveau qui dirige et le module de bas niveau subit l'interface.

Inversion de dépendance

Amélioration ; DONC . . .



On rajoute une interface :

- Les deux modules peuvent maintenant être réutilisés dans n'importe quel contexte

Faire le lien :

- Pour faire le lien entre les deux modules on utilise le patron adaptateur et la notion de DTO.

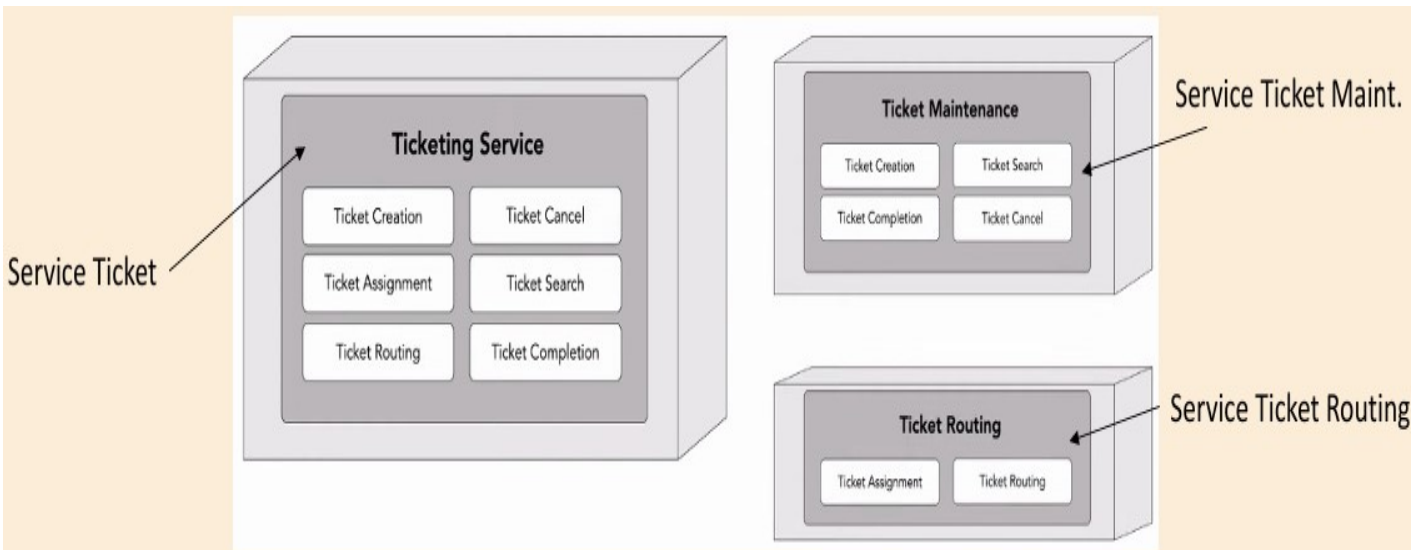
- L'adaptateur implémente l'interface du module de haut niveau
- L'adaptateur utilise l'interface du module de bas niveau
- L'adaptateur permet aussi de convertir les structuteurs de données entre les deux modules.

Service

Définition Service

Un service est une unité déployable qui accomplit une activité métier ou d'infrastructure.

Granularité d'un service ?



On fait quoi ? Un gros service ou plusieurs petits ?

La réponse . . . ça dépend !

Architecture microservices

- Une fonctionnalité métier
- Plusieurs centaines de services
- Forte interconnexion.

Architecture service-based

- Plusieurs fonctionnalités métiers
- De 3 à 12 services
- Eviter les interconnexions.

