I - Présentation de l'architecture logicielle

1. Qu'est-ce-que l'architecture logicielle ?

L'**architecture logicielle** **décrit d'une manière symbolique et schématique les différents éléments d'un ou de plusieurs systèmes informatiques, leurs inter-relations et leurs interactions .**

Les éléments de l'architecture logicielle sont généralement appelés "***composantes logicielles***".

Le **modèle d'architecture**, *produit lors de la phase de conception* **décrit comment un système informatique doit être conçu de manière à répondre aux spécifications**.

1.1. Principaux enjeux de l'architecture

L'architecture logicielle constitue l'***épine dorsale*** d'un ***système informatique*** moderne.

Sur ces fondations, toute une logique métier sera construite et l'ensemble aura ou pas les qualités déterminantes suivantes :

- **Application** complexe/évoluée **fiable et fonctionnant bien** (sans bug , sans problème de performance, avec bonne ergonomie) ?
- **Maintenance et évolution aisées** (sur le court , moyen ou long terme) ?
- **Pas trop coûteux** (développement , fonctionnement, ressources optimisées , ...) ?

1.2. Les points de vue et les influences

- **Point de vue "métier/fonctionnel/utilisateur"** *(ergonomie , enchaînement des écrans fonctionnels , performances, ...)*
- **Point de vue "utilité/rentabilité" pour moa** *(valeur ajoutée , R.O.I. , intégration avec SI existant ou à venir, ...)*
- **Point de vue "développement/maintenance/évolution" pour moe**
- …

NB : ces différents *points de vue sont quelquefois un peu antagonistes* et des ***compromis*** ou des ***choix*** (selon priorités) devront être effectués.
Exemple : une voiture est soit "très spacieuse et difficile à stationner" , soit "petite et simple à stationner" mais peut difficilement être "très spacieuse et simple à stationner" .

Autrement dit :

- "***On ne peut pas être contre la vertu***" mais "***on ne peut pas avoir toutes les vertus***" .
- L'architecture logicielle se situe au niveau d'une **confluence d'influences** (points de vue des utilisateurs , des développeurs , ….)

L'architecture influence la structure organisationnelle de développement et réciproquement.
L'architecte et la moe peuvent éventuellement influencer le client (sacrifier quelques fonctionnalités pour d'autres qualités).
Quelques influences (nouvelles tendances , vision précise d'un client) font évoluer l'expérience de l'architecte.

1.3. Le rôle de l'architecte

- Déterminer les réels besoins et contraintes logicielles *(volume de données , nombres d'utilisateurs simultanés , sécurité et fiabilité exigées , ...).*
- Tenir compte des moyens/ressources disponibles *(budget , ...)*
- Elaborer une architecture globale adéquate (à base d'assemblage de technologies idéalement compatibles et éprouvées)
- Documenter clairement cette architecture
- Accompagner les intervenants (développeurs , DBA, ...)

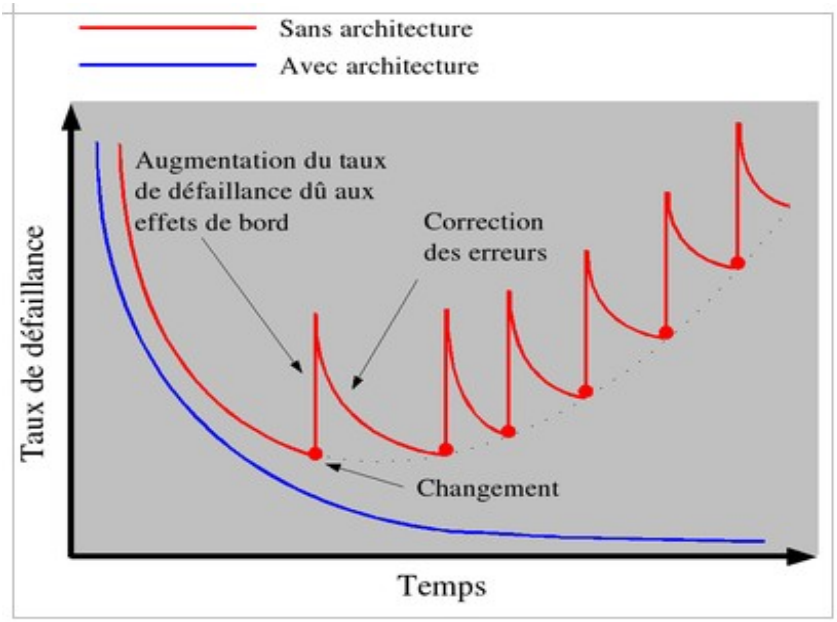
1.4. Particularité de la conception architecturale d'un logiciel

La phase de conception logicielle est l'équivalent, en informatique, à la phase de conception en ingénierie traditionnelle (mécanique, civile ou électrique); cette phase consiste à réaliser entièrement le produit sous une forme abstraite avant la production effective. Par contre, la nature immatérielle du logiciel (modelé dans l'information et non dans la matière), rend la frontière entre l'architecture et le produit beaucoup plus floue que dans l'ingénierie traditionnelle.

1.5. Erosion (à anticiper) de l'architecture logicielle

Dégradation du logiciel dans le temps (suite à des changements/évolutions) :
Une architecture faible ou absente peut entraîner de graves problèmes lors de la maintenance du logiciel. En effet, toute modification d'un logiciel mal architecturé peut déstabiliser la structure de celui-ci et entraîner, à la longue, une dégradation (principe d'entropie du logiciel). L'architecte devrait donc concevoir, systématiquement, une architecture maintenable et extensible.

La gestion des changements est primordiale. En effet, il est implicitement considéré que les besoins des utilisateurs du système peuvent changer et que l'environnement du système peut changer. L'architecte a donc la responsabilité de prévoir le pire et de concevoir l'architecture en conséquence; la plus maintenable possible et la plus extensible possible.



