Chapitre 1

Démarche d'Architecture logicielle

L'objectif de ce chapitre est :
- d'introduire quelques notions d'architecture
- avoir une vision globale en amont de la conception du logicielle
- préparer la conception logicielle

1 Introduction

La "démarche d'architecture" est la démarche d'un informaticien pour réaliser le dossier d'architecture de tout un Système d'Information.

Ce dossier est réalisé, par exemple, lors de l'avant-vente pour estimer et ainsi décrocher un contrat, ou plus généralement lors de la phase de conception pour déterminer ce que l'on doit développer et comment on doit le développer.

Il est à noter qu'il est aussi utilisé pour maintenir l'exploitation et le correctif du SI.

Il existe de nombreuses étapes, méthodes (cours NSY 206), outils et documents permettant de mener à bien une telle démarche qui peut varier en fonction du type d'architecture.

Le cours de NSY205 : *NSY205-Chapitre-01-01_ArchitectureLogicielle* présente cette démarche.

L'objectif de ce chapitre de cours est de faire un résumé de ce cours adapté à notre contexte :

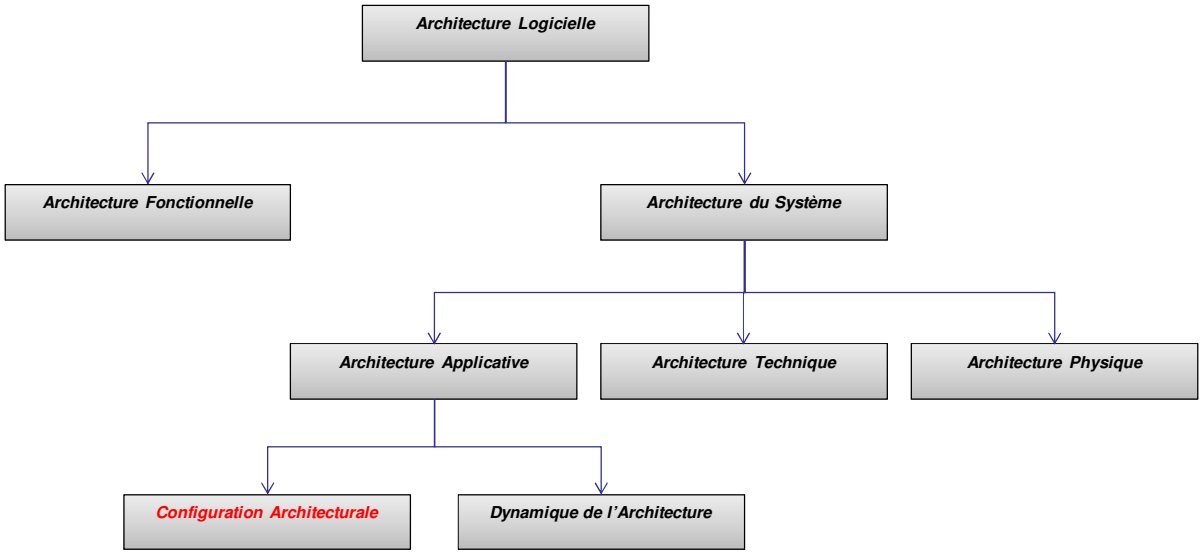
- nous n'aborderons que le type d'architecture à base de composants.
- nous feront la conception de ces composants en utilisant les Designs Patterns lors de la réalisation des diagrammes de classes de chacun de ces composants.

2 La notion d'architecture logicielle

La définition d'une architecture est essentielle dans un projet.
Elle permet :

- de valider les choix faits avant la phase de réalisation des composants. On s'assure que l'organisation proposée permet bien d'assurer ce qu'on attend du système, tel que défini dans la spécification.
- •d'organiser le projet: structuration du travail, coût et délais.
- de préparer l'intégration à partir des produits élémentaires à réaliser et des « Contrats » (interfaces) entre équipes réalisatrices.

La démarche d'architecture se décompose en différentes parties.



La **Configuration Architecturale** est la décomposition d'un Système d'Information en composants logiciels.

Dans beaucoup de projet, l'architecture du logiciel se résume à cet aspect là. Les autres aspects étant plus ou moins abordés, souvent sur le tas.

Dans ce cas, par abus de langage, on parle de cet aspect comme étant l'Architecture Logicielle.

Dans le cadre du cours :

Nous couvrons une partie de l'Architecture Technique dans la mesure où nous imposons le choix de la technologie utilisée : Conception Objet (java), Communication (RMI[1]), MOM[2] (JMS[3]).

Nous couvrons une partie de l'Architecture Physique dans la mesure où nous faisons le choix des machines à utiliser, de la répartition des "environnements d'exécution" (JVM) et de leurs répartitions sur le réseau.

