

Architecture logicielle et conception avancée

Conception architecturale

1. Introduction
2. Modéliser l'architecture avec UML
3. Éléments architecturaux
4. Styles architecturaux
 1. Architecture pipeline
 2. Architecture avec référentiel de données
 3. Architecture MVC
 4. Architecture multi-couches
 5. Architecture n-niveaux
5. Développer un modèle architectural

1. Introduction

Qu'est-ce qu'une architecture logicielle ?

- L'architecture d'un logiciel est la structure des structures (modules) d'un système – Elle inclut
 - Les composants logiciels
 - Les propriétés externes visibles de ces composants
 - Les relations entre ces composants

Cf. [Bass, Clements, and Kazman (1998)]

Introduction

Qu'est que la description d'une architecture logicielle ?

- La définition de 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
- Les propriétés de ces composants
- Leur contenu (e.g., classes, autres composants)

- Les machines ou dispositifs matériels sur lesquels ces modules seront déployés

Introduction

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 – Pour faciliter la réutilisation et la réutilisabilité

Utilité d'une architecture logicielle [Garlan 2000]

Introduction

- 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, composants, caractéristiques, fonctions ou données communes
- Construction : fournit un plan de haut-niveau du développement et de l'intégration des modules en mettant en évidence les composants, les interactions et les dépendances

Utilité d'une architecture logicielle [Garlan 2000]

Introduction

- Évolution : met en évidence les points où un système peut être modifié et étendu. La séparation composant/connecteur facilite une implémentation du type « plug-and-play »
- 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 : contribue à la gestion générale du projet en permettant aux différentes personnes impliquées de voir comment les différents morceaux du casse-tête seront agencés. L'identification des dépendance entre composants permet d'identifier où les délais peuvent survenir et leur impact sur la planification générale

2. Modéliser avec UML

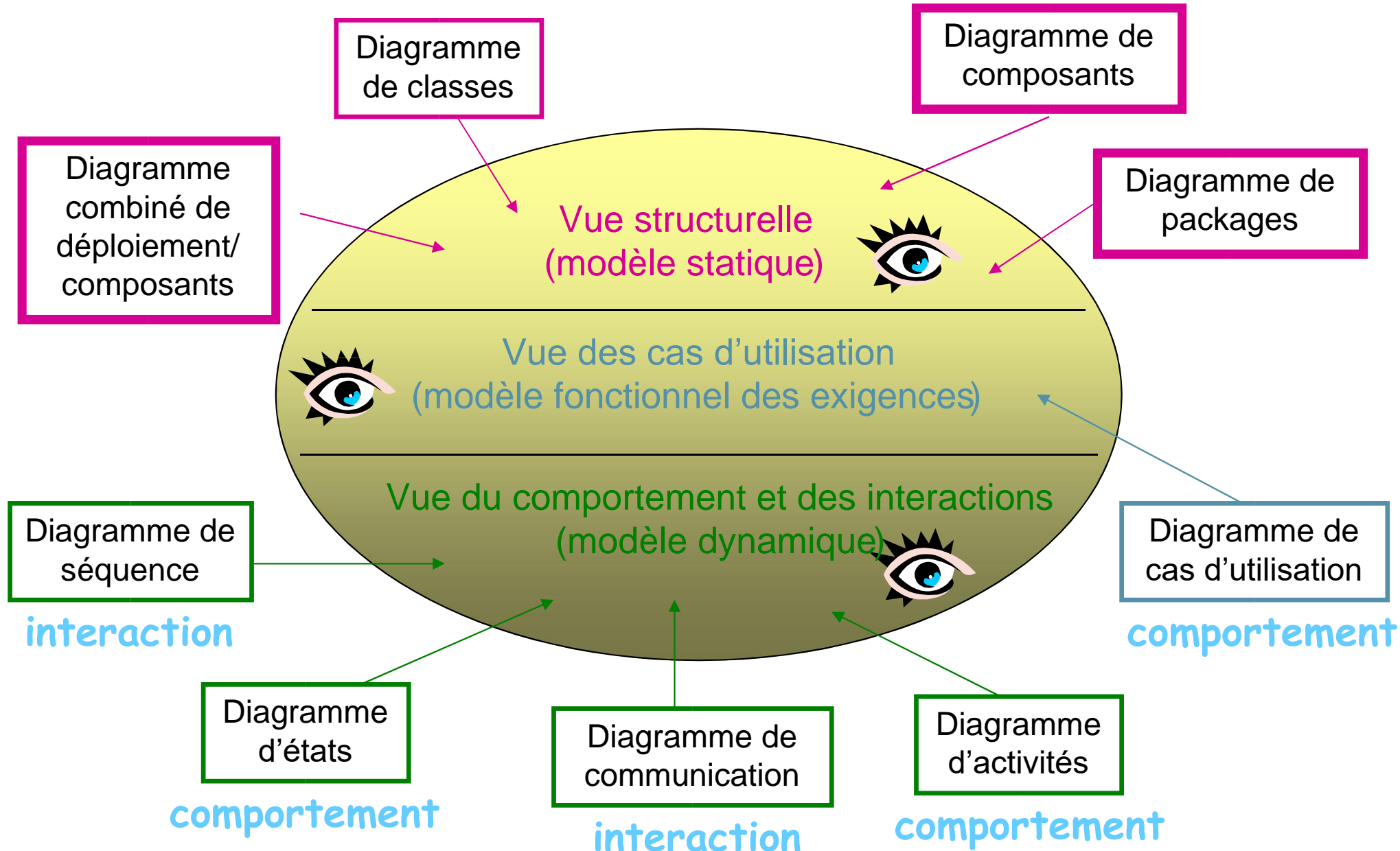
Les vues (structurelles) d'une architecture logicielle

- Vue logique. Description logique du système décomposé en sous-systèmes (modules + interface)
 - UML : diagramme de paquetages
- Vue d'implémentation. Description de l'implémentation (physique) du système logiciel en termes de composants et de connecteurs
 - UML : diagramme de composants
- Vue de déploiement. Description de l'intégration et de la distribution de la partie logicielle sur la partie matérielle

Modéliser l'architecture avec UML

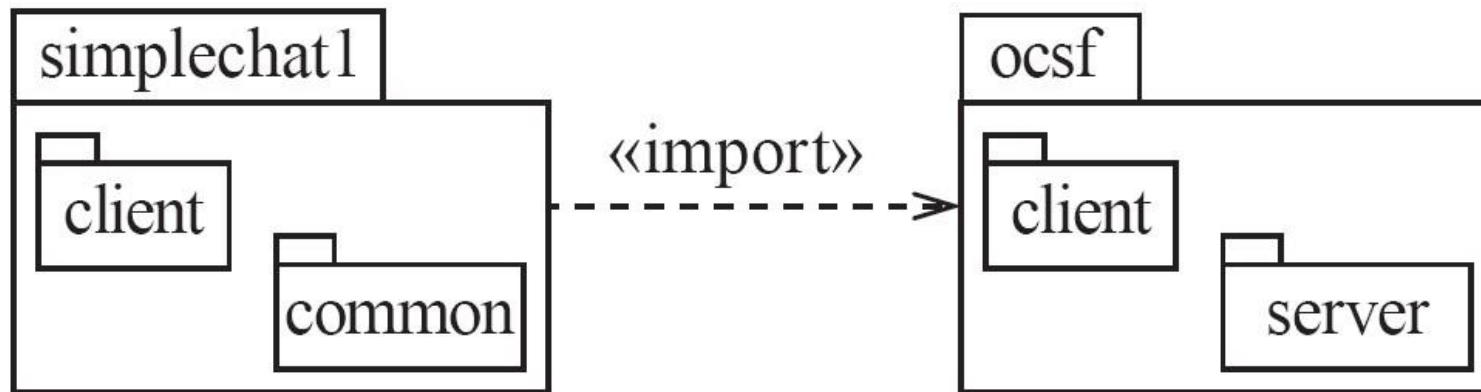
- UML: diagramme combiné de composants et de déploiement