2. Les attributs de qualité

2.1. Critères déterminants

- **Disponibilité** (ex : 24h/24 , jours ouvrés)
- **Ergonomie** intuitive et efficace
- **Souplesse/évolutivité**
- **Performance** (bon temps de réponse même en cas d'utilisations simultanées)
- **Intégration simple** d'une nouvelle application dans le SI existant
  et intégration simple de diverses technologies complémentaires

2.2. facteurs déterminants

- **Anticipations** (évolutions prévues ou pré-senties , …).
- **Optimisations** (mesures , prise en compte de celles-ci , comparaisons/extrapolations , …).
- **Modélisation** (pour prendre le recul nécessaire).
- **Identifications** précises des **contraintes**.
- **Expérience** (être conscient des difficultés à surmonter , connaître les remèdes aux problèmes , savoir estimer/chiffrer le temps de développement)
- **Rigueur**/Sérieux
- Gestion des **impondérables** (marge prévue, gestion du stress,...)
- **Environnement technique** (intégration continue , …)
- **Connaissances techniques** (subtilités/limitations des API , frameworks , compatibilités et incompatibilités)

2.3. Enjeux métiers et techniques

Enjeux techniques de l'architecture logicielle :

- Maintenance et évolution simples/rapides ou bien difficiles/longues/coûteuses
  et par conséquence utilisation effective ou pas de l'application sur le long terme.
- Stabilité du comportement de l'application sur le long terme (fiabilité , encaissement des pics de charge , non régression en cas d'ajout fonctionnel, ...)
- Crédibilité du savoir-faire de la MOE vis à vis de la MOA

Enjeux métiers de l'architecture logicielle :

- Avantage concurrentiel pour toutes les parties (MOE, MOA, ...)
- Pertes ou Gains

2.4. Catégories d'attributs de qualités

- **Visibles ou pas à l'exécution** par les utilisateurs (ex : "ergonomie" et "performance" = visibles mais "modifiabilité" et "testabilité" invisibles).
- **Purement techniques** *(ex : performance)* ou plutôt "**métier/fonctionnel**" *(ex : exactitude des algorithmes et règles de gestion)*
- **dû à l'architecture** *(par conséquences)* ou **intrinsèque** à l'architecture elle même *(ex : intégrité conceptuelle , complétude , faisabilité).*

2.5. Les principaux attributs de qualités

Attributs de qualité	*Caractéristiques*	*Mesures/précautions*
Disponibilité	Application momentanément indisponible (changement de version, évolution, panne maintenance technique, sauvegarde, ….). **taux dispo =** **temps dispo** **----------** **temps dispo + temps indispo**	Cluster avec mécanisme "fail/over" , redondance , limiter les conséquences des inter-dépendances chaînées.
Souplesse/évolutivité ("***Modifiabilité"***) (maintenabilité , extensibilité , ….)	Application rigide (difficile à faire évoluer) ou bien souple (extensible/reconfigurable)	Bien utiliser les concepts objets et l'architecture SOA. Bien dimensionner le nombre de couches logicielles (ni trop , ni pas assez). Bien identifier/modéliser les besoins fonctionnels (présents et futurs pré-sentis) Tenir compte du schéma directeur de l'urbanisation
Performance	"Bons ou mauvais **temps de réponse ou de traitement**" et "**Nombre de réponses calculées/renvoyées par unité de temps**" (débit /throughput) pour une bonne gestion de la charge (utilisateurs simultanés)	Mesures des temps d'exécutions (ihm , services , SQL , …) et optimisations. Tests de charge (requêtes simultanées) . Optimisation du trafic réseau (ajax, …). Utilisation de caches lorsque c'est possible. Surveillance régulière en production et redimensionnement des clusters. Séparation fonctionnelle pour parallélisme
Fiabilité	Peu de pannes / plantages	Détecter et éliminer les "fuites de mémoires" , bien gérer les exceptions , ...
Intégrité	Données toujours cohérentes ?	Intégrité référentielle dans SGBDR , GDR/.... en SOA , …. Tests des valeurs des données via dbUnit , …. Utilisation de l'encapsulation (données internes privées bien protégées, …).
Validité fonctionnelle (exactitude)	Selon cahier des charges (faire ce qui était demandé)	**Tests unitaires** pour chaque composants métiers (ex : service) et **tests d'intégration** (application complète avec interface graphique)
Sécurité	capacité du système à **résister à l'attaque**, et à **continuer à offrir un service normal aux usagers légitimes**	Pare-feu , DMZ , long mots de passes , cryptages , authentification forte , ...
Utilisabilité ***(convivialité ,***	Facilité d'apprentissage et d'utilisation du logiciel par les usagers	Soigner l'interface graphique : - intuitive et claire - efficace (raccourcis , …)

ergonomique)		- non ambiguë (opération bien réalisée ou messages d'erreurs explicites, …) - look agréable

2.6. Quelques autres attributs de qualité

Attributs de qualité	*Caractéristiques*	*Mesures/précautions*
Intégration simple (de niveau **SI**)	Intégrer dans des applications "cœur de métier" des services offerts par des applications annexes (de supports)	Utilisation d'ESB / EAI et bonne modélisation transverse , bonne urbanisation.
Intégration simple (des différentes technologies complémentaires)	Permettre à des composants développés séparément d'inter-opérer correctement	Via **interfaces** normalisées , via éventuels adaptateurs , ….
Interopérabilité	Import/export de documents, communications avec d'autres applications . Format compréhensible des messages échangés.	Utilisation de technologies "standard" et interopérables (ex : XML , HTTP) et/ou utilisation d'intermédiaires qui transforment les messages (ex : ESB)
Portabilité	Adaptabilité à différents environnements d'exécution (systèmes d'exploitations , serveurs d'applications , ...)	Utilisation de langage multi-plateformes (ex : "**Java**" ou …).
réutilisabilité	de certains composants et/ou de certains services et/ou de l'architecture	Choisir des technologies ayant à priori une certaines pérennités. Etudier les "dénominateurs communs" entre différents besoins et décomposer/spécialiser avec des choses génériques (facilement réutilisables) et d'autres choses plus spécifiques. Utiliser des "serveurs applicatifs ou frameworks" prédéfinis pour une réutilisabilté de l'architecture.
Testabilité (vérifiabilité) **[** *Testabilité = observabilité + contrôlabilité*]	Tests/vérifications/contrôles possibles , simples et déterministes	Prévoir systématiquement un test unitaire par composant logiciel important. Enregistrer pour re-jouer. Éviter les éléments incontrôlables (maîtriser les technologies et maîtriser la complexité du code) Incorporer quelques éléments de pilotage dans l'application (vérification ordre de marche, mesures/minitoring interne, …).
Efficacité	Exploitation optimale des ressources (CPU, RAM, ….)	Mesurer , optimiser , re-mesurer , comparer, ...
Autonomie	Dépendances limitées vis à vis d'autres systèmes. → favorise une haute	éviter les dépendances inutiles. Utiliser si possible des infrastructures "lights" (sans complexité/lourdeur

	disponibilité et une évolution moins contrainte.	inutile).
Composabilité , modularité	Pouvoir construire un système comme un assemblage de composants modulaires	Utilisation d'un framework d'intégration (ex : Spring ou CDI) basé sur de l'injection de dépendances.
Robustesse	capacité d'un logiciel à fonctionner lorsque l'on sort de ses spécifications.	Bien gérer les contrôles de saisies et les contraintes pour les valeurs possibles.
Coût (raisonnable?) de développement et fonctionnement		Chiffrages , estimations R.O.I.
Durée de vie projetée	"jetable" ou "à maintenir sur du long terme" ?	
Marché cible / ….		
...		
Intégrité conceptuelle	Cohérence par rapport au thème dominant quitte à abandonner/externaliser/déléguer certains éléments annexes	Bonne modélisation (ex : UML)
Complétude (et exactitude)	Répond bien à toutes les exigences	
Faisabilité	En tenant compte des ressources et des contraintes	Chiffrage , expérience , estimations .