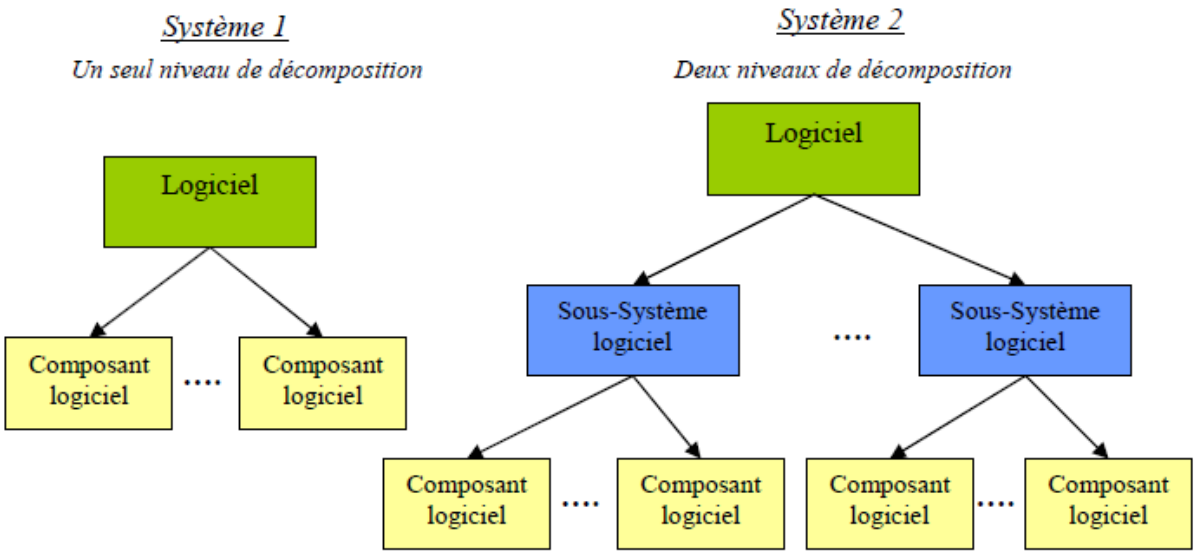
[1] Remote Method Invocation
[2] Middleware Oriented Message
[3] Java Message Service

2.1 Les constituants de la Configuration Architecturale

L'élément de base d'une architecture est le **composant**. Un composant (logiciel) est une unité de composition logicielle, exposant des interfaces bien spécifiées, et susceptible d'être déployé de manière indépendante.

Une architecture est une structure **récursive** (on dit parfois, très improprement, « fractale ») : dès qu'un logiciel a une certaine taille, il est en général décomposé en un premier niveau de constituants, appelés « sous-systèmes logiciels », eux-mêmes structurés en constituants plus fins, comme illustré par la figure ci-dessous. Dans la pratique, on peut aller ainsi, pour des grands ou très grands systèmes, jusqu'à 3 ou 4 niveaux d'imbrication (avec des sous-systèmes constitués de « sous-sous-systèmes », etc.).

Nous emploierons ici le terme général de « **constituant** » pour désigner aussi bien un composant logiciel (éléments terminaux de décomposition) qu'un sous-système logiciel.



De manière générale, un composant logiciel peut être :

- un logiciel développé spécifiquement durant le projet de développement;
- un logiciel réutilisé depuis un autre projet (ou système);
- ou encore un progiciel sur étagère que l'on intègre.

2.2 Le Composant d'une architecture logicielle

Les composants peuvent être de granularités très variables allant d'une simple **classe**, jusqu'à un ensemble complexe de **modules**, ou un **progiciel** complet.

Le plus souvent, un composant sera un regroupement cohérent de classes « packagées » dans un module de programmation.

Il faut noter que des éléments tels que les **fichiers** de données, tels divers fichiers de configuration, ou encore les **bases de données** sont également des composants, et qu'ils doivent donc être intégrés dans la description de l'architecture logicielle.

Le Composant est caractérisé par :

- son identification (il est important d'être précis et synthétique)
- son rôle
- ses propriétés externes (**interface** d'entrée et de sortie)
- ses relations avec les autres composants (topologie) : les **liens** entre ces composants sont typés (nature) dans un premier temps et sont ensuite précisés sous la forme de **connecteurs**.
- la liste des **fonctionnalités** fournies (ou son rôle (résumé))
- Les propriétés **non fonctionnelles.** Une propriété « non fonctionnelle » est tout ce qu'on peut attendre d'un système ou d'un constituant, et que l'on ne peut pas exprimer directement comme une fonctionnalité. On exprime donc comme une propriété (par exemple performances en temps de réponse, fiabilité, de sécurité, capacités liées aux invocations concurrentes, etc.)

2.3 Le diagramme de la Configuration Architecturale

Pour mener à bien la description structurelle d'une architecture logicielle (configuration architecturale), on représente un système sous la forme d'un assemblage de composants et de liens entre ces composants.

Un lien entre deux composants est avant tout un lien fonctionnel. Il décrit un échange d'information entre les deux composants. Il traduit souvent un sens de "circulation" de l'information.

Ce type de diagramme est un dérivé du diagramme de communication UML :

Définition : "*Les diagrammes de communication représentent une vue dynamique du système.*

Ils présentent un ensemble de rôles joués par des objets, ici des composants*, dans un contexte particulier, ainsi que les liens entre ces objets*.