Rappel : vues d'un programme



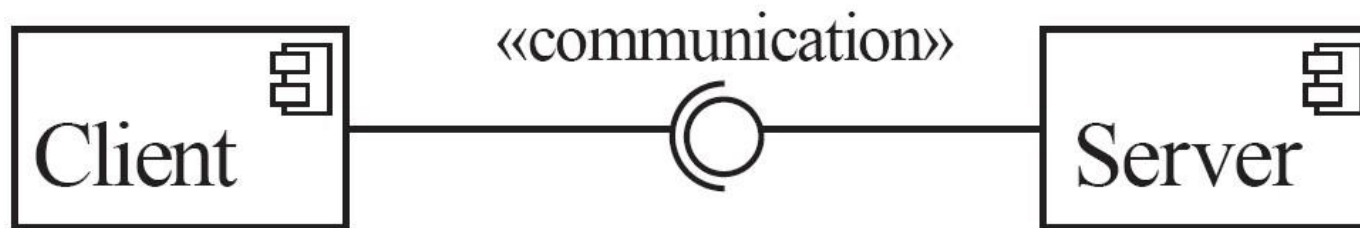
Modéliser l'architecture avec UML

Diagramme de paquetages



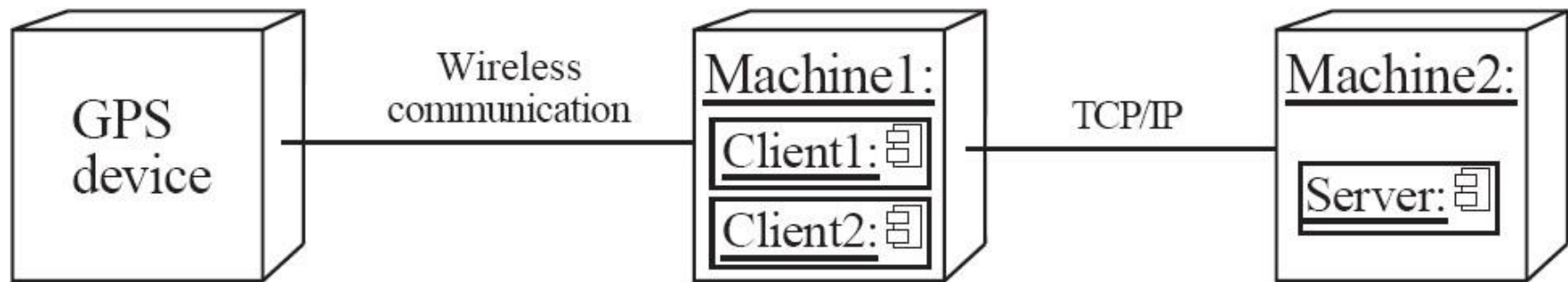
Modéliser l'architecture avec UML

Diagramme de composants

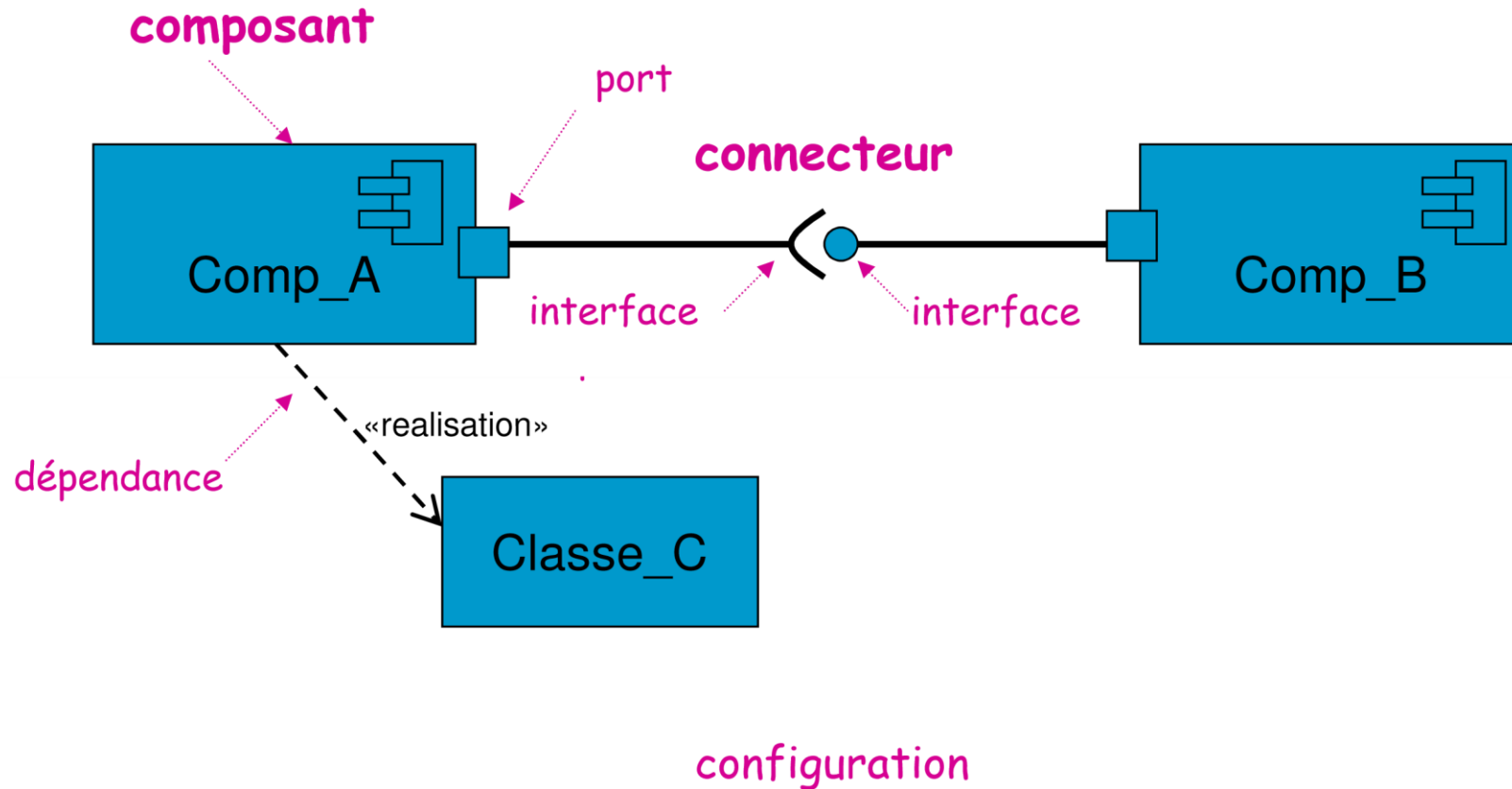


Modéliser l'architecture avec UML

Diagramme combiné de déploiement/composants



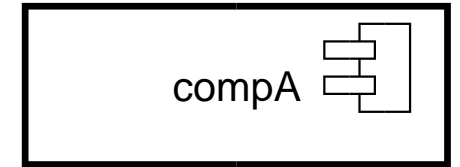
3. Éléments



- Deux ou plusieurs composants interagissent via un connecteur
- Chaque élément architectural possède une structure et/ou comportement pouvant être décrit par un modèle UML approprié

Modéliser l'architecture avec UML architecturaux

Éléments architecturaux



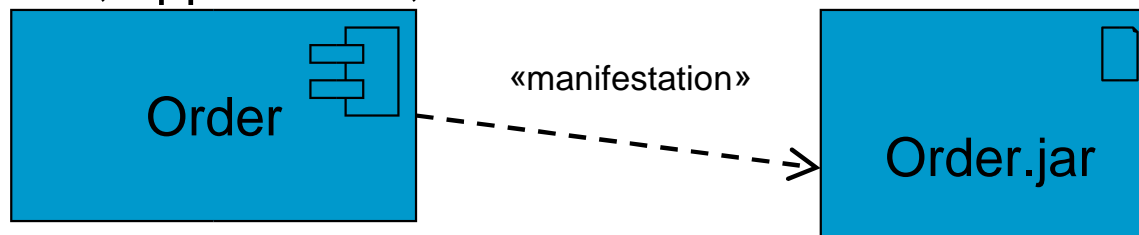
Composant

- Encapsule un traitement et/ou des données –
- Encapsule un sous-ensemble de fonctionnalités et/ou de données du système –
- Restreint l'accès à ce sous-ensemble au moyen d'une interface définie explicitement –
- Possède des dépendances explicitement définies pour exprimer les contraintes requises par son contexte d'exécution ou sa réalisation

Composant

Éléments architecturaux


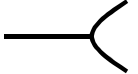

- Unité autonome servant de bloc de construction pour le système
- Les composants implémentent typiquement des services spécifiques à l'application
- La manifestation concrète d'un composant est appelé artéfact (instance du composant déployée sur le matériel)
 - N'importe quel type de code sur n'importe quel support numérique
 - Code source, fichiers binaires, scripts, fichiers exécutables, bases de données, applications, etc.



Interface de composant

- Permet à un composant d'exposer les moyens à utiliser pour communiquer avec lui

Éléments architecturaux

- Types d'interfaces
 - Interface offerte : définit la façon de demander l'accès à un service offert par le composant 
 - Interface requise : définit le type de services (aide) requis par le composant 
- Une interface est attachée à un port du composant
 - Port = point de communication du composant 
 - Plusieurs interfaces peuvent être attachées à un même port

Éléments architecturaux

Dépendances entre composants ----->

- Dépendance = relation entre deux composants
- Types de dépendances
 - Un composant peut dépendre d'un autre composant qui lui fournit un service ou une information
 - Un composant peut dépendre d'une classe qui implémente une partie de son comportement.
Dépendance de réalisation
 - Un composant peut dépendre d'un artefact (code source, fichier .jar, etc.) qui l'implante concrètement.
Dépendance de manifestation

Connecteur 

Éléments architecturaux

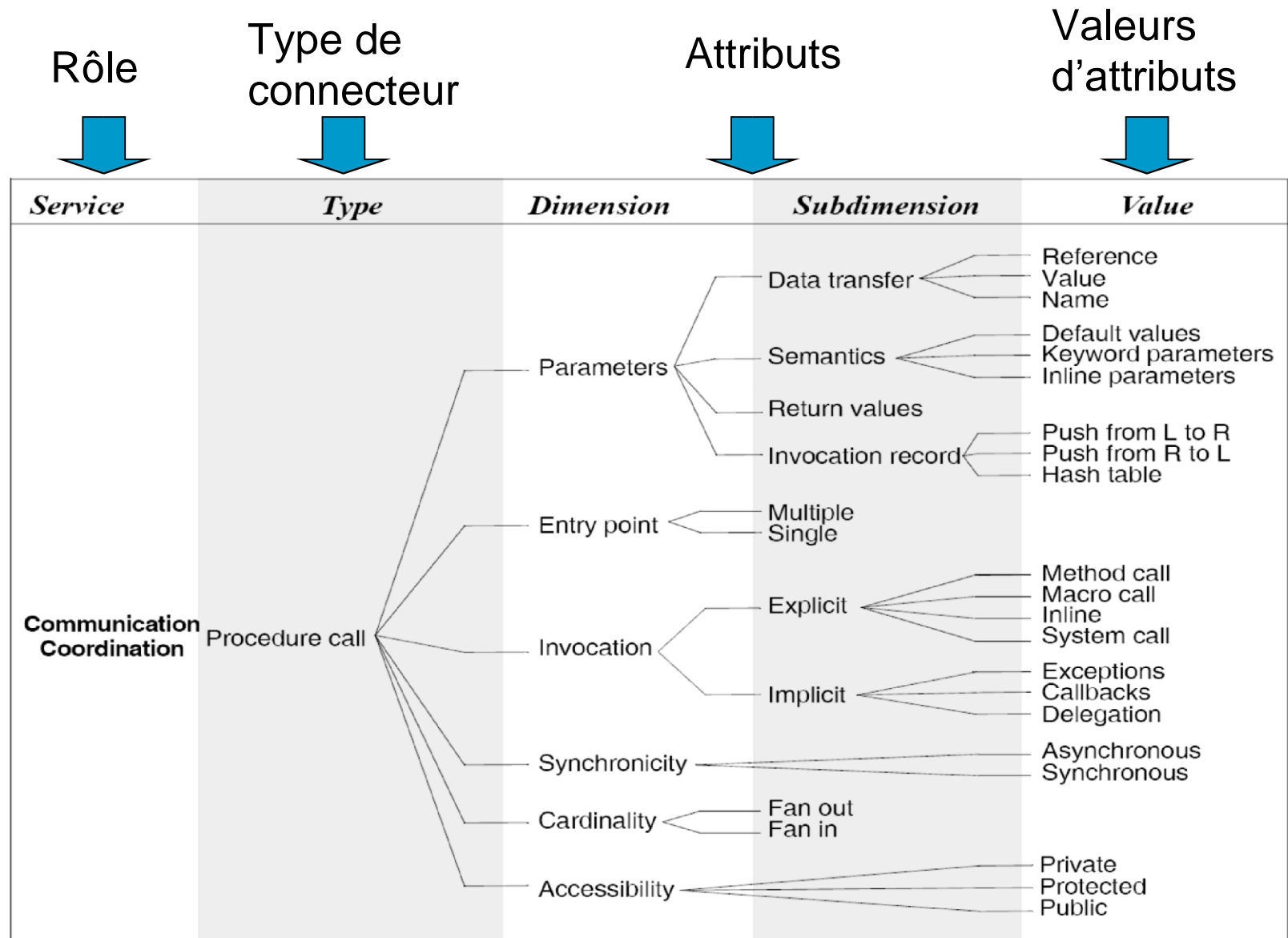
- Dans les systèmes complexes, les interactions peuvent constituer un enjeu encore plus important que les fonctionnalités des composants individuels
- Définition : élément architectural qui définit le type d'interactions entre les composants et les règles gouvernant ces interactions
- Un connecteur relie les ports de deux ou plusieurs composants
- Un connecteur décrit un mécanisme de connection indépendant de l'application
 - Représente un concept abstrait, paramétrable et indépendant des composants spécifiques qu'il relie
- Les attributs du connecteurs décrivent ses propriétés comportementales
 - E.g. sa capacité, le temps de latence, le type d'interaction (binaire/naire, asymétrique/symétrique, détails du protocole), etc.