2.7. Scénarios de qualité

Un attribut de qualité formulé par un simple nom (ex : "performance" , "ergonomie" , ….) est un peu vague. On a souvent **besoin de préciser finement (qualitativement et quantitativement) un "objectif qualité" visé**.

Si l'on réfère aux habitudes méthodologiques qui tournent autour d'UML , tout objectif "métier" ou "technique" à atteindre peut se modéliser comme un "use case" et l'on peut associer à ce "use case" un ou plusieurs scénarios.

Dans le jargon de l'architecture logicielle on appelle couramment "**scénario de qualité**" un **scénario qui explicite tous les détails suivants concernant un "objectif qualité" à atteindre** :

- **acteur**(s)/**intervenant**(s) mis en jeu *(ex : développeur , exploitant , architecte , ...)*.
- **stimulus** déclencheur/ **événement source** *(ex : plantage / stack_overflow)*
- éventuel **environnement** ***(contexte** dans lequel l'événement peut se produire)*
- éventuel **artifact logiciel** *(partie du système touchée ou mise en jeu)*
- **but(s)** précis à atteindre *(ex : réactivation rapide de l'application + analyse du problème à résoudre)*.
  NB : dans beaucoup de cas assez automatisés le but à atteindre peut être assimilé à la **réponse** donnée par le système au stimulus déclencheur *(ex : basculement vers le système redondant de secours)*
- éventuel enchaînement des **actions** à réaliser *(ex : arrêt + sauvegarde + redémarrage)*
- une ou plusieurs **mesure(s) quantitative(s)** *(ex : en 5 minutes , via 2j/h , ... , avec 2Go)* souvent associées à la réponse donnée par le système .

NB : On peut envisager des scénarios plus ou moins concrets ou génériques .

TP ---> rédiger quelques scénarios de qualités .

3. Théorie architecturale générale

Formule de *Perry et Wolf*:

Architecture = **Eléments** + **Formes** + **Motivations**.

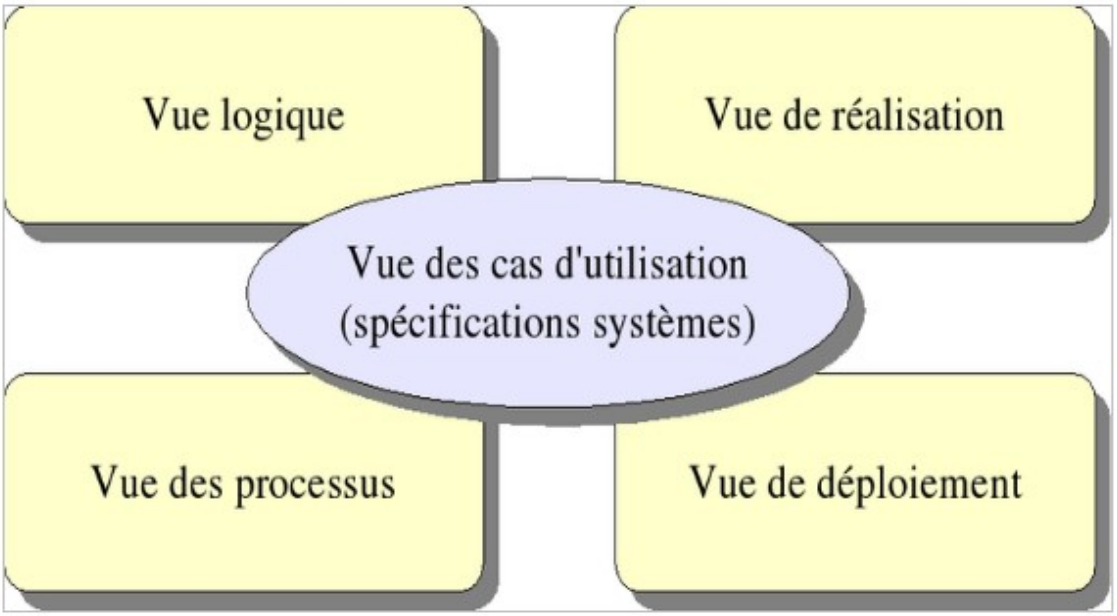
Eléments de l'architecture :
Selon le niveau de granularité, les éléments peuvent varier en tailles (lignes de code, procédures ou fonctions, modules ou classes, paquetages ou couches, applications ou systèmes informatiques), ils peuvent varier en raffinement (ébauche, solution à améliorer ou solution finale) et en abstraction (idées ou concepts, classes ou objets, composants logiciels). Les éléments peuvent également posséder une *temporalité* (une existence limitée dans le temps) et une *localisation* (une existence limitée dans l'espace).
Principaux types de liens entre éléments: *dépendance fonctionnelle, flot de contrôle, transition d'état,changement d'activité, flot de données, lien de communication, composition, héritage (généralisation), envoi de message* .

***Formes** (des représentations)* :
Syntaxe utilisées au niveau des diagrammes (losange, rectangle, …) . exemple : **Formalisme** UML.

***Motivations** (points de vue) :*
Indépendamment de la forme que prend un diagramme d'architecture, celui-ci ne représente toujours qu'un point de vue sur le système considéré, le plus important étant les motivations. En effet, à quoi sert de produire un diagramme s'il est inutile (pas utilisé) ou si les raisons des choix architecturaux sont vagues et non-explicités. Pour éviter de formuler les motivations pour chaque diagramme, l'architecte produira les différents diagrammes en fonction d'un *modèle de conception* et réutilisera des patrons de conception éprouvés.

Un modèle de conception (ou d'architecture) est composé d'un ensemble de points de vue, chacun étant composé d'un ensemble de différentes sortes de diagrammes. Il propose également des moyens pour lier les différentes vues et diagrammes les uns aux autres de manière à naviguer aisément, il s'agit des mécanismes de traçabilité architecturale. La traçabilité doit également s'étendre aux spécifications systèmes et même jusqu'aux besoins que ces spécifications comblent. La devise des trois pères fondateurs d'UML est « *Centré sur l'architecture, piloté par les cas d'utilisation et au développement itératif et incrémentiel* ». Cette devise indique clairement qu'aucune décision architecturale ne doit être prise sans que celle-ci ne soit dirigée (pilotée) par la satisfaction des spécifications systèmes (cas d'utilisation).