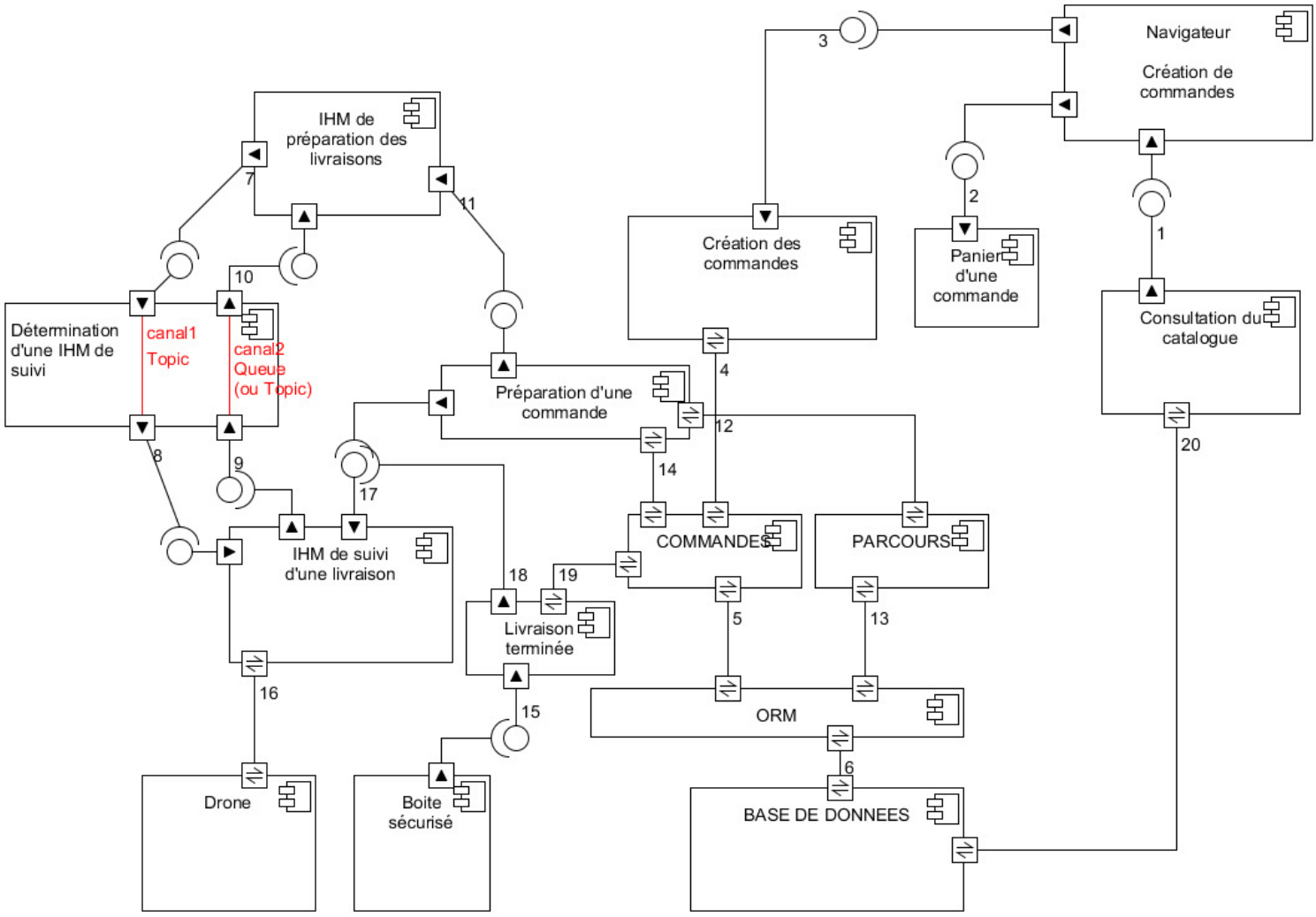
Ils insistent plus particulièrement sur la structure spatiale" : localisation des composants dans les logiciels.

L'ajout d'une dimension temporelle requiert la définition de numéros de séquence pour les messages" ou appels*."*

Quand on affine la nature, la localisation, le rôle des composants, on embarque dans la description des éléments de l'Architecture technique et de l'Architecture Physique :

- les liens sont locaux à un composant logiciel (même JVM). Cela se traduit par l'appel de méthodes de classe mais, si cela est bien fait sous la forme de Designs Patterns connus (MVC, Factory, …)
- les liens sont distants entre les composants logiciels. Cela se traduit par le choix d'un Design Pattern de communication (connecteurs).

Exemple d'un diagramme de Configuration Architecturale :



2.4 La Dynamique de l'architecture

Il y a deux aspects de la dynamique d'une architecture logicielle :

- la dynamique entre les constituants de l'architecture qui correspond à la logique de l'exécution globale du logiciel
- la dynamique interne de certains des constituants;

Le premier aspect consiste pour chaque cas d'utilisation du système, d'expliciter les séquences d'invocations des différents composants.

Cela permet de :

- vérifier que les grandes fonctionnalités du système sont bien **faisables** à travers son architecture.

Le deuxième aspect consiste à s'intéresser à l'exécution interne des composants :

- relatif à son comportement autonome, en parallèle, et donc à son impact sur le reste du système durant le cycle de vie du système,
- relatif à son comportement interne face à son utilisation par plusieurs autres composants en parallèle

En résumé, il s'agit de mettre en évidence la **coopération** des composants entre eux. On se demandera par exemple si les opérations offertes par le composant sont invocables ou non de manière concurrente par plusieurs composants appelants, si un ordre particulier doit être respecté dans ces invocations, etc.

2.5 L'Architecture Technique et Physique

Nous avons évoqué pour l'instant l'architecture logicielle, qui concerne la structure du logiciel dit « applicatif », propre au système à développer. C'est une architecture logique que l'on trouve au plus haut niveau d'abstraction. Il reste à réaliser techniquement cette vue logique : d'où l'Architecture Technique et l'Architecture Physique.

De plus, un système développé en réponse à un besoin comporte des éléments de différentes natures :

- 1) le logiciel applicatif, mais aussi
- 2) l'infrastructure technique (le ou les environnements logiciels nécessaires à l'exécution des composants applicatifs, aussi appelée « plate-forme » logicielle), et enfin
- 3) l'infrastructure matérielle (machine ou réseau de machines équipées de leur système d'exploitation).

Ainsi, comme déjà indiqué, une architecture applicative repose en général sur une infrastructure technique, sur laquelle s'exécutent concrètement les composants applicatifs. L'organisation des éléments ainsi présents constitue l'architecture technique.

Ces éléments reposent eux-mêmes sur un ensemble de dispositifs physiques, principalement des postes de travail, serveurs, réseaux, dispositifs mobiles, etc. munis de leurs systèmes d'exploitations et pilotes matériels, mais aussi tous autres éléments matériels (capteurs, boîtiers électroniques divers, …). L'ensemble de ces éléments constitue la plate-forme matérielle (dite aussi « infrastructure physique »).
Leur organisation d'ensemble constitue l'architecture physique.

Ainsi, in fine, un composant logiciel n'aura de réalité concrète que placé dans un environnement d'exécution, sur une machine physique.
Quand l'environnement d'exécution et un environnement générique, on parle de **déploiement sur l'infrastructure** sur laquelle vont s'exécuter les composants (logiques).