Connecteur

Éléments architecturaux

- Un connecteur peut avoir un ou plusieurs rôles
 - Communication : rôle le plus fréquent
 - Coordination : contrôle du calcul, de la transmission des données, etc. Orthogonal aux autres rôles
 - Conversion : permet l'interaction entre composants développés indépendamment dans des langages différents par exemple
 - Facilitation : permet l'interaction entre composants conçus pour interagir ensemble. Synchronisation, accès contrôlés aux données partagées, etc.
- Selon le(s) rôles qu'il doit remplir et le type de communication souhaitée entre deux composants, choix d'un type
 - Procedure call (comm + coord) , Data access (comm + conv), Event, Stream, Linkage, Distributor, Arbitrator, Adaptor (conv)

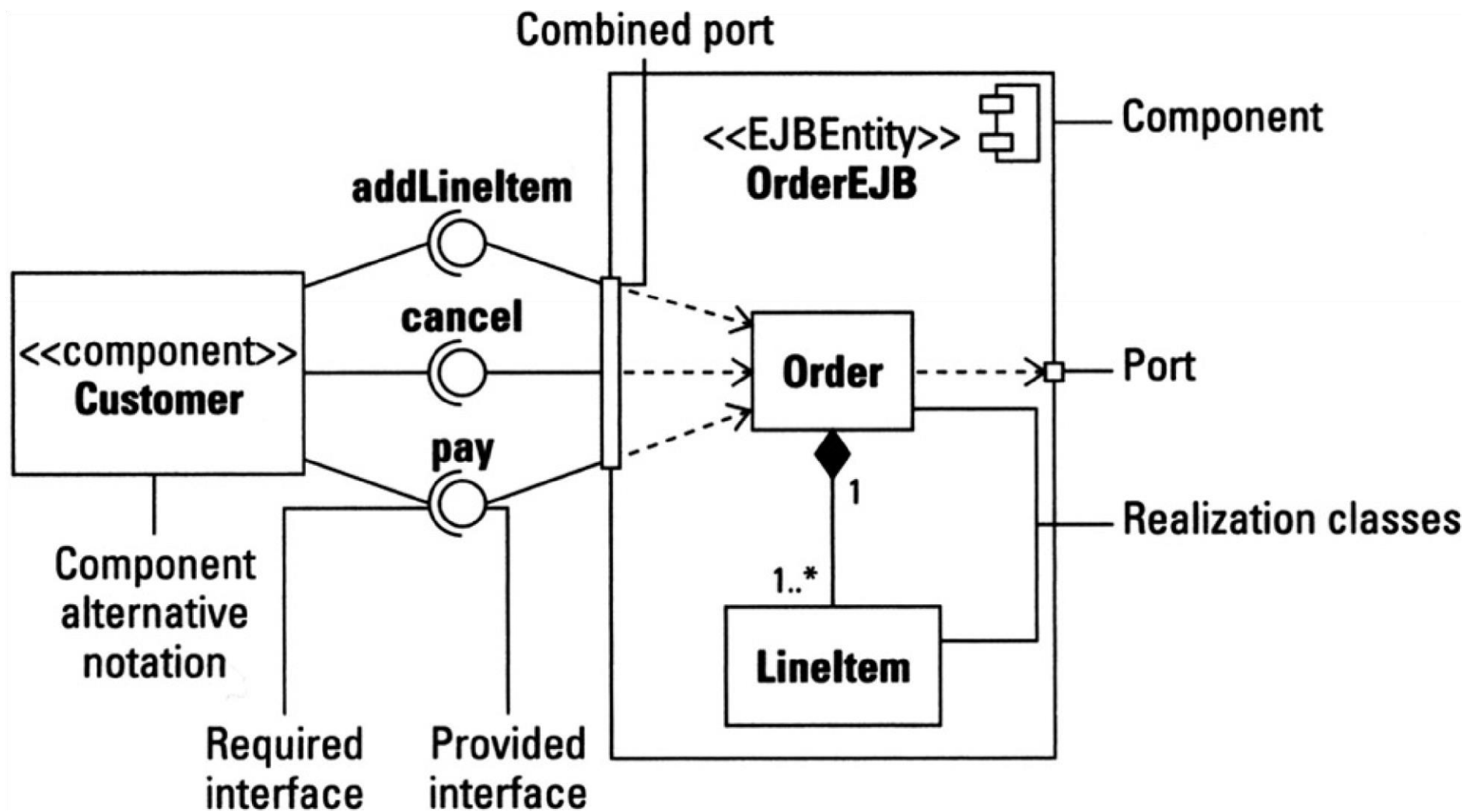
Éléments architecturaux



Éléments architecturaux

Software Architecture: Foundations, Theory, and Practice; R. N. Taylor, N. Medvidovic, and E. M. Dashofy; © 2008 John Wiley & Sons, Inc.

Exemple d'un diagramme de composants



Choix d'une architecture

Choix d'une architecture

- Dépend des exigences fonctionnelles et non fonctionnelles du logiciel
- Choix favorisant la stabilité : l'ajout de nouveaux éléments sera facile et ne nécessitera en général que des ajustements mineurs à l'architecture
- Influencé par certains « modèles connus » de décomposition en composants (styles architecturaux) et de mode d'interactions (e.g., orienté objet)

4. Style architecturaux

Un style architectural

- Est un patron décrivant une architecture logicielle permettant de résoudre un problème particulier
- Définit
 - Un ensemble de composants et de connecteurs (et leur type)
 - Les règles de configuration des composants et connecteurs (topologie)
 - Une spécification du comportement du patron
 - Des exemples de systèmes construits selon ce patron
- Constitue un modèle éprouvé et enrichi par l'expérience de plusieurs développeurs

- Compréhensibilité, maintenance, évolution, réutilisation, performance, documentation, etc.

Architecture pipeline

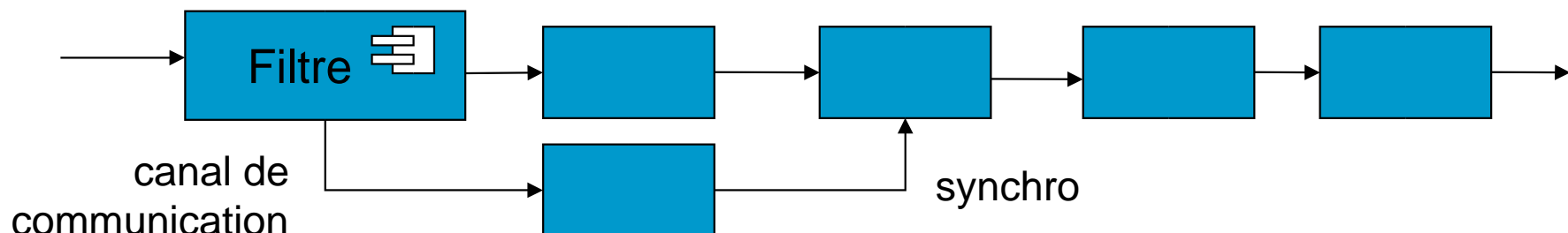
Convient bien aux systèmes de traitement et de transformation de données
Composants = filtre ; Connecteur = canal

— Filtre

- Traite indépendamment et asynchrone
- Reçoit ses données d'un ou plusieurs canaux d'entrée, effectue la transformation/traitement des données et envoie les données de sortie produites sur un ou plusieurs canaux de sorties
- Fonctionnent en concurrence. Chacun traite les données au fur et mesure qu'il les reçoit

— Canal

- Unidirectionnel au travers duquel circulent un flot de données (stream).
- Synchronisation et utilisation d'un tampon parfois nécessaire pour assurer le bon fonctionnement entre filtre producteur et filtre consommateur