4. Modèle d'architecture de P. Kruchten (4 + 1 vues) :



Vue centrale (cas d'utilisation) :
vision des utilisateurs , fonctionnalités / objectifs (avec scénarios/activités)

La vue logique :

La vue logique constitue la principale description architecturale d'un système informatique et beaucoup de petits projets se contentent de cette seule vue. Cette vue décrit, de façon statique et dynamique, le système en termes d'objets et de classes. La vue logique permet d'identifier les différents éléments et mécanismes du système à réaliser. Elle permet de décomposer le système en abstractions et constitue le cœur de la réutilisation.

L'éventuelle vue des processus/threads/tâches :

La vue des processus décrit les interactions entre les différents processus, threads (fils d'exécution) ou tâches, elle permet également d'exprimer la synchronisation et l'allocation des objets. Cette vue permet avant tout de vérifier le respect des contraintes de fiabilité, d'efficacité et de performances des systèmes multitâches.

NB : au sein des serveurs d'applications , les threads sont gérés automatiquement et cette vue peut donc souvent être considérées comme secondaire.

La vue de réalisation (à base de composants / frameworks / librairies) :

La vue de réalisation permet de visualiser l'organisation des composants (bibliothèque dynamique et statique, code source...) dans l'environnement de développement. Elle permet aux développeurs de se retrouver dans le capharnaüm que peut être un projet de développement informatique. Cette vue permet également de gérer la configuration (auteurs, versions...).

La vue de déploiement (topologie) :

La vue de déploiement représente le système dans son environnement d'exécution. Elle traite des contraintes géographiques (distribution des processeurs dans l'espace), des contraintes de bandes passantes, du temps de réponse et des performances du système ainsi que de la tolérance aux fautes et aux pannes. Cette vue est fort utile pour l'installation et la maintenance régulière du système.

5. Les points de vue architecturaux

5.1. Le point de vue "modules"

Un module regroupe un ensemble d'éléments cohérents et est souvent packagé en tant que fichier (.dll , .lib , .zip , .jar , ….) séparé de façon à permettre des déploiements partiels et des changements de versions.

Un module à souvent deux parties :

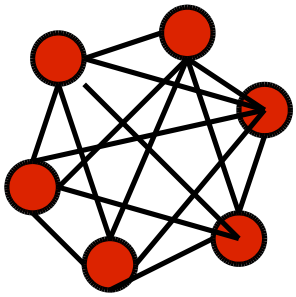
- interface publique (ou façade)
- implémentation privée (volontairement masquée)

Tout comme les objets au sein d'une application , les modules sont souvent organisés selon une structure hiérarchique et décentralisée :

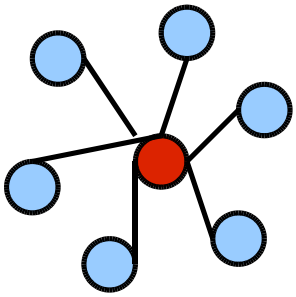
Couplage entre objets (idéalement faible)

Un ***objet élémentaire (tout seul)*** rend souvent des ***services assez limités***.

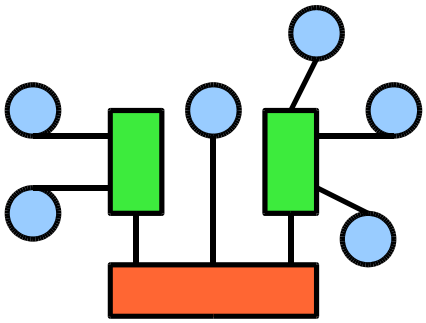
Un ***assemblage d'objets complémentaires*** rend globalement des ***services plus sophistiqués***. Cependant la complexité des liens entre les éléments peut éventuellement mener à un édifice précaire:



couplage trop fort
==> *complexe* ,
trop d'inter-relations
et de dépendances
==> *inextricable*.



couplage faible
mais ***centralisé***
==> peu flexible
et point central
névralgique



couplage faible
et ***décentralisé***
==> simple ,
relativement flexible
et plus robuste

5.2. Le point de vue "composants et connecteurs"

Un **composant** logiciel est conçu pour faire quelque-chose de précis dans le cadre d'un certain style architectural (ex : retourner des données sur demande , filtrer des messages , afficher une données , effectuer un calcul , …).

De façon à ce qu'un assemblage de composants ne soit pas trop rigide, les composants sont en règle générale associés entre eux via des intermédiaires appelés génériquement "connecteurs".

Un **connecteur** peut souvent être assimilé à un "**point d'accès**" , un "**point d'entrée**" ou à un "**port**" et peut être quelquefois vu comme une sorte la destination d'un câblage virtuel ou physique (ex : "port usb" , "port tcp/ip" , ….) .

Quelques exemples :

- On invoque un service web via son point d'accès ("endpoint" identifié par une URL) . Le composant logiciel offrant le service est "caché derrière" et peut être implémenté dans un langage quelconque (ex : "java" , "c++" , "php" , …).
- On se connecte (via tcp/ip) à un des processus d'un ordinateur via son port "tcp" ou "udp" .
- On se connecte quelquefois à un module fonctionnel via le biais d'une façade (simple intermédiaire) .
- On accède en lecture à une propriété d'un objet via une méthode de type "getter".
- On accède en écriture (indirecte et contrôlée) à une propriété d'un objet via une méthode de type "setter"
- ...

5.3. Le point de vue "ressources et allocation"

Tout système informatique utilise (ou consomme) certaines **ressources** :

- mémoire vive (RAM)
- espace disque (Ko , Mo , Go, ...)
- unité de temps d'un processeur (CPU)
- bande passante réseau
- périphériques divers (écran , imprimante , ….)
- courant électrique
- ….

Certains systèmes distribués utilisent également les ressources suivantes :

- machines physiques ou virtuelles (ex : nœud/membre d'un cluster)
- …

D'autre part , un ordinateur (ou un parc d'ordinateurs) offre des **ressources** qui devront être **partagées** par plusieurs processus logiciels (et par plusieurs tâches/threads au sein d'un même programme).

Il faut donc préciser clairement quelles sont les **ressource** qui doivent être **allouées** à telle ou telle **unité d'exécution** dans un intervalle de temps donné.

A un bas niveau (à l'échelle d'un ordinateur), l'**allocation des ressources** est en grande partie la **tâche** du ***système d'exploitation*** (ex : unix/linux , windows , android , …).
La plupart des O.S. actuels sont du type "multitâches / préemptif / à temps partagé" et *SMP=Symetric Multi Processing (tous les processeurs accèdent à un même espace de mémoire partagé).*

Un O.S. ordinaire n'est généralement pas "temps réel" . Il ne garantit pas l'exécution d'une tâche en un temps maximal borné et déterministe car il peut y avoir d'autres tâches parallèles qui ralentissent considérablement le système.
Des applications ayant des contraintes "temps réels" devront s'appuyer sur des systèmes

d'exploitations spéciaux (RTLinux , QNX, VxWorks, ...) et être codés dans des langages compilés (ex: C/C++) .