Un Module

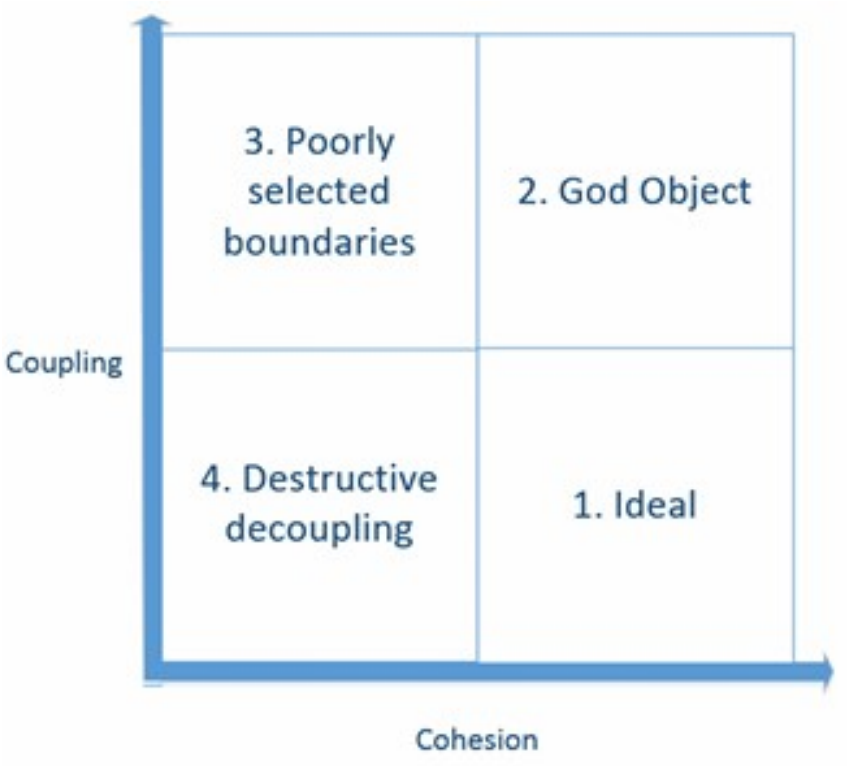
Définition module

Un module logiciel est une unité logicielle déployable, « manageable », réutilisable, composable et stateless qui fournit une interface concise au client.

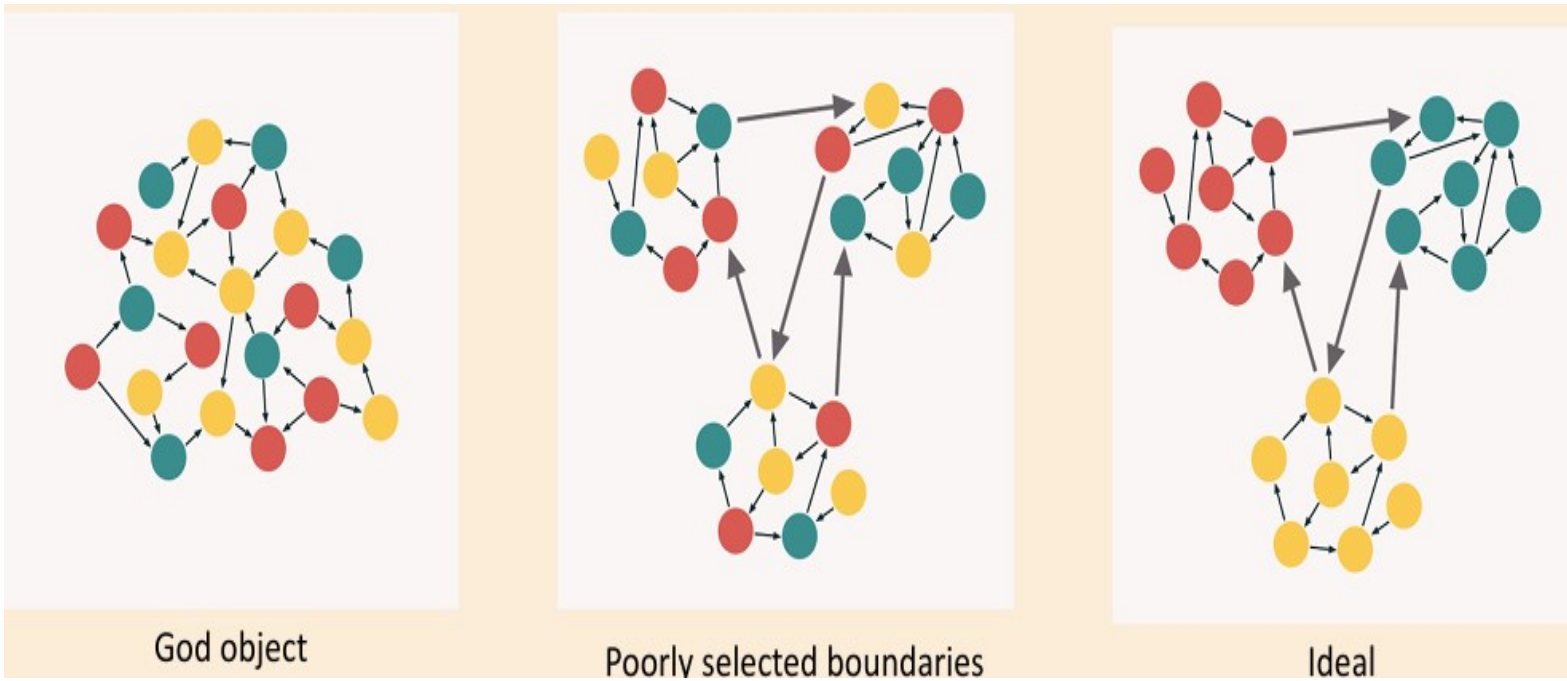
A noter que la modularité participe à la réussite d'un logiciel dans le temps et est un sujet très complexe.