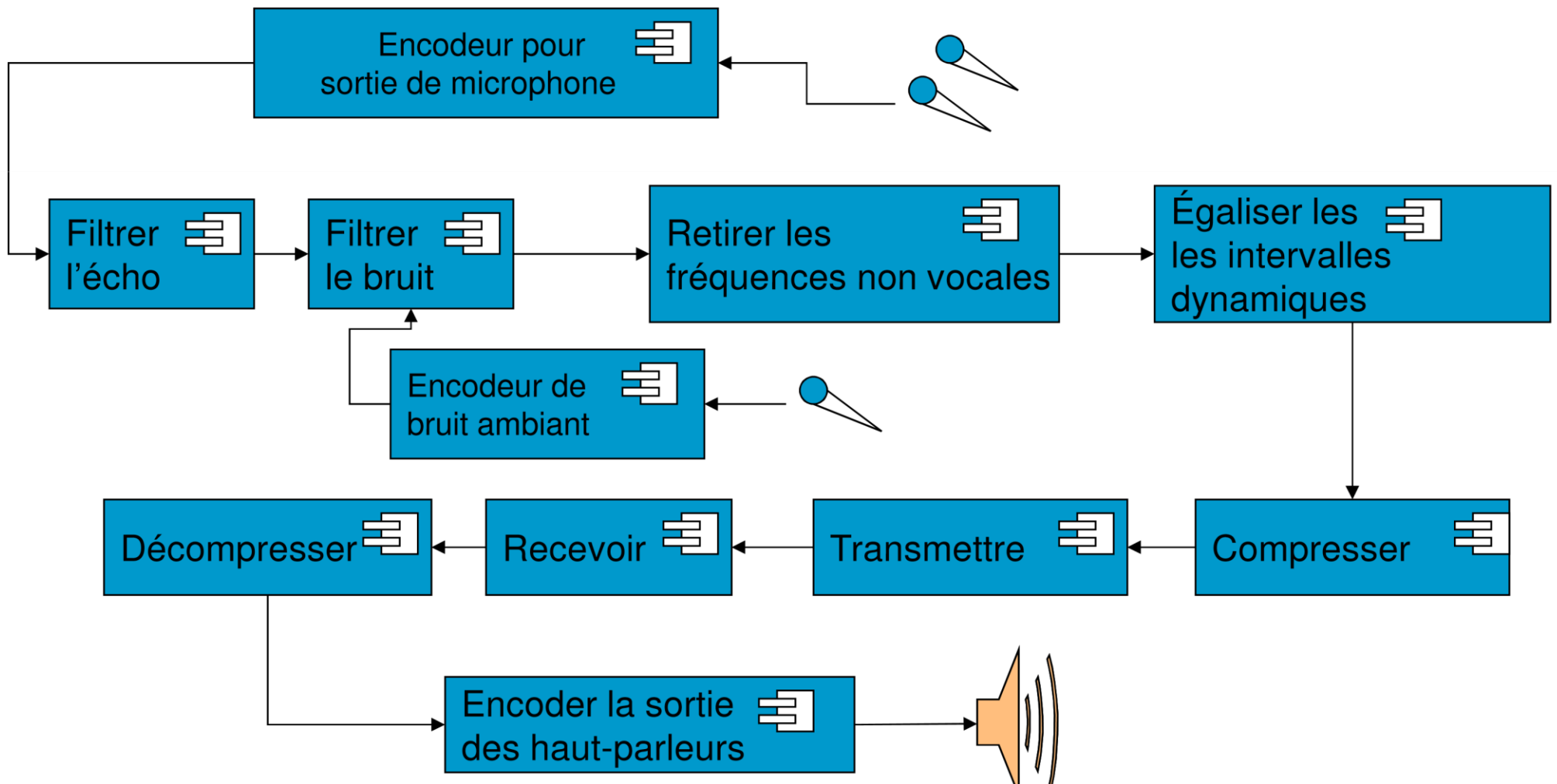


Architecture pipeline

- Exemples : application de traitement de texte, de traitement de signaux.
Compilateur (analyse lexicale, syntaxique, sémantique)

Architecture pipeline

Système de traitement du son



Architecture pipeline

Avantages

- Bon pour traitement en lot (batch)
 - Très flexible
- Analyse facilitée : performance, synchronisation, goulot d'étranglement, etc.
- Se prête bien à la décomposition fonctionnelle d'un système

Inconvénients

- Mauvais pour le traitement interactif
- Couplage : les filtres n'ont qu'une entrée et une sortie en général
- Abstraction : les filtres cachent généralement bien leurs détails internes

D'un point de vue conception

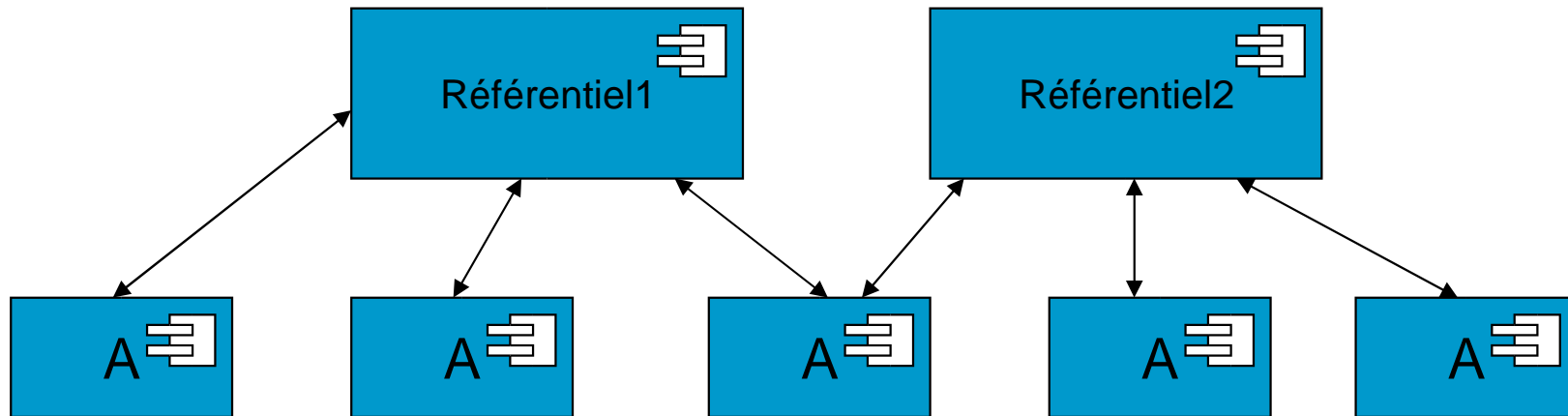
- Diviser pour régner : les filtres peuvent être conçus séparément
- Cohésion : les filtres sont un type de cohésion fonctionnelle
 - Réutilisabilité : les filtres peuvent très souvent être réutilisés dans d'autres contextes
- Réutilisation : il est souvent possible d'utiliser des filtres déjà existants pour les insérer dans le pipeline

Architecture avec référentiel de données (*shared data*)

Utilisée dans le cas où des données sont partagées et fréquemment échangées entre les composants

- Deux types de composants : référentiel de données et accesseur de données
 - Les référentiels constituent le medium de communication entre les accesseurs
- Connecteur : relie un accesseur à un référentiel
 - Rôle de communication, mais aussi possiblement de coordination, de conversion et de facilitation

Architecture avec référentiel



Variantes

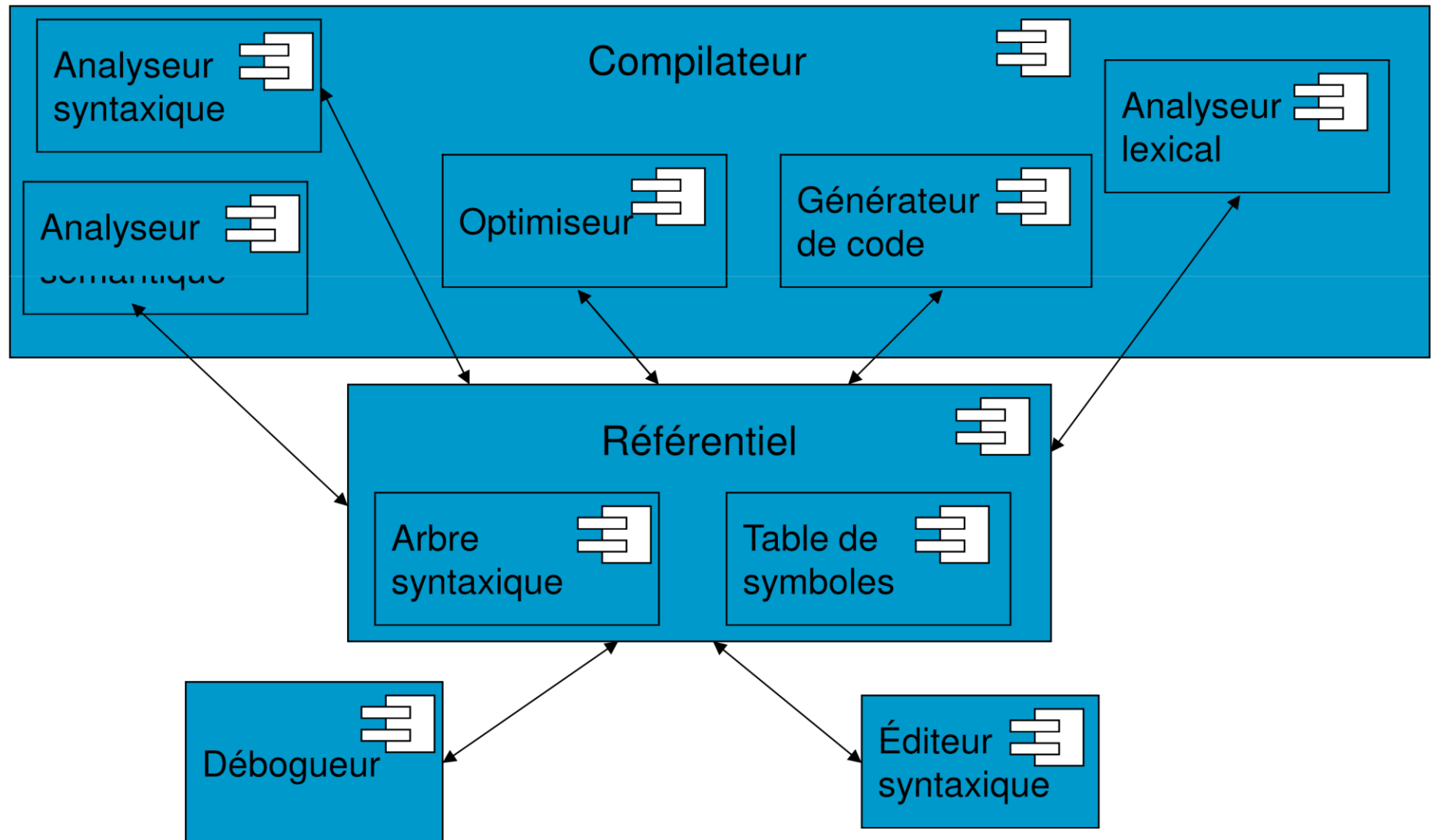
- Style tableau noir : les référentiels sont des agents actifs. Lorsqu'ils reçoivent une données, ils informent tous les accesseurs concernés
- Style référentiel : les référentiels sont passifs. Simple vocation de stockage. Les composants accède aux référentiels comme ils le désirent

Architecture avec référentiel

- Exemples : application de bases de données, systèmes Web, systèmes centrés sur les données (e.g. système bancaire, système de facturation ,etc.), environnement de programmation

Architecture avec référentiel

Environnement de programmation



Architecture avec référentiel

Avantages

- Avantageux pour les applications impliquant des tâches complexes sur les données, nombreuses et changeant souvent. Une fois le référentiel bien défini, de nouveaux services peuvent facilement être ajoutés

Inconvénients

- Le référentiel peut facilement constituer un goulot d'étranglement, tant du point de vue de sa performance que du changement