Si l'on prend le *problème d'allocation des ressources à un haut niveau* (système informatique complet) , le principal travail de l'architecte est d'évaluer (chiffrer) les différentes consommations de ressources logicielles nécessaires:

- ***Topologie de l'environnement cible de production (répartition des serveurs)***
  et pour chaque partie/chaque ordinateur :
  - *Machine virtuelle + O.S.*
  - ***Espace disque , Taille RAM , type , nombre et puissance des processeurs***
  - ***Logiciels nécessaires (avec configurations)***
- ....
- *Environnement de développement nécessaire* : Outils UML , JDK , Référentiel de code source (svn/git/...) , référentiel maven , IDE et plugins (ex: eclipse , ...) , serveur d'intégration continue (ex: Jenkins) , serveurs d'applications , base de données , ...
- ....

Outre les consommations "software" et "hardware" , il faudra tenir compte des ***ressources humaines*** nécessaires à la modélisation , au développement et au fonctionnement du système informatique:

- nombre et type de développeur
- DBA , analyste/concepteur , ...
- exploitant/pilote/administrateur
- support (résolution des problèmes) / ...

En règle générale, l'utilisation de technologies "open source" est moins onéreuse que l'achat de lourds serveurs commerciaux mais quelquefois le coût d'assemblage et de paramétrage de diverses technologies complémentaires revient quasiment aussi cher que l'acquisition de solution se voulant assez complète et un peu "clef en main".

Pour effectuer un choix entre différentes alternatives et bien estimer le nombre et la taille des ressources à allouer , l'expérience de l'architecte compte énormément !!!!

6. Styles architecturaux

Un style architectural est un *patron ou* ***modèle récurrent*** (commun) **d'organisation de l'architecture de système.**

Les styles sont généralement décrits par des noms/phrases simples et compactes , mais évoquent toute une architecture (ex: "client/serveur" , "orienté objet et orienté aspect" , "soa" , …).

Un style architectural est généralement uniformément appliqué à l'architecture d'un (sous-)système.
Cependant un ***système informatique peut éventuellement utiliser plusieurs styles selon le niveau de granularité ou l'aspect du système décrit*** (ex : *"orienté services" à grande échelle* et *"orienté objets" à petite échelle*) .

Un style architectural est essentiellement déterminé par :

- Un ensemble de *types* de composants (ex: bases de données, processus, procédure)
- Une structuration *topologique* des composantes indiquant leur relations durant l'exécution
- Un ensemble de *contraintes sémantiques* sur les composants et leur interrelations
- Un ensemble de *connecteurs* (ex:appel de procédures ou méthodes,interruptions, etc.) pour gérer les communications et la coordination entre les composants

Et l'analyse d'un style architectural passe par la compréhension de :

- Quels sont les types de composantes et de connecteurs ?
- Comment le contrôle est géré et passé entre composantes (synchronisation , …) ?
- Comment les données sont échangées ? (d'où à où ? , régulièrement ou sporadiquement ? , partagées ? Diffusées (broadcast) , envoyées (unicast) ? , résolution du vis à vis lors d'un échange ? ...)
- Comment le contrôle et les données interagissent ? (isomorphe si "orienté objet" ou pas sinon , dans quelles directions ? , ...)
- Quel type de raisonnement est possible ?

Une classification possible (parmi d'autres) de certains styles architecturaux :

Grands Styles architecturaux

Composantes indépendantes

Processus communiquant *(via invocations de services ,via RPC, ... , communications synchrones ou asynchrones , clients/serveur,)*

Systèmes à base d'événements *(avec invocations implicites de type "publication/souscription" ou bien explicites de type "point à point ") (exemple interne: composants graphiques et logique événementielle: écoute / réaction)*

Flux de données

Séquentiel batch *(avec zone de stockage temporaire entre 2 batchs) ---> adaptés au recouvrements d'erreurs mais pas interactifs*

Pipes/tuyaux/files *et* ***filtres****(idéalement réutilisables) ou routeurs*

"événements" de types "données/messages qui arrivent"

Orienté ***données partagées*** *(avec intégrité souvent garantie)*

*avec approche "**référentiel**" (ex : SGBDR , ...)*

*avec approche "**blackboard**" (avec **notifications actives des changements**)*

Machine virtuelle *(où le programme est traité comme une donnée → portabilité)*

interprétation *(totale ou partielle si semi-compilé)*

à base de règles *(déductions, ...)*

“Call-and-return"

décomposition fonctionnelle structurée *(fonction principale et sous routines)*

orienté objets *(+ éventuellement orienté aspects)*

en ***couches logicielles***

6.1. Styles liés aux architectures fondamentales (paradigmes)

Architecture en appels et retours (décomposition fonctionnelle) :

Cette architecture est basée sur le raffinement graduel proposé par Niklaus Wirth. Cette approche consiste à découper une fonctionnalité en sous-fonctionnalités qui sont également divisées en sous sous-fonctionnalités et ainsi de suite; la devise diviser pour régner est souvent utilisée pour décrire cette démarche.

NB : ce style d'architecture est compatible avec l'architecture orientée objet ---> les fonctions sont appelées sur des objets.

Architecture en couches :

Une bibliothèque très spécialisée utilise des bibliothèques moins spécialisées qui elles-mêmes utilisent des bibliothèques génériques.

La division en couches consiste alors à regrouper les composants possédant une grande cohésion (sémantiques semblables) de manière à créer un empilement de paquetages de composants; tous les composants des couches supérieures dépendants fonctionnellement des composants des couches inférieures.

NB : ce style d'architecture est compatible avec l'architecture orientée objet ---> les composants des couches sont assimilables à des gros objets.

Architecture centrée sur les données (client/serveur) :

Dans cette architecture, un composant central (SGBD,Datawrehouse , Blackboard) est responsable de la gestion des données (conservation, ajout, retrait, mise-à-jour, synchronisation, ...) . Les composants périphériques, baptisés clients, utilisent le composant central, baptisé serveur de données, qui se comporte, en général, de façon passive (SGBD, Datawarehouse). Un serveur passif ne fait qu'obéir aveuglément aux ordres alors qu'un serveur actif (Blackboard) peut notifier un client si un changement aux données qui le concerne se produit.

Cette architecture sépare clairement les données (serveurs) des traitements et de la présentation (clients) et permet ainsi une très grande intégrabilité, en effet, des clients peuvent être ajoutés sans affecter les autres clients. Par contre, tous les clients sont dépendants de l'architecture des données qui doit rester stable et qui est donc peu extensible. Ce style nécessite donc un investissement très important dans l'architecture des données.