L'objectif est

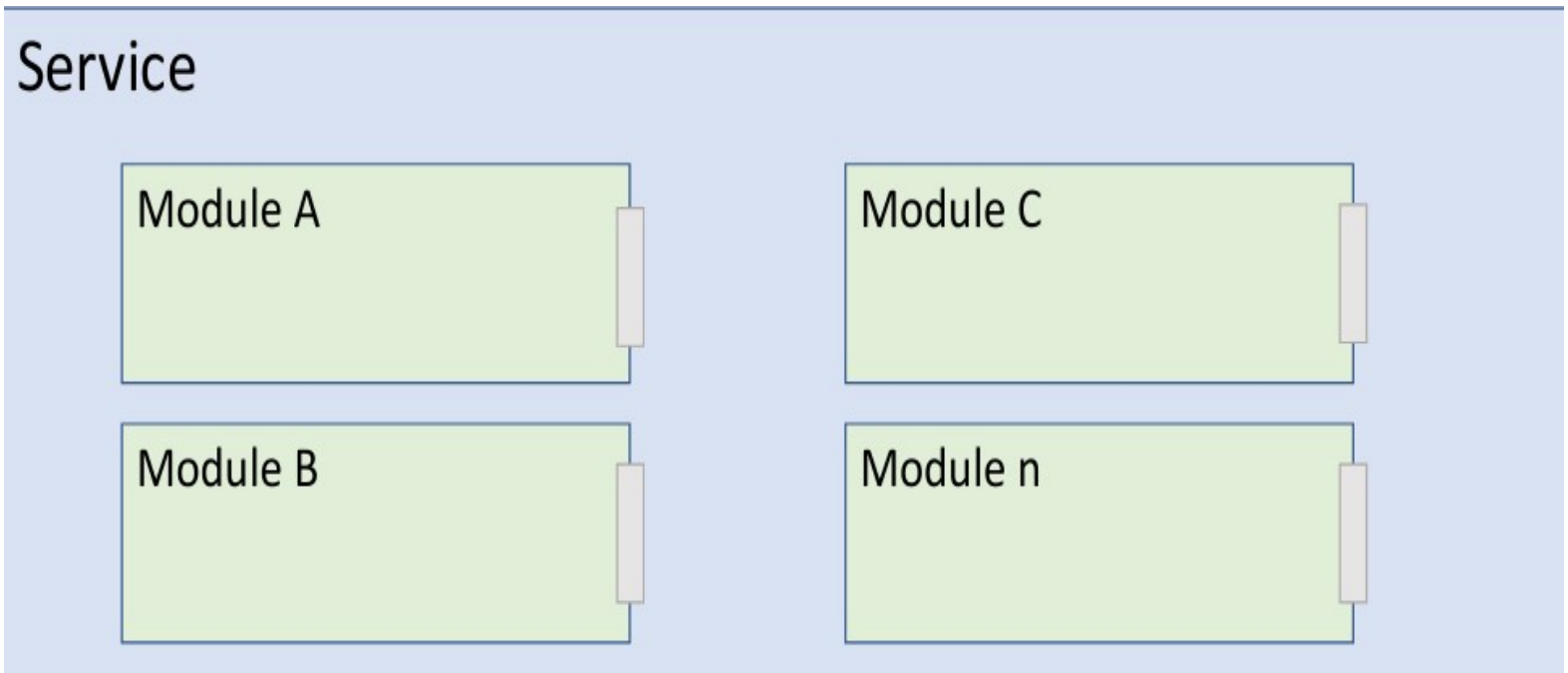
Respecter le principe « Low coupling, High cohésion ».