Il s'agit essentiellement d'indiquer :

- dans quel environnement logiciel s'exécute chaque composant = **Architecture Technique**
- sur quelle machine = **Architecture Physique**

Dans le cadre de notre cours :
Architecture Technique =

- Langage de programmation = Java
- Infrastructure de communication = RMI
- Conteneur de déploiement = JVM

3 Les types d'architectures

Il existe différents types d'architecture, c'est-à-dire des modèles d'architectures récurrents (ou DP d'Architecture) plus ou moins bien adaptés aux besoins fonctionnels.

La difficulté est le choix d'utiliser tel type d'architecture plutôt qu'un autre. L'objectif de ce cours n'est pas de faire ce choix car nous nous intéresserons par la suite qu'aux architectures dites "à base de composant".

Etant donné qu'il est très difficile de changer d'architecture en cours de réalisation, il est important de prendre le temps de faire cette conception architecturale.
Il est à noter que souvent, on réalise une maquette ou un prototype qui représente au mieux l'ensemble du futur système permettant de s'assurer des choix réalisés tant sur la capacité que sur les performances.

Les différents types d'architecture sont (liste non exhaustive) :

- architectures en couche
- architectures à base de composant (exemple techno : **JEE**)
  - **architecture 2-tiers, 3-tiers, 4-tiers**
  - **architecture distribuée**
  - **architecture MVC (dont Internet)**
- **architecture MOM**
- architecture Web Services
- architecture REST FULL
- architectures orientées Web (WOA)
- architecture orientées services (SOA)
- architecture Cloud Computing

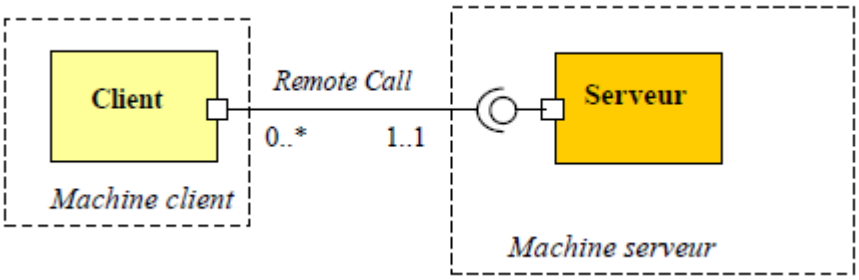
En rouge les types d'architecture qui serviront de support à notre cours pour la conception de Système d'Information.

3.1.1 Les architectures 2-Tiers

Simple architecture dans lequel il n'y a que les tiers : les clients et le serveur.

Dans ce type d'architecture le serveur est un composant logiciel unique qui assure à la fois le traitement des requêtes provenant des clients, la gestion des données et leurs persistance en base de données.

Cela concerne les client-serveur (client lourd)

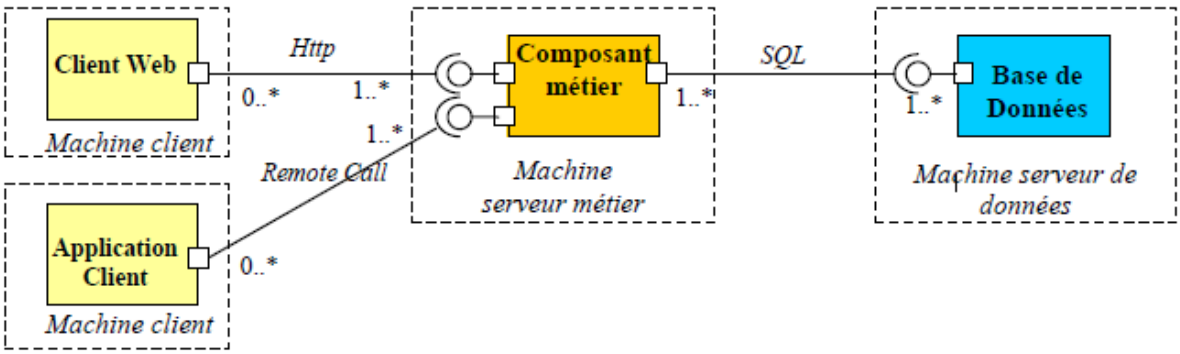


Exemples :
- serveur de base de données
- serveur de plusieurs clients lourds

Le serveur exécute les transactions en assurant le respect de l'intégrité des données.

Cas particulier quand tous les composants sont des serveurs : architecture peer-to-peer.

3.1.2 Les architectures 3-Tiers



Remarque : 1 ou plusieurs composants métiers (ne communiquant pas entre eux)

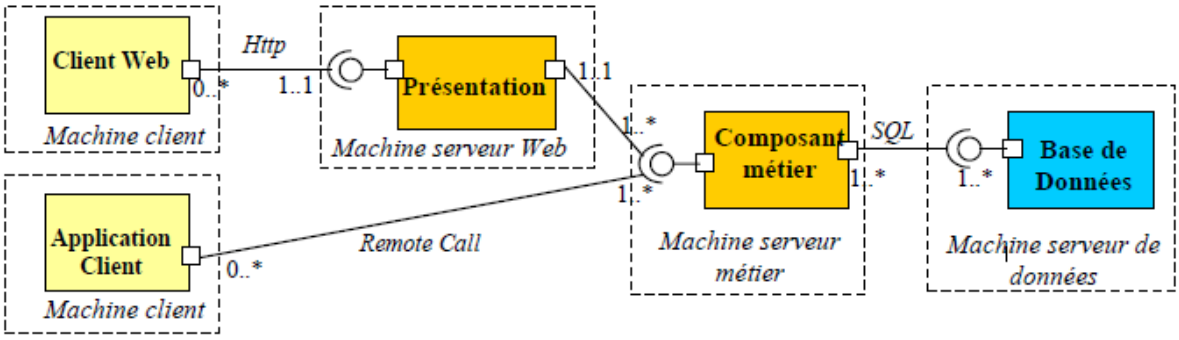
On a une organisation en trois tiers :

- Le tier **client** exécute les interfaces (boundary objects) permettant d'interagir avec l'utilisateur (fenêtres, formulaires, pages Web, etc.) ;
- Le tier logique applicative dit « métier » comporte tous les composants assurant la logique métier et la manipulation des entités du domaine requises par l'application ;
- Le tier persistance réalise le stockage des données persistantes dans une ou plusieurs bases de données par exemple.