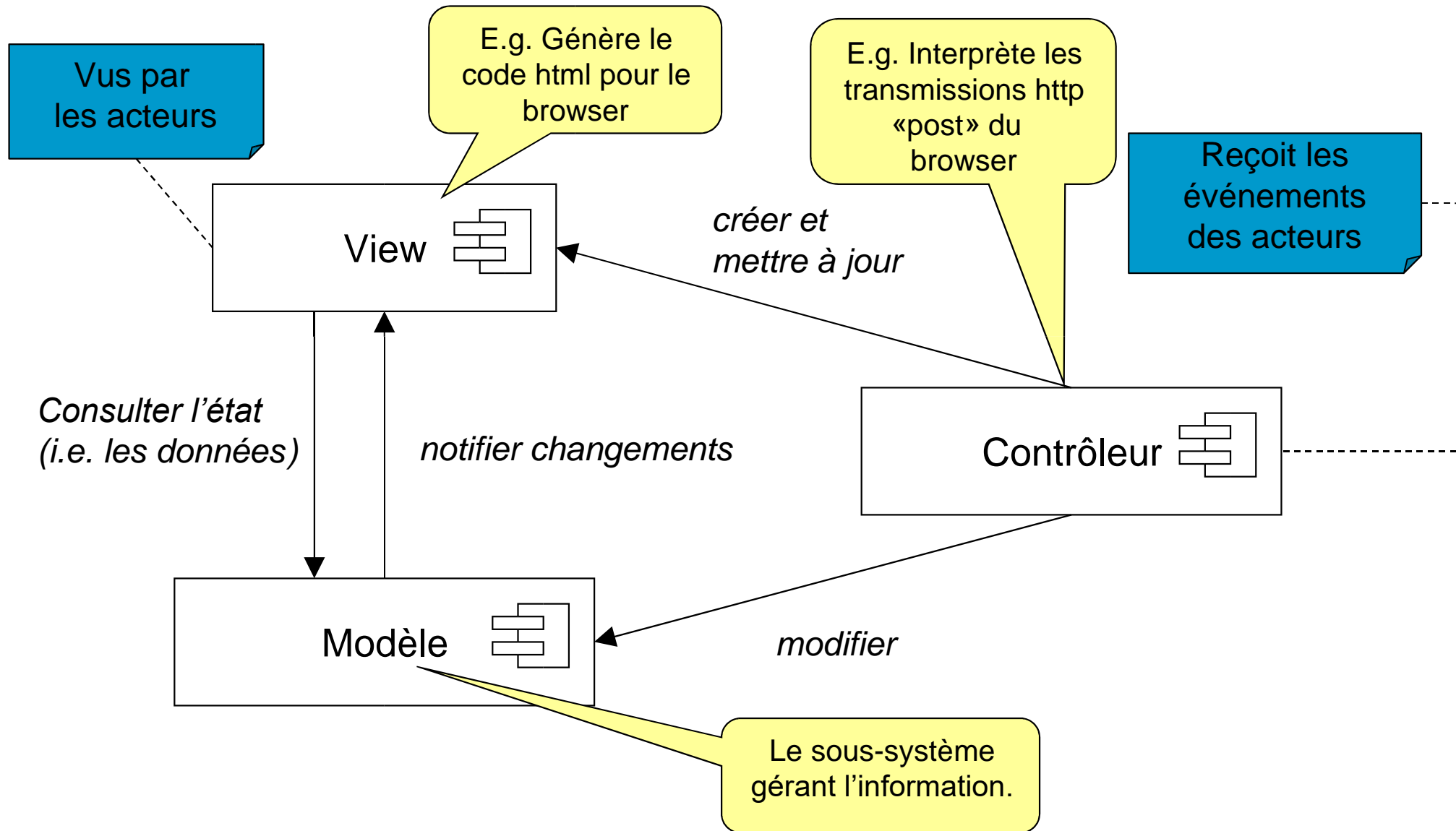
Architecture Modèle-Vue-Contrôleur (MVC)

Séparer la couche interface utilisateur des autres parties du système (car les interfaces utilisateurs sont beaucoup plus susceptibles de changer que la base de connaissances du système)

Composé de trois types de composants

- Modèle : rassemble des données du domaine, des connaissances du système. Contient les classes dont les instances doivent être vues et manipulées
- Vue : utilisé pour présenter/afficher les données du modèle dans l'interface utilisateur
- Contrôleur : contient les fonctionnalités nécessaires pour gérer et contrôler les interactions de l'utilisateur avec la vue et le modèle

Architecture Modèle-Vue-Contrôleur



Architecture Modèle-Vue-Contrôleur

Modèle : noyau de l'application

- Enregistre les vues et les contrôleurs qui en dépendent
- Notifie les composants dépendants des modifications aux données

Vue : interface (graphique) de l'application

- Crée et initialise ses contrôleurs
- Affiche les informations destinées aux utilisateurs
- Implante les procédures de mise à jour nécessaires pour demeurer cohérente
- Consulte les données du modèle

Contrôleur : partie de l'application qui prend les décisions

- Accepte les événements correspondant aux entrées de l'utilisateur
- Traduit un événement (1) en demande de service adressée au modèle ou bien (2) en demande d'affichage adressée à la vue
- Implémente les procédures indirectes de mise à jour des vues si nécessaire

Architecture Modèle-Vue-Contrôleur

Avantages : approprié pour les systèmes interactifs, particulièrement ceux impliquant plusieurs vues du même modèle de données. Peut être utilisé pour faciliter la maintenance de la cohérence entre les données distribuées

Inconvénient : goulot d'étranglement possible

D'un point de vue conception

- Diviser pour régner : les composants peuvent être conçus indépendamment
- Cohésion : meilleure cohésion que si les couches vue et contrôle étaient dans l'interface utilisateur.
- Couplage : le nombre de canaux de communication entre les 3 composants est minimal
- Réutilisabilité : la vue et le contrôle peuvent être conçus à partir de composants déjà existants

Architecture Modèle-Vue-Contrôleur

- Flexibilité : il est facile de changer l'interface utilisateur
- Testabilité : il est possible de tester l'application indépendamment de l'interface

Architecture multi-couches

Architecture multi-couches

Composants : chaque composant réalise un service

- Une couche offre un service (serveur) aux couches externes (client)

- Service créé indépendamment du logiciel ou spécifiquement

- Met à profit la notion d'abstraction, les couches externes sont plus abstraites (haut niveau) que les couches internes

Connecteurs : dépendent du protocole d'interaction souhaité entre couches

- Système fermé : une couche n'a accès qu'aux couches adjacentes. Les connecteurs ne relient que les couches adjacentes

- Système ouvert : toutes les couches ont accès à toutes les autres.

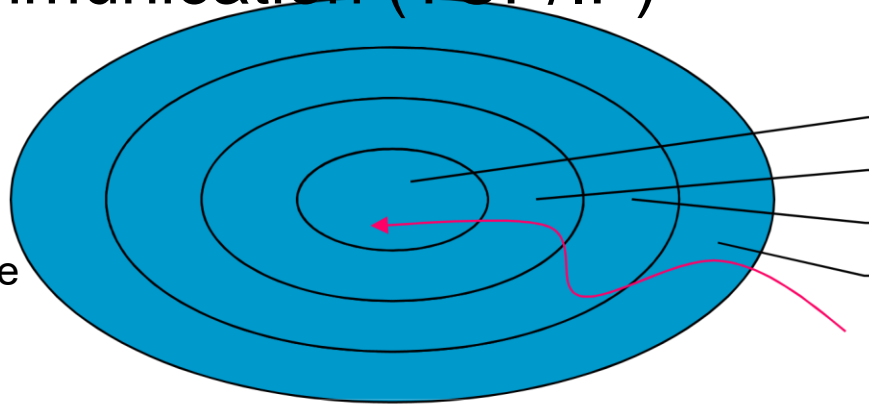
Les connecteurs peuvent relier deux couches quelconques

Exemples

: souvent utilisé pour les systèmes implémentant des protocoles de communication (TCP/IP)

Encrypter/décrypter un fichier

1. Entrée d'un mot de passe
2. Obtention d'une clef d'accès
3. Encryptage/décryptage du fichier
4. Fonctions d'encryptage/décryptage



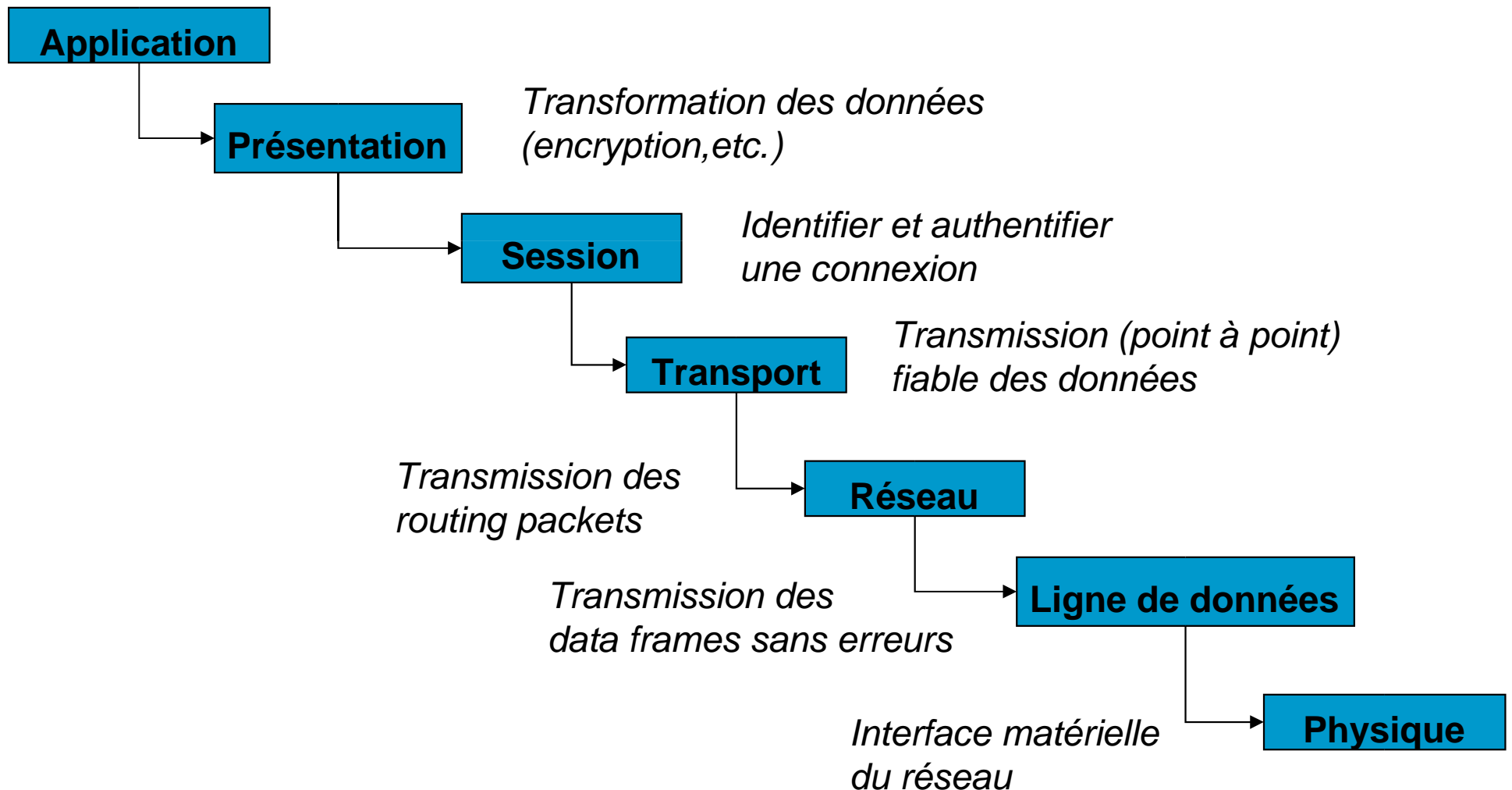
- Chaque couche est un composant avec une interface bien définie utilisée par la couche juste au dessus (i.e., externe)
- La couche supérieure (externe) voit la couche inférieure (interne) comme un ensemble de services offerts
- Un système complexe peut être construit en superposant les couches de niveau d'abstraction croissant
- Il est important d'avoir une couche séparée pour l'IU
- Les couches juste au dessous de l'IU offrent les fonctions applicatives définies par les cas d'utilisation
- Les couches les plus basses offrent les services généraux
- E.g., communication réseau, accès à la base de données