*NB : Via le mapping objet/relationnel (ORM) ce style d'architecture est **à un sous niveau** compatible avec l'architecture orientée objet ---> les données sont vues comme des objets de type "entité persistante".*

Architecture en flot de données (messages, documents,événements, ….) :

Cette architecture est composée de plusieurs composants logiciels reliés entre eux par des flux de données. L'information circule dans le réseau et est transformée par les différents composants qu'elle traverse. Si les composants sont répartis sur un réseau informatique et qu'ils réalisent des transformations et des synthèses intelligentes de l'information, on parle alors d'architecture de médiation. Les architectures orientées évènements font également partie de cette catégorie.

NB : En considérant les "messages" , "événements" et "documents" comme des objets, ce style d'architecture est compatible avec l'architecture orientée objet.

Architecture orientée objets :
Les composants du système (objets) regroupent des données internes et les opérations de traitement sur ces données. La communication et la coordination entre les objets sont réalisées par un mécanisme d'invocation de messages. Cette architecture est souvent décrite par les trois piliers : encapsulation, héritage et polymorphisme. Le principal apport de la programmation orienté objet est la modularité (déterminante au niveau de système complexe).

Architecture orientée agents :

L'architecture orientée agents correspond à un paradigme où l'objet, de composant passif, devient un composant projectif :

En effet, dans la conception objet, l'objet est essentiellement un composant passif, offrant des services, et utilisant d'autres objets pour réaliser ses fonctionnalités; l'architecture objet n'est donc qu'une extension de l'architecture en appels et retours, le programme peut être écrit de manière à demeurer déterministe et prédictible.

L'agent logiciel, par contre, utilise de manière relativement autonome, avec une capacité d'exécution propre, les autres agents pour réaliser ses objectifs : il établit des dialogues avec les autres agents, il négocie et échange de l'information, décide à chaque instant avec quels agents communiquer en fonction de ses besoins immédiats et des disponibilités des autres agents.

6.2. Synchrones ou asynchrones ?

Services synchrones (Remote Procedure Call, ...):

- dialogues à base de requêtes et de réponses quasi-immédiates
  - mode très répandu car très intuitif (assimilable à un appel de méthode / RPC).
- développement rapide avec api moderne (ex : Web Service Soap , EJB/RMI)
- logique plutôt "stateless" : paquet de méthodes partagées (en mode ré-entrant).
- s'adapte bien au structure n-tiers et en couches logicielles
- dépendance forte entre client et serveur (haute disponibilité souvent souhaitée)

Middlewares asynchrones (file d'attente, e-mail, …) :

- efficacité (traitements potentiellement effectués en grande série à l'image d'une chaîne de personne qui se passent rapidement des seaux d'eau pour éteindre un feu)
- permet des rythmes différents entre client et serveur (accumulation si nécessaire de messages en attente).
- permet d'établir des communications entre éléments logiciels qui ne sont pas connectés en permanence (stockage des messages dans une zone temporaire puis acheminement ultérieur vers une prochaine destination).
- communications plus flexible (selon intermédiaire) mais souvent un peu plus longues à programmer (gestion des corrélations , des priorités, ...)

NB :Des passerelles entre les modes synchrones et asynchrones sont quelquefois envisageables :

- un service synchrone peut s'appuyer sur des sous services asynchrones très rapides
- un appel de méthode peut être vu comme un micro document(message) à traiter.

6.3. Styles liés aux langages et aux configurations

Aujourd'hui, beaucoup de langages au repris la fonctionnalité "compilation juste à temps" et les vitesses d'exécution ne sont donc plus très différentes d'un langage à l'autre.

Au final, le choix d'un langage informatique (et donc des librairies disponibles dans ce langage) dépend donc essentiellement des **critères stylistiques suivants** :

- ***syntaxe compacte de type script*** (ex : javascript avec frameworks modernes , python, rubby , …)
  ou bien
  syntaxe développée et très structurée de type classe d'objet *(ex : java)*

- langage ***fortement typé*** (integer, string, double, …) ou bien ***faiblement typé*** (variant) .
- …

C'est subjectivement une affaire de goût .
La compacité des lignes de code de type script compense parfois en temps de développement le gain de "debug" apporté par une programmation plus structurée/rigoureuse.
C'est objectivement ***une affaire de connaissances déjà acquise ou pas*** et de ***temps d'apprentissage souhaité ou disponible*** (un langage de type script est en apparence plus rapidement accessible et le temps de formation est moins long).

D'autre part , au sein d'un même langage (ou framework) informatique, on peut opter pour différents modes de configuration :
- ***configuration xml*** (verbeux , explicite , plutôt centralisée et quelquefois hiérarchique)
- ***configuration par annotations*** (compact, au plus proche des lignes de code)

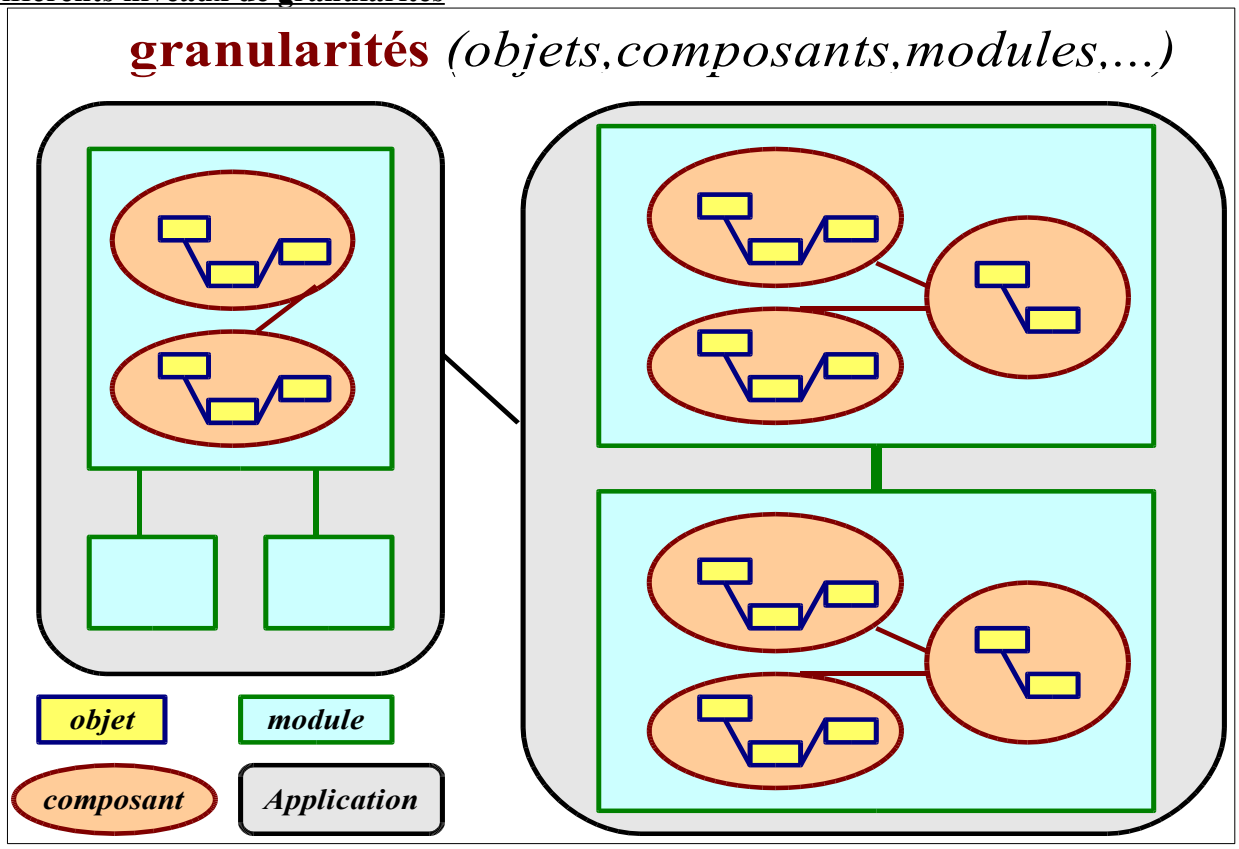
C'est encore partiellement une affaire de goût .
Objectivement, une configuration XML est incontournable lorsqu'il s'agit de configurer des éléments logiciels que l'on à pas codé (sans code source).