Il a y autant de clients que nécessaire. Ceux-ci peuvent être exclusivement des clients légers (architecture 3-tiers à client léger), exécuté sur un navigateur Web, ou des clients lourd (composants déployés sur la machine client physique), ou encore une combinaison des deux.

La mise en œuvre de ce type d'architecture se fait notamment en utilisant une couche logicielle prédéfinie appelée le ORB (Object Request Broker).

3.1.3 Les architectures 4-Tiers



Dans cette architecture, on introduit un tier, dit tier « Web » ou encore tier « Présentation » entre le tier client et le tier logique applicative, qui assure :

- La fourniture des contenus statiques : contenu (HTML) à afficher par le client ;
- La génération de la présentation du contenu dynamique (par exemple JSP, servlets, etc.) ensuite affiché en HTML par le client.

Le tier logique applicative ne réalise que le coeur des traitements de l'application.

Il y a autant de clients que nécessaire. Ceux-ci sont en général des clients légers. L'architecture peut intégrer des clients lourds, mais alors, ceux-ci communiquent alors directement avec le tier applicatif comme représenté sur la figure.

3.1.4 Les systèmes répartis (ou distribués)

Le type d'architecture des systèmes répartis est l'ancêtre du type d'architecture à base de composant (voir ci-après).

Le type d'architecture des systèmes répartis a poussé, surement à l'excès, le principe de la décentralisation des ressources informatiques. Elle nécessite l'ouverture de nombreux ports de communication.

Or avec les nouvelles technologies issues du monde Internet, ce type d'architecture est tombé en désuétude. Il reste encore présent sur les réseaux intranet.

RMI (Remote Method Invocation), CORBA (Common Object Request Broker), et DCOM (Distributed Component Object Model), sont tous des implémentations de système réparti.

"Un système réparti est un système d'information dans lequel les ressources ne sont pas centralisées". Ces ressources sont notamment :

- les moyens de stockage (données, fichiers)
- la charge CPU
- les utilisateurs
- les traitements

Les principes de base d'un système réparti sont :

- une indépendance de la localisation géographique des ressources ce qui permet de mettre en place une politique de répartition de la charge globale du système efficace
- une difficulté à obtenir un état global stable
- une adaptabilité aux contraintes du réseau

- une répartition de la charge globale du système
- les techniques de réplication des ressources et la mise en place d'une persistance des données névralgiques permettent une meilleure disponibilité des services en cas de panne de certains d'entre eux

Il existe deux modèles des systèmes répartis.

- *modèle des processus communicant par envoi de message*
  - *modèle fondateur (modèle standard)*
  - *basé sur le parallélisme des communications*
  - *communication synchrone ou asynchrone*
  - *communication point à point ou par diffusion*
  - *modèle CSP (Communicating Sequentiel Processes) et CCS (Calculus of Communicating Systems) (http://fr.wikipedia.org/wiki/Communicating_sequential_processes)*
- modèles fondés sur la notion d'OBJET (ex : RMI, CORBA, DCOM)
  - très répandus par l'usage des technologies objets dont les LOO
  - masquer le phénomène de communication (et donc de répartition)
  - les traitements : procédure accessible à distance (RPC)
  - les données : mémoires partagées (ex: les objets)
  - difficulté : éviter une trop forte synchronisation des accès à ces mémoires partagées
  - compromis entre une sémantique "centralisée" et des performances acceptables

La notion d'Objet Distant est au cœur de l'architecture répartie.
Un Objet Distant (OD) représente une ressource mémoire et une ressource de traitements distants qu'il est possible d'utiliser en tout point du réseau et indépendamment de sa situation géographique (mise en place d'un service de nommage).

Comme nous le verrons plus loin dans le cadre des DP, tout OD est accessible à travers son **Proxy de Communication**.

Un « client » obtient le Proxy de communication de deux façons :

- En consultant un annuaire qui contient l'adresse IP et le Port de communication de l'OD via son nom logique, avec lequel il est possible de construire le Proxy de communication (lookup).
- En demandant à un OD de plus haut niveau (par ex un Factory d'OD), le Proxy de communication des OD qu'il gère (le Proxy de Communication est un objet comme un autre qui peut être mémorisé dans n'importe quelle classe).

3.1.5 Les Architectures à base de Composants

3.1.5.1 Les principes de base

Le fort besoin de réalisation des applications internet, a conduit, dès la fin des années 90, la réalisation des **Frameworks**.

Un Framework (source Wikipédia https://fr.wikipedia.org/wiki/Framework) est un ensemble d'outils et de composants logiciels organisés conformément à un plan d'architecture et des patterns, l'ensemble formant ou promouvant un « squelette » de

programme. Il est souvent fourni sous la forme d'une bibliothèque logicielle, et accompagné du plan de l'architecture cible du framework

Un framework est conçu en vue d'aider les programmeurs dans leur travail. L'organisation du framework vise la productivité maximale du programmeur qui va l'utiliser.

Exemples : JEE, Spring, .NET, Zend (PHP), Ruby On Rails, Struts, Cocoa (Apple), Dojo, Eclipse RCP, Hibernate, Symfony (PHP).

On conçoit ainsi le tier métier d'une application sous la forme d'un **assemblage de « composants »**.

Les composants sont répartis sur un réseau et s'exécutent dans des **Structures d'Accueil**.