Architecture multi-couches

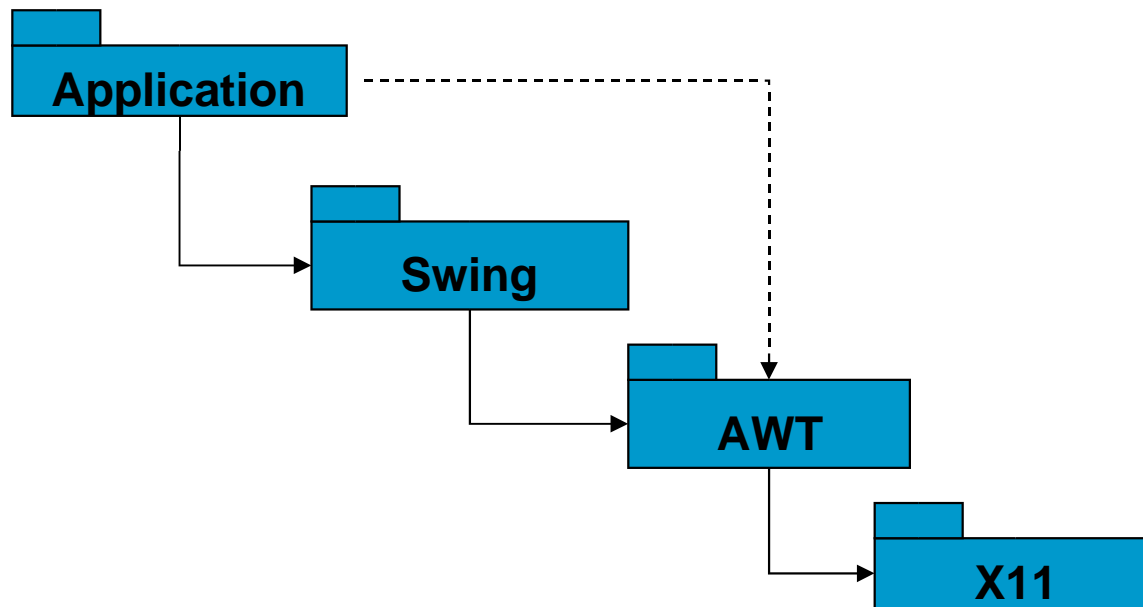
Architecture multi-couches

Systeme fermé

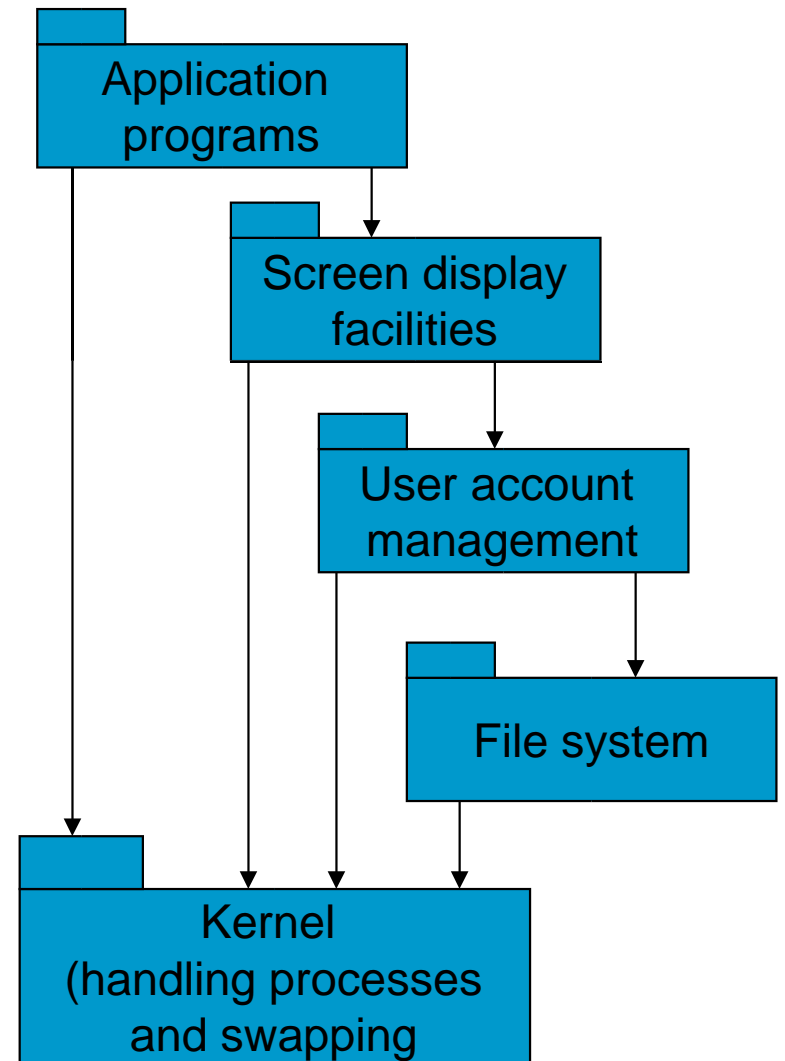
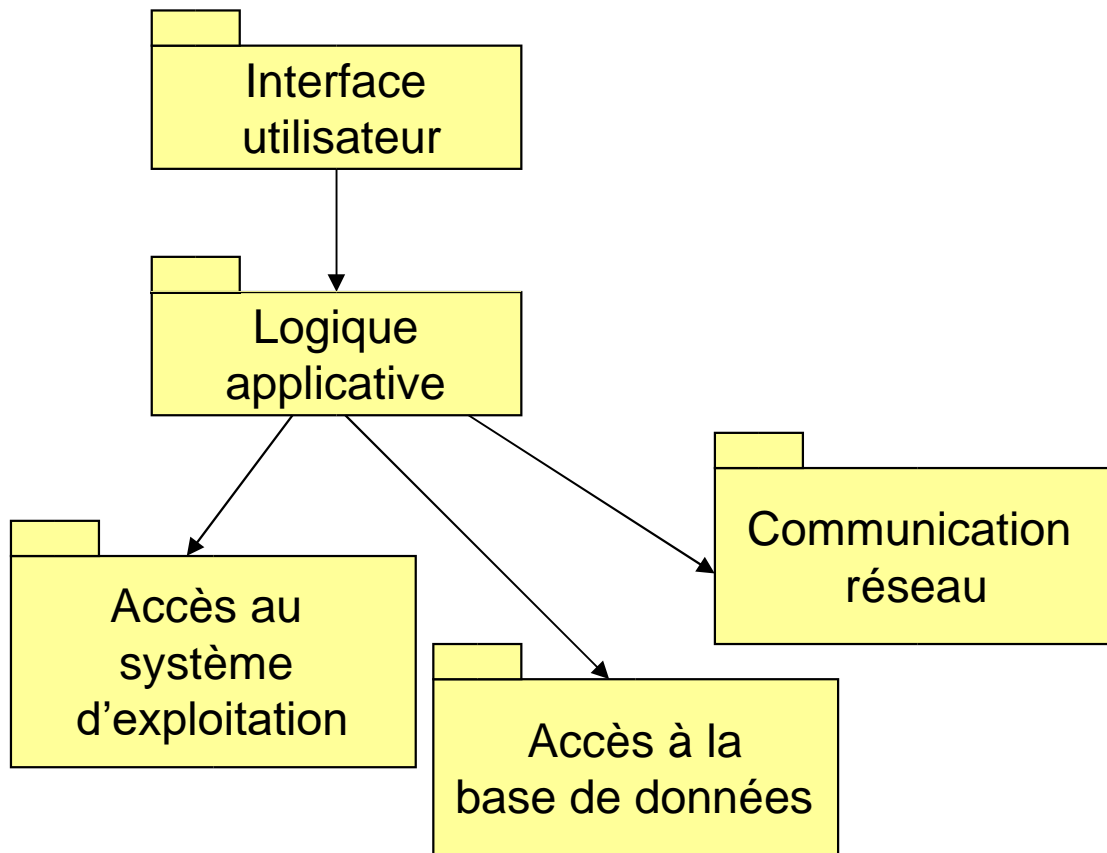
- Reference Model of Open Systems Interconnection (OSI model)



Architecture multi-couches



Architecture multi-couches



Architecture multi-couches

D'un point de vue conception

- Diviser pour régner : les couches peuvent être conçues séparément
- Cohésion : si elles sont bien conçues, les couches présenter une cohésion par couche
- Couplage : des couches inférieures bien conçues ne devraient rien savoir à propos des couches supérieures et les seules connexions autorisées entre couches se font via les API
- Abstraction : on n'a pas à connaître les détails d'implémentation des couches inférieures
- Réutilisabilité : les couches inférieures peuvent être conçues de

façon à offrir des solutions génériques réutilisables

- Réutilisation : on peut souvent réutiliser des couches développées par d'autres et qui proposent le service requis
- Flexibilité : il est facile d'ajouter de nouveaux services construits sur les services de plus bas niveau
- Anticipation du changement : en isolant les composants dans des couches distinctes, le système devient résistant
- Portabilité : tous les services relatifs à la portabilité peuvent être isolés
- Testabilité : les couches peuvent être testées indépendamment

Architecture n-niveaux

- Conception défensive : les API des couches constituent des endroits stratégiques pour insérer des assertions de vérification