Illustration



Modules et Services

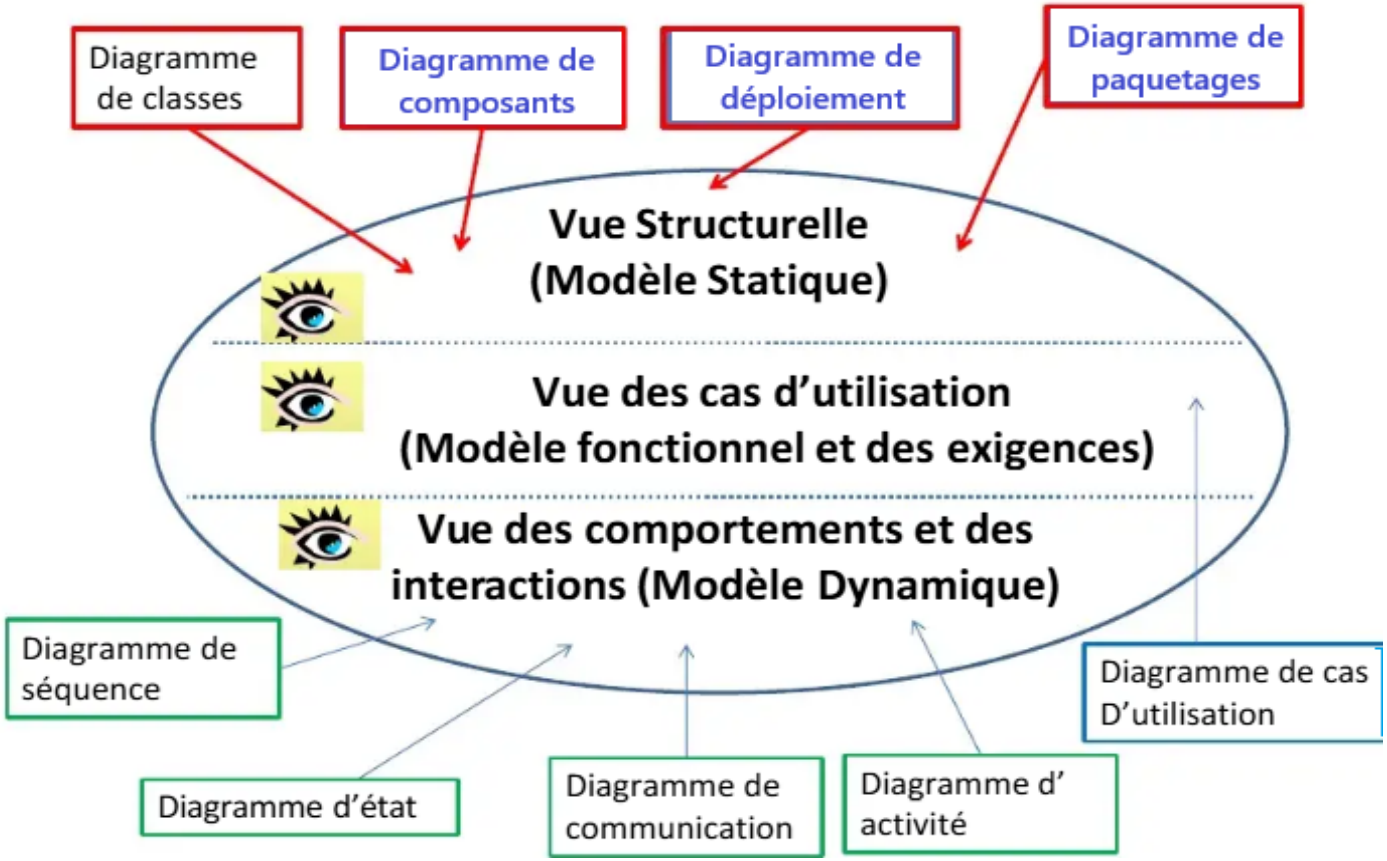


Conclusion

- Ces concepts sont utilisés partout, tout le temps
- Nous avons vu
  - Les interfaces : définissent comment on appel un service (paramètres) et qu'est-ce qu'on obtient d'un service (type de retour)
  - L'inversion de dépendance : éviter de tout package $\rightarrow$ **Build** $\rightarrow$ **Test** $\rightarrow$ **Deploy**
    - (Utilise la notion d'interface).
  - Qu'est-ce qu'un service : granularité différente suivant l'architecture
    - (Utilise la notion d'inversion de dépendance).

Plan

Les diagrammes UML



Diagrammes Structurels ou Diagrammes statiques (1/)

- Diagramme de classes : il représente les classes intervenant dans le système
- Diagramme d'objets : il sert à représenter les instances de classes (objets) utilisées dans le système
- Diagramme de composants : il permet de montrer les composants du système d'un point de vue physique, tels qu'ils sont mis en œuvre (fichiers, bibliothèques, bases de données...)

Diagrammes Structurels ou Diagrammes statiques (2/)

- Diagramme de déploiement : il sert à représenter les éléments matériels (ordinateurs, périphériques, réseaux, systèmes de stockage...) et la manière dont les composants du système sont répartis sur ces éléments matériels et interagissent avec eux
- Diagramme des paquetages : permet de représenter la hiérarchie des paquetages du projet, leur organisation et leurs interdépendances, simplifie les diagrammes (donc plus simple à comprendre)
- Diagramme de structures composites : permet de décrire la structure interne d'un objet complexe lors de son exécution (c'est à dire, décrire l'exécution du programme), dont ses points d'interaction avec le reste du système.