Quand le réseau est un réseau est local ou Intranet, la communication est réalisée en utilisant les technologies des systèmes répartis (RMI, CORBA, DCOM).

Quand le réseau est Internet, la communication est réalisée en utilisant les technologies Web-Services, REST, WebSocket (NodeJS>Socket.IO).

Un composant est vu et développé comme un **objet**, mais avec certaines limitations par rapport à l'ensemble des facilités offertes par la programmation objet. Ces objets peuvent être **déployés** sur différentes machines serveur. Ils peuvent alors **interagir** entre eux de manière simple, et dans le respect **d'interfaces standard**.

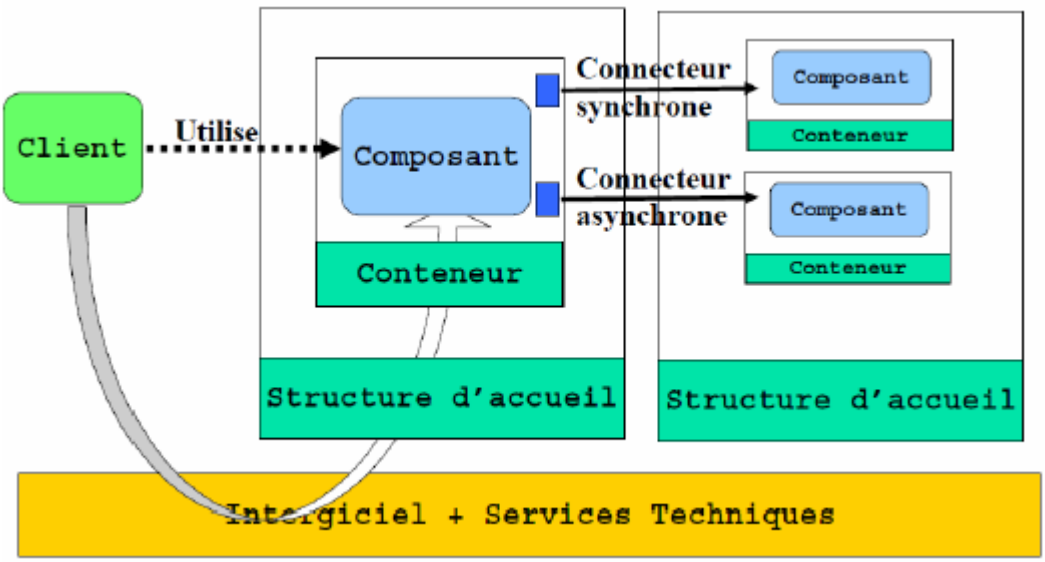
Dans cette optique, un **composant** est un élément logiciel développé selon certaines spécifications, offrant un ou des services prédéfinis, et capable de communiquer simplement avec d'autres composants ou avec des applications distantes.

Ainsi, l'écriture d'un composant se focalise sur la partie réellement applicative, les aspects « techniques » (c'est-à-dire le relais des appels entre composants locaux ou distants, la prise en compte de la sécurité, de la concurrence d'accès, des transactions, etc.) étant à **la charge de l'infrastructure**.

Ainsi, les objets sont facilement déployables sur un réseau de machines, potentiellement réutilisables, puisque non spécifiques de la manière particulière dont ces services sont implémentés. **Il y a bien séparation entre le code proprement applicatif et les mécanismes techniques de base**.

Le modèle d'architecture résultant d'un tel principe est synthétisé sur la figure ci-dessous :

Soit un Client qui fait l'appel d'un Service de la couche métier dont la localisation a été déterminée par les Services Techniques.



Ce schéma montre que l'exécution d'un "**composant"** s'appuie sur une unité d'exécution, commune à plusieurs composants, appelée « **conteneur** », reposant elle-même sur une structure d'accueil spécifique.

Il peut y avoir un ou plusieurs conteneurs par machine serveur physique.

Il peut y avoir plusieurs "composants" dans un même Conteneur.

Remarque : Ici le terme de "composant" est ici employé dans un sens plus spécifique que dans le contexte général des architectures logicielles. Les « composants » dont il est question ici sont des composants fonctionnels orientés métier.

Composant d'Architecture (Technique) = Composant métier + Conteneur + Structure d'accueil + Services techniques.

Un conteneur assure:

- La **localisation** et la **résolution des dépendances entre composants**.

  Ceci permet à un composant de localiser un autre composant et d'invoquer ses méthodes. Le conteneur donné « gère » ainsi les composants dans lequel ceux-ci sont déployés. Ainsi, le conteneur peut relayer convenablement les appels de service entre les composants ;

- **Le lien vers les services de base**, permettant à un composant de s'appuyer sur des services de l'infrastructure effective choisie pour le déploiement, sans lien en dur avec celle-ci ;

- **D'autres services**, tels par exemple la répartition de charge, la gestion du cycle de vie, avec de multiples instances concurrentes en exécution ou en attente. Tout le code nécessaire à ces « passivations » temporaires puis ces « réactivations » peut faire partie des services que peut fournir un conteneur.

L'environnement logiciel sur lequel peuvent être exécutés différents composants et conteneurs est souvent appelé « **serveur d'application** ».

Le Composant :

- est **réutilisable** dans d'autres applications, d'autres conteneurs. C'est une caractéristique importante de qualité logicielle
- est à **usage multiple** : c'est-à-dire que le même composant peut être invoqué dans différents cas d'utilisation

- est **composable** avec d'autres composants
- assure une **encapsulation** : son implémentation est transparente à travers son interface d'utilisation
- peut être **indépendant** de son déploiement dans un conteneur

Les composants prennent la forme des **classes d'objets**, au sens du langage de programmation utilisé (en général Java), en respectant certaines conventions, avec des informations complémentaires qui sont données sous la forme de **fichiers de configuration** (hors du code) ou encore **d'annotations** placées dans le code.