Pour les systèmes distribués

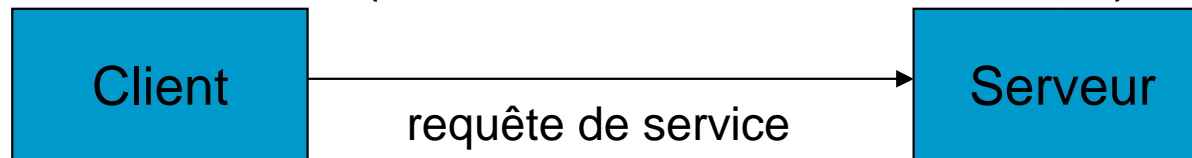
- Comparable à une architecture par couches... dont les couches seraient distribuées !
 - N.b. L'architecture multi-couche est généralement utilisée pour décrire la structure interne (non distribuée) d'un composant qui peut lui-même appartenir à une architecture (distribuée) n-partie
- Par abus de langage, la notion de tier a pris le sens de couche distribuée
- Composants : chaque niveau est représenté par un composant qui joue le rôle de client et/ou de serveur
- Connecteurs : relie un client à un serveur. Connexion asymétrique. Doit supporter les communications distantes (e.g., requêtes RPC, HTTP, TCP/IP)

Architecture multi-couches

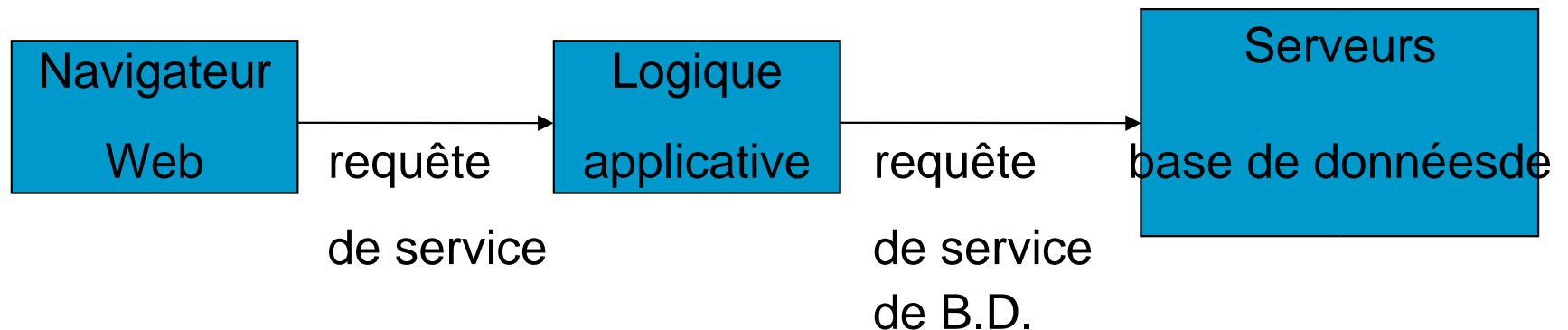
- Exemples
 - Client-serveur, Trois niveaux, Quatre niveaux

Architecture n-niveaux

Architecture 2-niveaux (client-serveur ou client lourd)

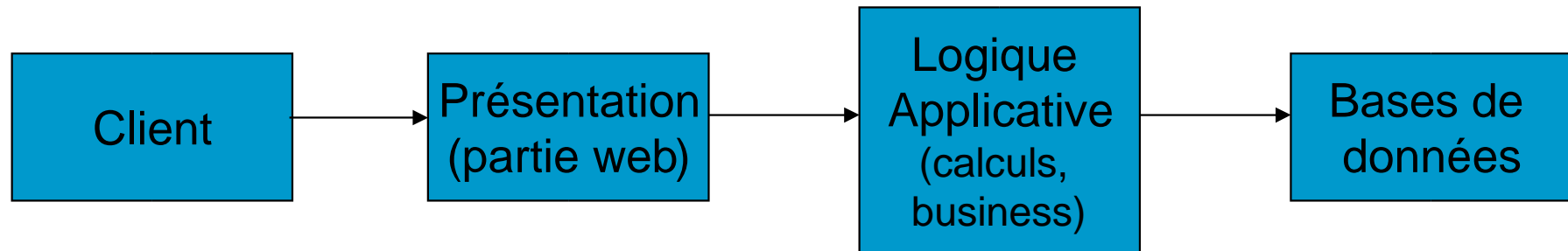


Architecture 3-niveaux (client léger)



Architecture 4-niveaux

Architecture n-niveaux



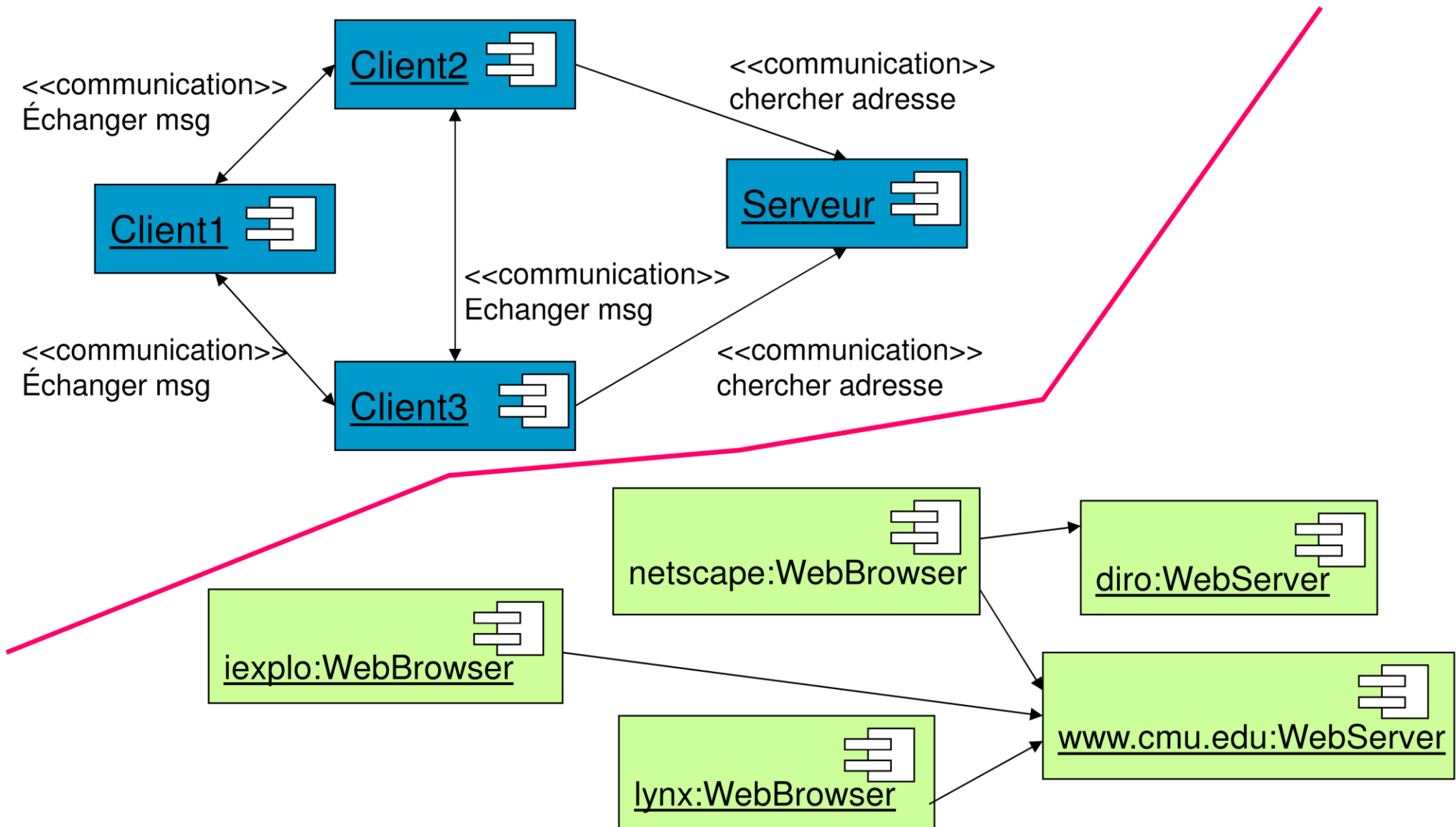
Architecture client-serveur

- Exemple typique : un système d'informations utilisant une base de données centrale. (cas spécial de l'architecture avec référentiel)
 - Clients : reçoivent les données de l'utilisateur, initient les transactions, etc.
 - Serveur : exécute les transactions, assure l'intégrité des données
- Architecture pair-à-pair = une généralisation de l'architecture client-serveur

Architecture n-niveaux

- Les composants peuvent tous à la fois être client et serveur
- Conception plus difficile: flot de contrôle plus complexe dû à la possibilité de *deadlocks*...