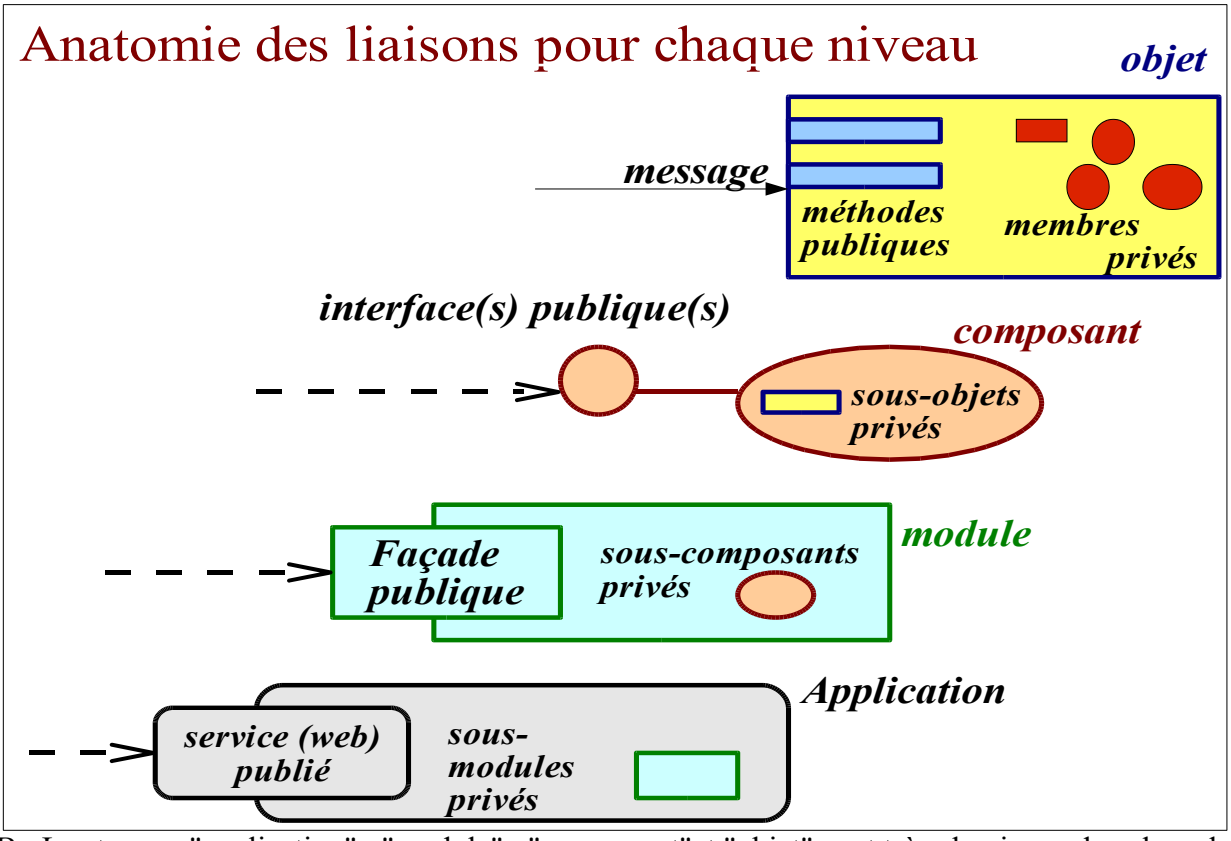
6.4. Design patterns et frameworks

Notions basiques et classiques en architecture/conception logicielle.
→ voir l'annexe adéquate si besoin.