3.1.5.2 L'injection de dépendance

L'injection de dépendance est un principe de base des Architectures à base de composants.

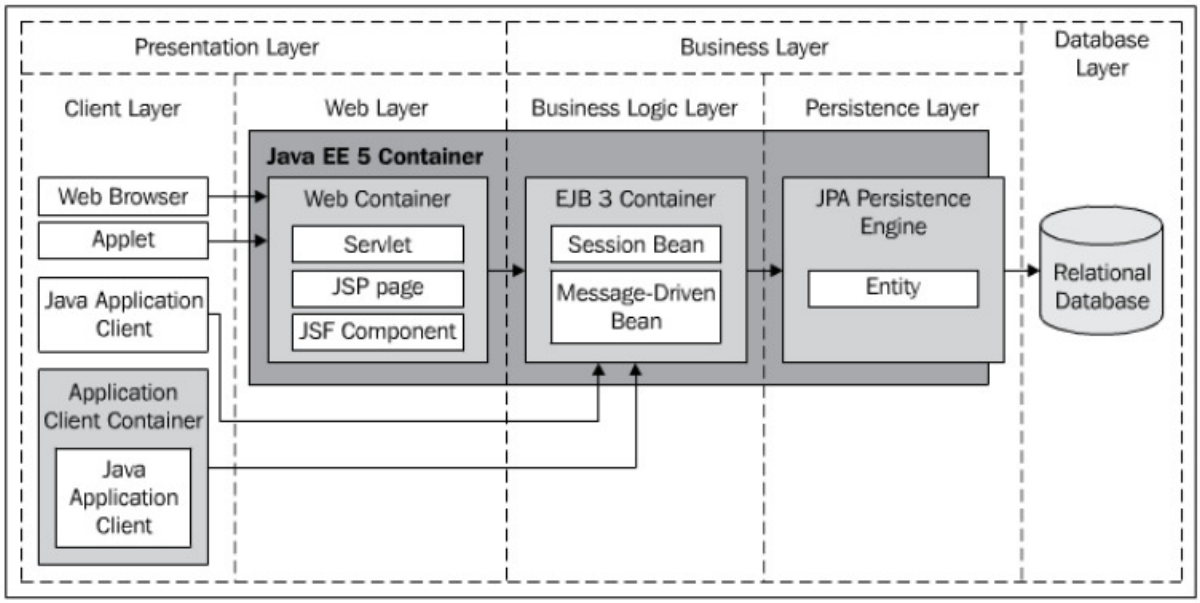
Elle permet de **lier les composants** entre eux sans écrire de code (pas de compilation du code source), c'est-à-dire de **relier dynamiquement l'invocation d'un composant à son implémentation concrète**.

Comme nous le verrons plus loin dans le chapitre 5, en termes de design pattern, l'injection de dépendance est : l'association du pattern « **inversion de contrôle** » (IoC, Inversion of Control) et de la « **délégation** » à travers une interface abstraite.

3.1.5.3 Java 2 Enterprise Edition (JEE)

Java 2 EE, ou encore J2EE (Java 2 Enterprise Edition) est un standard élaboré par Sun à l'origine (maintenant ORACLE), supporté par un large consortium d'éditeurs, définissant **un modèle de développement d'applications distribuées N-tiers**, basé sur des composants : Apache Struts, Spring MVC, Tapestry, JBoss, …

Le modèle architectural de Java EE est un **modèle N-tiers**, représenté sur la figure ci-dessous.



Le modèle d'architecture de J2EE comprend ainsi **quatre tiers**.

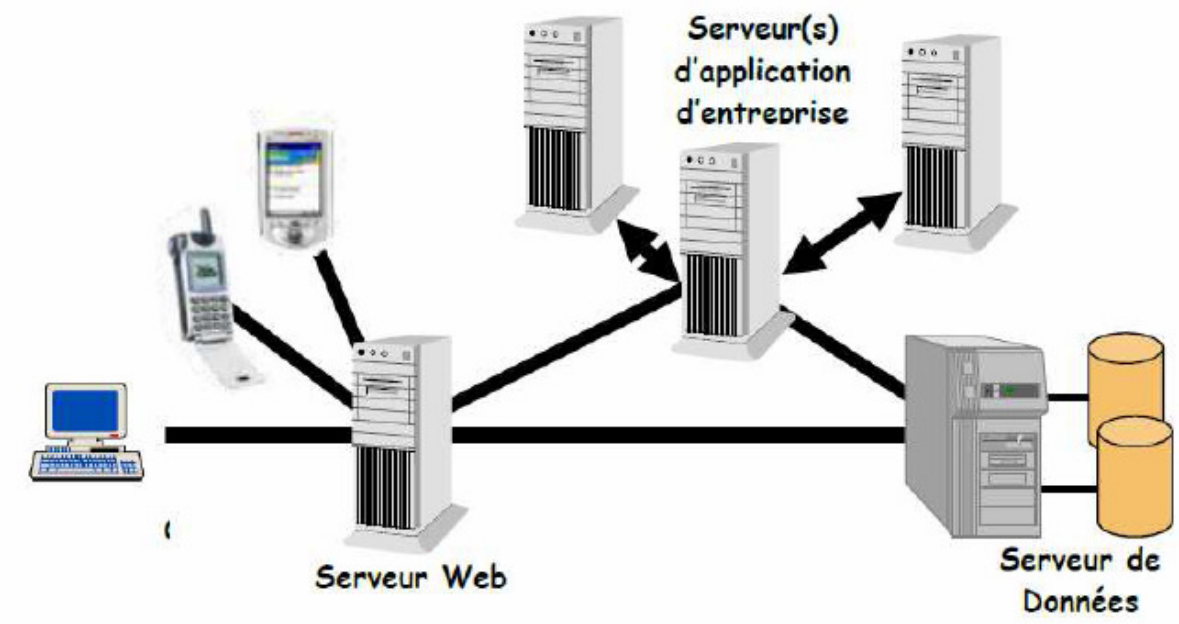
- **Le tier « client »**, situé sur les postes de travail clients, responsable de l'interaction avec les usagers. Ce tiers travaille avec un navigateur Internet sur la base de pages HTML, et/ou avec l'exécution d'applets dans le navigateur, ou encore avec des applications Java classiques (client lourd) ;
- **Le tier « Web »**, qui assure la traduction des requêtes de l'usager en appels convenables adressés à la partie métier, et qui génère les pages HTML à destination du navigateur. Ceci peut être développé grâce à différentes technologies telles que les **Servlet**, les pages **JSP**, ou encore les composants **JSF**[4] (Java Server Faces) ;