Architecture n-niveaux



Architecture n-niveaux

Architecture 3-niveaux

– Organisation en trois couches

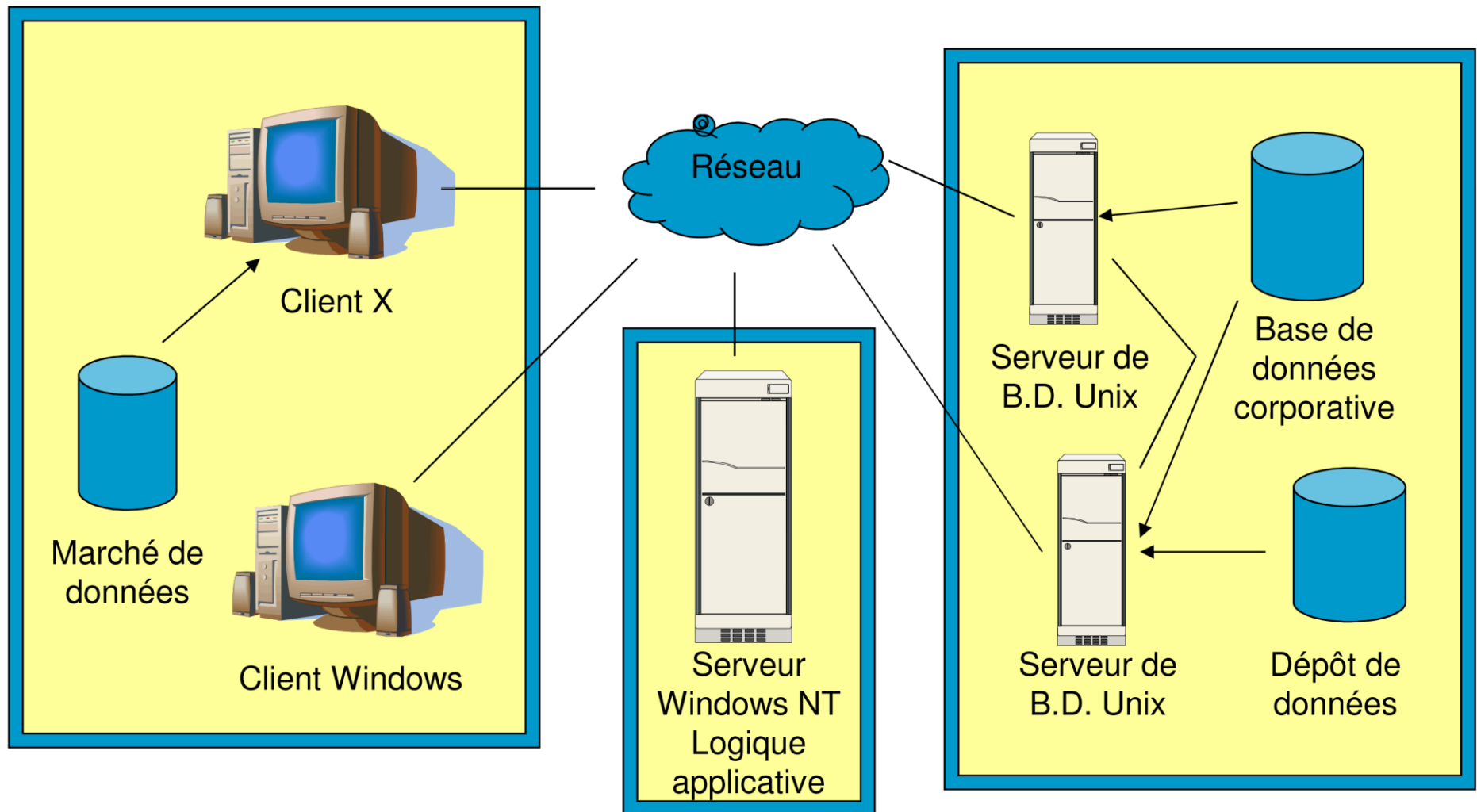
- Couche interface: Composé d'objets interfaces (*boundary objects*) pour interagir avec l'utilisateur (fenêtres, formulaires, pages Web, etc.)
- Couche logique applicative : Comporte tous les objets de contrôle et d'entités nécessaire pour faire les traitements, la vérification des règles et les notifications requises par l'application
- Couche de stockage : réalise le stockage, la récupération et la recherche des objets persistants

– Avantage sur l'architecture client-serveur

- Permet le développement et la modification de différentes interfaces utilisateurs pour la même logique applicative

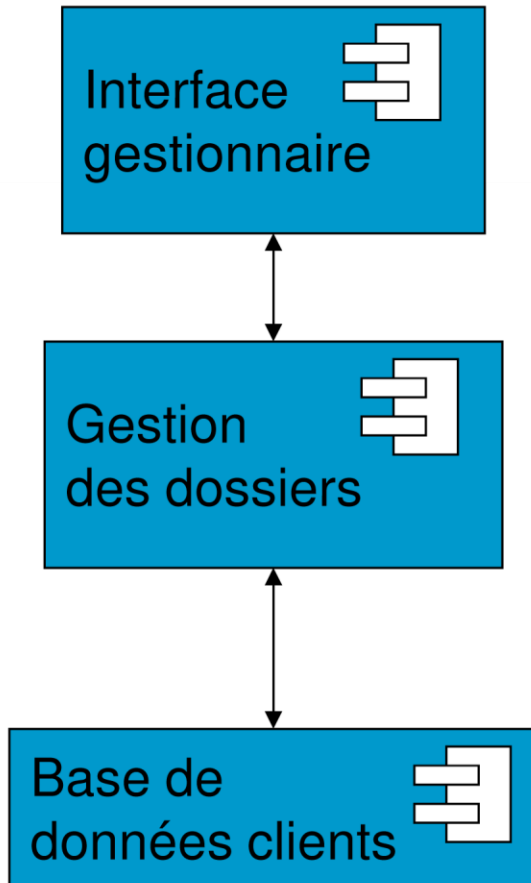
Architecture n-niveaux

Architecture 3-parties



Architecture n-niveaux

Architecture 3-niveaux



- À noter que la tendance veut que la partie cliente soit
 - De plus en plus mince, i.e., le client ne fait qu'afficher un contenu HTML
 - La logique applicative est alors responsable de créer les pages Web à afficher par le client
 - Il faut dissocier logique applicative et présentation des données

Architecture n-niveaux

Architecture 4-niveaux

- Architecture dont la couche logique applicative est décomposée en
 - Couche Présentation (JSP, servlets)
 - Présentation du contenu statique: Contenu HTML ou XML affiché par le client
 - Présentation du contenu dynamique : contenu organisé et créé dynamiquement par le serveur web (pour ensuite être affiché en HTML/XML par le client)
 - Couche Logique applicative (calculs purs) : réalise le cœur des traitements de l'application
- Avantage sur l'architecture 3-niveaux
 - Permet de supporter un grand nombre de formats de présentation différents (propres à chaque client), tout en réutilisant certains des objets de présentation entre les clients. Réduction des redondances...

Architecture n-niveaux

- Bien adaptée pour les applications Web devant supporter plusieurs types de clients

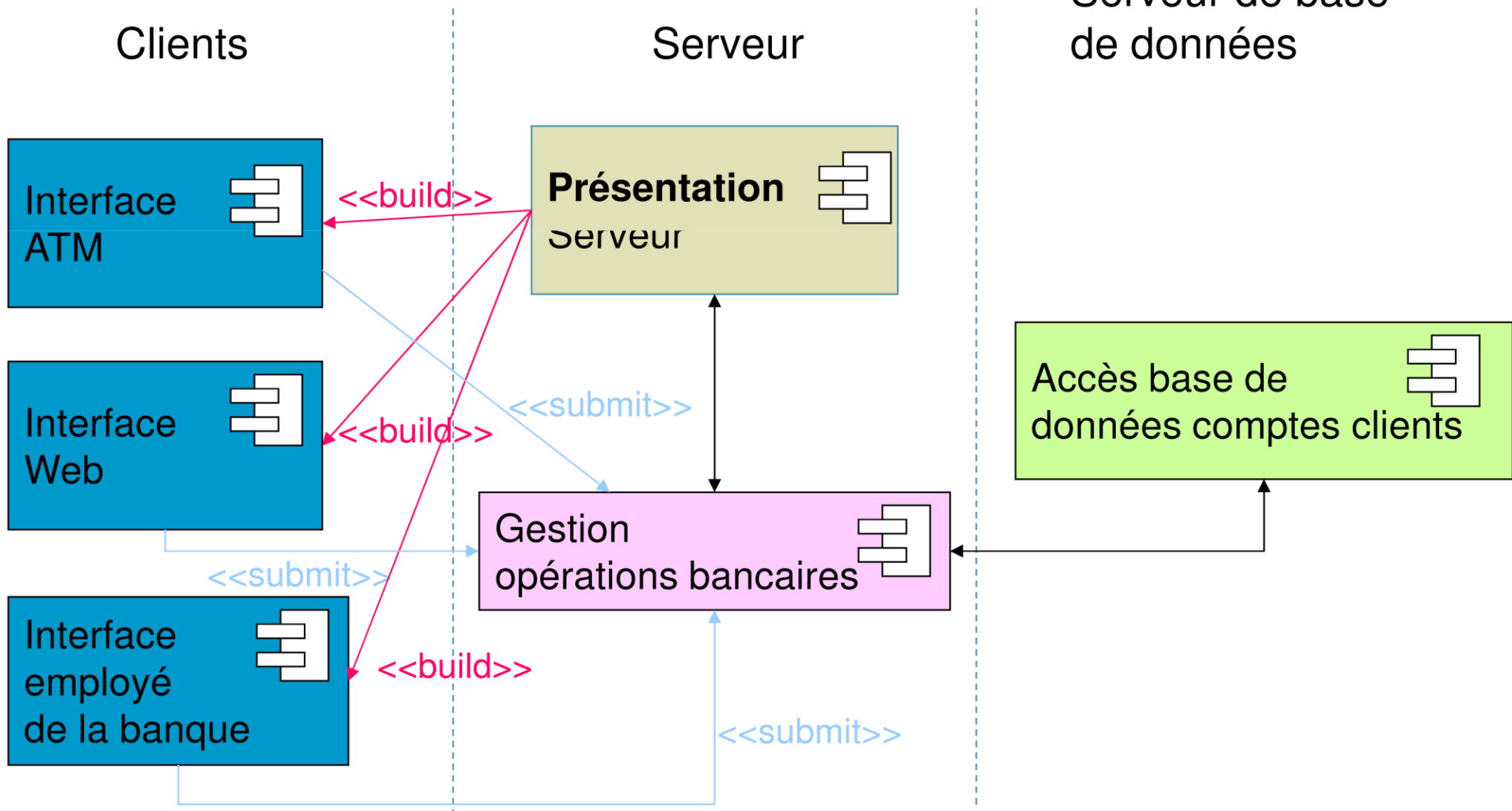


Java Server
Pages

Architecture n-niveaux

- Partie cliente

■ Architecture 4-niveaux



Architecture n-niveaux

- Partie serveur

Architecture n-niveaux

D'un point de vue conception

- Diviser pour régner : de façon évidente, client et serveur peuvent être développées séparément
- Cohésion : le serveur peut offrir des services cohésifs au client
- Couplage : un seul canal de communication existe souvent entre client et serveur
- Réutilisation : il est possible de trouver une bibliothèque de composants, interfaces, etc. pour construire les systèmes
 - Toutefois, les systèmes client-serveur dépendent très souvent de nombreuses considérations liées intimement à l'application

Flexibilité : il est assez facile d'ajouter de nouveaux clients et serveurs

Portabilité : on peut développer de nouveaux clients pour de nouvelles plateformes sans devoir porter le serveur

Testabilité : on peut tester le client et le serveur indépendamment

Conception défensive : on peut introduire des opérations de vérification dans le code traitant les messages reçus de part et d'autre

5. Développer un modèle architectural

Commencer par faire une esquisse de l'architecture

- En se basant sur les principaux requis des cas d'utilisation ;
décomposition en sous-systèmes
- Déterminer les principaux composants requis
- Sélectionner un style architectural

Raffiner l'architecture

- Identifier les principales interactions entre les composants et les interfaces requises
- Décider comment chaque donnée et chaque fonctionnalité sera distribuée parmi les différents composants
- Déterminer si on peut réutiliser un cadriceel existant (réutilisation) ou si on peut en construire un (réutilisabilité)

Considérer chacun des cas d'utilisation et ajuster l'architecture pour qu'il soit réalisable

Détailler l'architecture et la faire évoluer