Diagrammes Comportementaux ou Diagrammes dynamiques

- Diagramme des cas d'utilisation : il décrit les possibilités d'interaction entre le système et les acteurs, c'est-à-dire toutes les fonctionnalités que doit fournir le système
- Diagramme états-transitions : il montre la manière dont l'état du système (ou de sous-parties) est modifié en fonction des événements du système
- Diagramme d'activité : variante du diagramme d'états-transitions, il permet de représenter le déclenchement d'événements en fonction des états du système et de modéliser des comportements parallélisables (multithreads ou multi-processus)

Diagrammes Comportementaux ou Diagrammes dynamiques

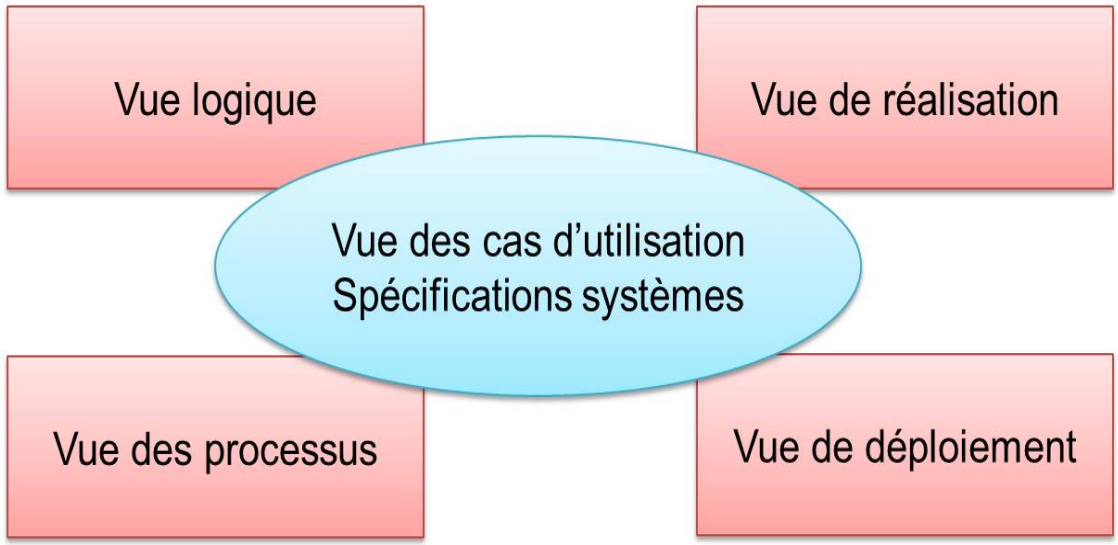
- Diagramme d'interactions
  - Diagramme de séquence : la représentation séquentielle du déroulement des traitements et des interactions entre les éléments du système et/ou des acteurs.
  - Diagramme de communication : la représentation simplifiée d'un diagramme de séquence se concentrant sur les échanges de messages entre les objets
  - Diagramme global d'interaction : variante du diagramme d'activité où les nœuds sont des interactions, permet d'associer les notations du diagramme de séquence à celle du diagramme d'activité, ce qui permet de décrire une méthode complexe
  - Diagramme de temps : la représentation des interactions où l'aspect temporel est mis en valeur ; il permet de modéliser les contraintes d'interaction entre plusieurs objets, comme le changement d'état en réponse à un événement extérieur