6.5. Styles pour "intégration" et "organisation des modules"

Différents niveaux de granularités



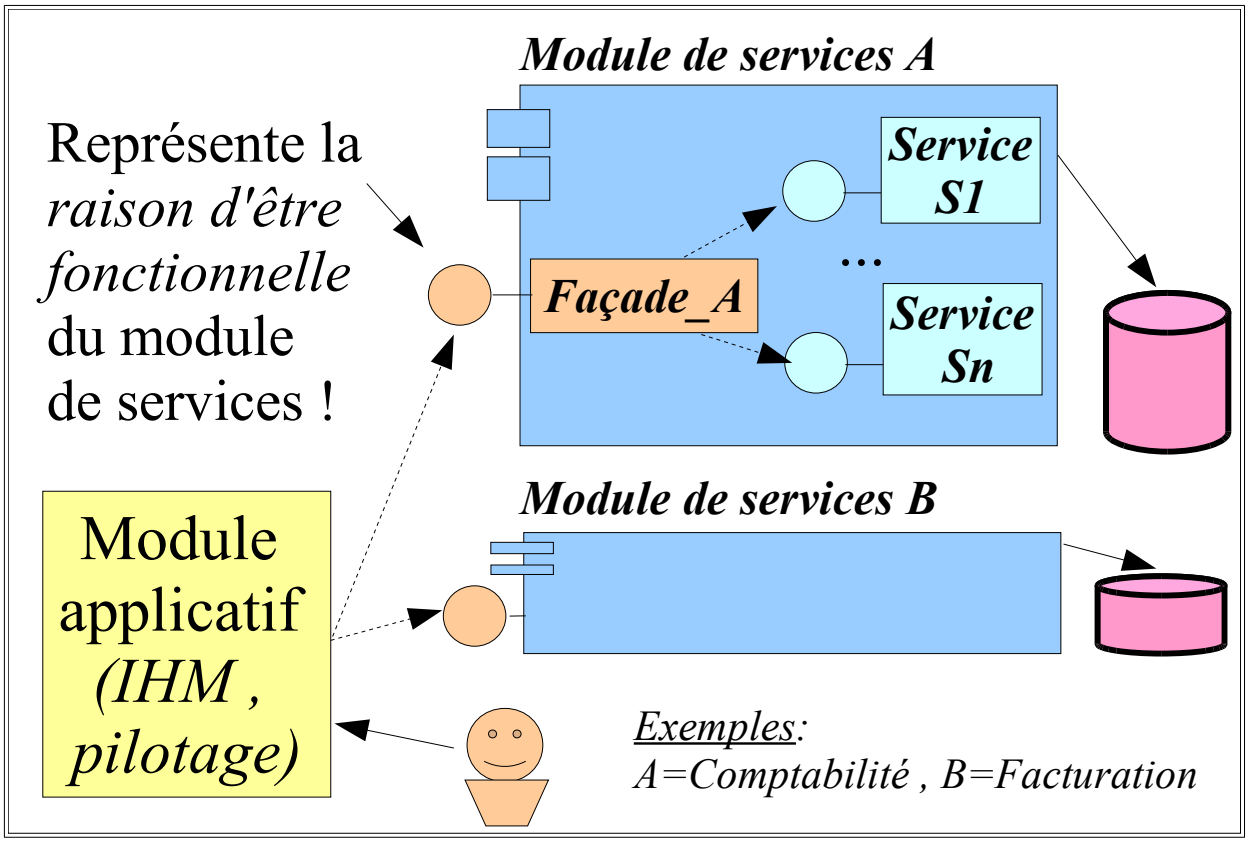


NB : Les termes "application" , "module" , "composant"et "objet" sont très classiques dans le cadre de la programmation orientée objet . On peut cependant utiliser d'autres synonymes .

L'organisation des modules est souvent hiérarchique avec généralement des règles de dépendances basées sur des niveaux :

- un module de bas niveau n'utilise jamais directement un module de haut niveau (il ne fait éventuellement que remonter des notifications vers une destination enregistrée au démarrage).
- ….

Façade



D'un point de vue fonctionnel , une façade donne du sens à un module de service. Son nom doit donc être bien choisi (pour être évocateur).
D'un point de vue technique, une façade n'est qu'un nouvel élément intermédiaire de type "Façade d'accueil" qui servira à orienter les clients vers les différents services existants.

Exemple:

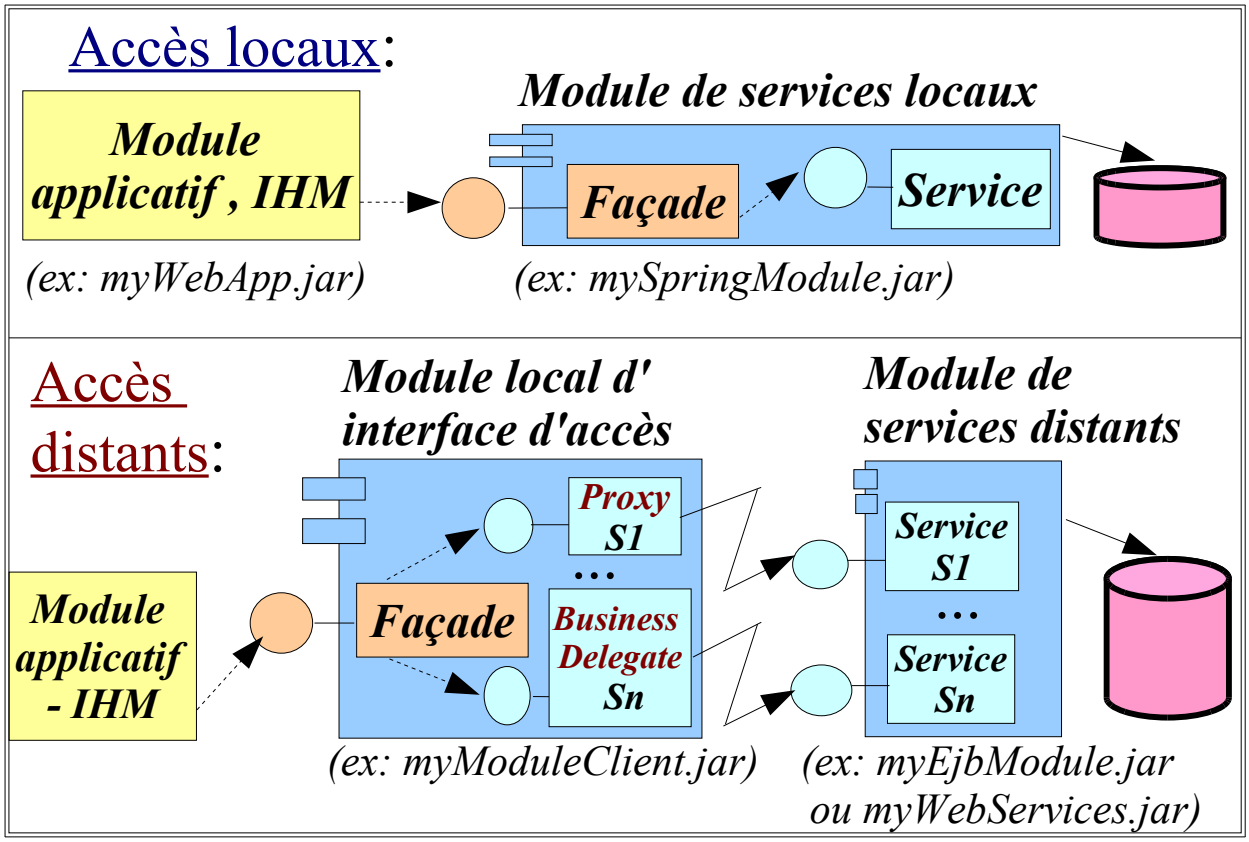
Façade_Comptabilité
. getServicePostesComptables() ***.getServiceBilan()*** ***.getServiceJournal()*** ***.getServiceGrandLivre()*** ***...*** ***.getServiceN()***

utilisation:

facadeCompta.getServiceBilan().xxx()

facadeCompta.getServiceGrandLivre().yyy();

Business Delegate (pour accès distants transparents) :



Derrière une façade (très souvent locale) on peut éventuellement trouver des services distants.

Un "**proxy**" est un représentant local d'un service distant .

Ce terme (plutôt technique) désigne assez souvent du code généré automatiquement (à partir d'un fichier WSDL par exemple).

Pour bien contrôler l'interface locale d'un service distant (de façon à n'introduire aucune dépendance vis à vis d'une technologie particulière), on met parfois en oeuvre des objets de type "**business delegate**" qui cache dans le code privé d'implémentation tous les aspects techniques liés aux communications réseaux:

- localisation du service , connexion (lookup(EJB) ou ...)
- préparations/interprétations des messages/paramètres .