  Note : précisons bien quelques points. Si nous considérons, comme on le fait d'ordinaire, que le code d'une IHM bien conçue est structuré en trois parties : 1) la présentation (les aspects visuels de l'IHM), 2) le contrôle du dialogue (la dynamique propre à l'interface interactive, non liée aux règles métier), et enfin 3) l'interface avec l'application). Dans le cas d'une interface simple de type navigateur, c'est ce tiers Web, joint au navigateur client, qui assure en fait les aspects liés à la « présentation ». Ce tiers assure également le « contrôle du dialogue », ainsi que l'interface avec le tier métier.

  C'est pourquoi il faut éviter d'appeler comme on le fait parfois le tier client tier « présentation », ce qui est inexact lorsque les pages Web affichées sont générées sur le serveur Web.
- **le tier « métier »** (business tier, ou middle tier), dans lequel la logique métier et les entités correspondantes sont gérées et manipulées. Dans l'architecture J2EE, ce tier est implémenté par assemblage de composants métiers appelés Java Beans, comme on le verra dans la suite ;
- **le tier « base de données »** (ou « Information », ou encore « Enterprise Information tiers »), qui regroupe les différents supports de persistance et les sources de données. On peut y trouver des applications existantes que l'on souhaite laisser en service et utiliser telles quelles avec, au besoin, rajout d'une nouvelle interface d'appel adaptée.

Sur le plan de l'architecture matérielle :

[4] Java Server Faces (JSF) est une technologie dont le but est de proposer un framework qui facilite et standardise le développement d'applications web avec Java.
Plusieurs outils commerciaux intègrent déjà l'utilisation de JSF notamment Studio Creator de Sun, WSAD d'IBM, JBuilder de Borland, JDeveloper d'Oracle, ...



- **Le tier client** est formé des postes de travail des usagers, ou d'autres dispositifs légers (téléphones, ..) qui peuvent être en très grand nombre et communiquent (soit par le biais d'Internet ou du réseau local) avec une machine serveur Web ;
- · **Le tier Web** est placé sur une machine serveur ;
- · **Le tier métier** peut être sur une machine serveur unique (de manière générale la même que le tier Web). Mais il peut être également réparti sur plusieurs serveurs d'entreprise ;
- · **Le tier bases de données** est placé sur un ou plusieurs machines serveurs de données.

On parle souvent de **trois tiers** en considérant que le tier Web et le tier métier sont placés sur
la même machine serveur , ce qui est souvent le cas dans la pratique.

3.1.6 Les Architectures MOM

MOM = Middleware Oriented Message

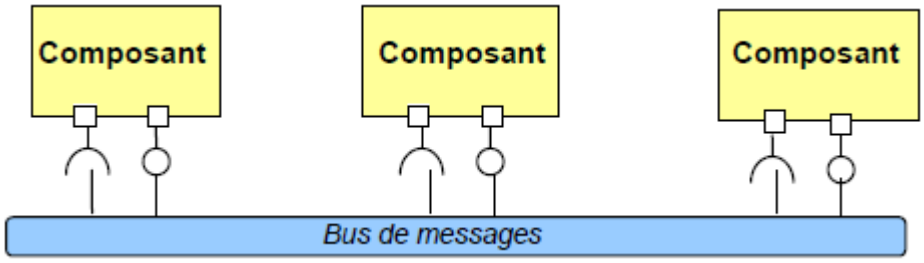
Ce style architectural utilise comme mode principal d'interaction entre les composants **l'appel indirect et implicite**.

Plus précisément, les appels sont exécutés de manière **asynchrone** en réponse à des **notifications d'événements**.

Il existe plusieurs variantes de ce style, mais la principale variante est le style **Publish & Subscribe** qui repose sur un connecteur « bus de message » de haut niveau supportant le mécanisme du même nom (ce bus est dans la pratique fourni par exemple par un middleware orienté messages).

De telles interactions indirectes (les composants ne se « connaissent » pas) apportent des propriétés très intéressantes en matière d'adaptation (ajout de nouveaux composants sans modification aucune des autres) et d'extensibilité (« scalability »).

C'est le connecteur qui implémente les mécanismes d'abonnement et de notification. C'est donc auprès de ce connecteur que le composant s'abonne (sur un "topic") et c'est du connecteur qu'il reçoit l'événement de notification exécutant la callback.



Avantage : Couplage faible entre composant autorisant l'évolutivité

4 Conclusion

Dans la mise en œuvre de tous ces différents types d'architecture, de nombreux Design Pattern sont utilisés.

Ces DP sont utilisés dans l'implémentation de tous les Framework que les développeurs utilisent aujourd'hui et qui permettent de développer de tels types d'architecture (JEE, WebService, Restfull, Corba, CMS, Micro-Services (Kubernetes)…)

A travers la découverte des Designs Patterns c'est aussi de connaitre une partie de ce qui se cache en-dessous des Frameworks.

Il est donc important de se mettre dans cet état d'esprit. Plus de framework !! Vous devez tout faire vous-même, à la main, en pensant "Design Pattern" 😊.