Le Rôle Crucial de la Documentation

Une documentation complète améliore

- la communication,
- la compréhension du système,
- facilite le transfert de connaissances aux nouveaux membres,
- Elle doit inclure le contexte, les objectifs architecturaux, les vues, et les exigences non fonctionnelles.

Le modèle « 4+1 » vues, dit de Kruchten



Le modèle de Kruchten dit modèle des 4 + 1 vues est celui adopté dans le Processus unifié. Ici encore, le modèle d'analyse, mentionné vue des cas d'utilisation, constitue le lien et motive la création de tous les diagrammes d'architecture.

Vue Logique

- Rôle : Décrit les classes clés, les objets, et les relations entre eux dans le système
- Objectif : Représenter l'espace problème (domaine métier) et l'espace solution (conception orientée objet). Elle permet de s'assurer que le système répond aux exigences fonctionnelles.
- Diagrammes UML : Le Diagramme de Classes et le Diagramme d'Objets sont les outils principaux pour exprimer cette vue.

La Vue des Processus (Systèmes Multitâches)

- Description : Décrit les interactions entre les processus, threads ou tâches. Elle exprime la synchronisation et l'allocation des objets
- Objectif : Vérifier le respect des contraintes de performance, d'efficacité et de fiabilité des systèmes multitâches
- Diagrammes UML : Exclusivement dynamiques (séquence, communication, états-transitions).

La Vue de Réalisation (Développement)

- Description : Permet de visualiser l'organisation des composants (code source, bibliothèques dynamiques et statiques) dans l'environnement de développement. Elle est utile pour la gestion de la configuration (auteurs, versions)
- Diagrammes UML : Uniquement les **diagrammes de composants**

La Vue de Déploiement (Environnement d'Exécution)

- Description : Représente le système dans son environnement d'exécution physique. Elle gère les contraintes géographiques, de bande passante, temps de réponse et tolérance aux pannes/fautes
- Diagrammes UML : Diagrammes de composants et diagrammes de déploiement.

Critères de documentation efficaces

- Utiliser un langage clair et simple pour être accessible à toutes les parties prenantes
- Les représentations visuelles (diagrammes UML, organigrammes) sont souvent plus efficaces que le texte pour transmettre des idées complexes.

Outils de Modélisation UML et de Documentation

- Les outils de création de diagrammes UML (comme Visio ou Lucidchart) facilitent la création de diagrammes,
- Pour la documentation structurée, des plateformes comme Atlassian Confluence peuvent être utilisées.

Approches de modélisation spécialisée

- Utilisation de langages de modélisation comme ArchiMate ou le C4 Model pour décrire l'architecture logicielle,
- Les plateformes no-code comme AppMaster permettent de concevoir visuellement l'architecture, réduisant le besoin d'une documentation écrite approfondie.