CNAM - CRA Nancy
2003

Génie Logiciel

Jacques Lonchamp

QUATRIEME PARTIE

Les techniques de conception.

1. Introduction

La conception propose une *solution* au problème spécifié lors de l'analyse :

- *architecture* de l'application (architecture logicielle et architecture physique),
- description détaillée des *modules*, des *interfaces utilisateurs*, des *données*.

Elle donne lieu à un dossier de conception avec souvent une partie destinée au client (présentation de la solution) et une partie pour les réalisateurs (conception technique).

Nous décrivons surtout la conception de l'*architecture logicielle*, c'est à dire de la décomposition du système en *modules*, avec la description abstraite de ce que chaque module doit faire et la description des relations entre les modules.
Nous évoquons plus brièvement les architectures physiques.
La description précise du contenu des modules relève de la phase de *conception détaillée*. Nous détaillons peu cet aspect, assez dépendant des techniques de mises en oeuvre que l'on prévoit d'utiliser.

2. Modules et relations

Un module est un composant d'une application, contenant des définitions de données et/ou de types de données et/ou de fonctions et constituant un tout cohérent. On peut définir un module comme un *fournisseur de ressources ou de services*. Quand on décompose un système en modules il faut décrire précisément les relations entre ces modules.

Soit un système $S = \{M_1, M_2, \ldots, M_n\}$. Une relation R est une partie de SxS ; on peut écrire $M_i \; R \; M_j$, si les deux modules sont en relation, c'est à dire si $(M_i, M_j) \subset SxS$.
La clôture transitive de R, notée R^+, peut être définie récursivement par :

$M_i \; R^+ \; M_j$ ssi $M_i \; R \; M_j$ ou s'il existe M_k tq $M_i \; R \; M_k$ et $M_k \; R^+ \; M_j$ (la clôture transitive inclut les relations directes et les relations indirectes).

Une relation est une *hiérarchie* s'il n'existe pas de modules M_i et M_j tq :

$M_i \; R^+ \; M_j$ et $M_j \; R^+ \; M_i$ (c'est à dire pas de cycle).

Le graphe correspondant est appelé DAG ('Direct Acyclic Graph'). La figure 1 décrit une relation quelconque (réseau) et une hiérarchie.

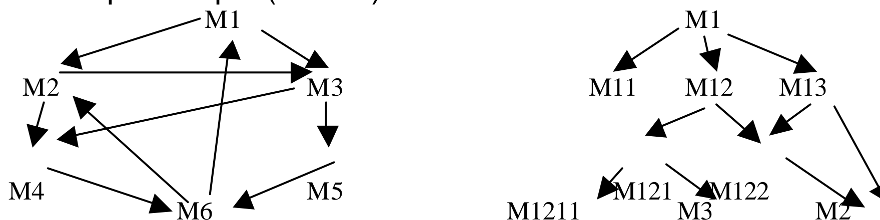


Figure 1 : une relation quelconque et une hiérarchie

On peut grossièrement distinguer deux démarches de conception :

- l'approche fonctionnelle,
- l'approche à objets.

Dans *l'approche fonctionnelle*, un module est un *sous-système* du système global. La relation de base est la relation de *décomposition,* qui constitue une hiérarchie (cf. Fig. 2).

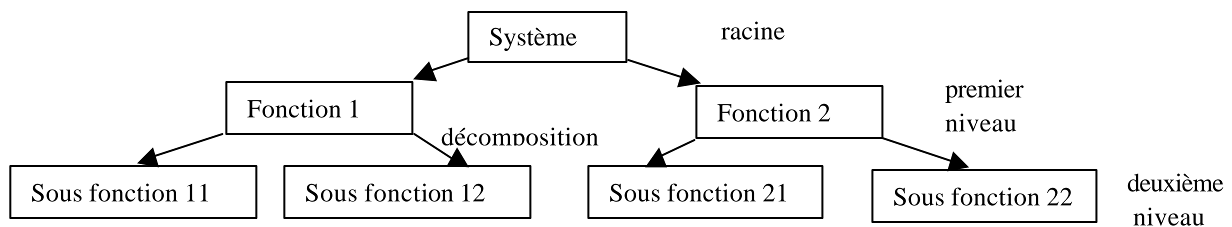


Figure 2 : décomposition fonctionnelle

Les modules peuvent également utiliser des ressources que d'autres modules offrent. Cette *relation d'utilisation* est en général non hiérarchique (réseau).
Une telle architecture est définie *en fonction des traitements*. Par exemple les modules du premier niveau d'une bibliothèque peuvent traiter de la gestion des prêts, de la gestion des emprunteurs, de la gestion des ouvrages, etc. Dans la gestion des prêts, on trouve au second niveau des modules pour l'emprunt, le retour, la relance, la réservation, etc. Cette architecture peut être remise en cause à chaque évolution des fonctionnalités : elle est très mal adaptée aux *évolutions des structures de données* qui peuvent avoir un impact sur énormément de modules.

Dans l'approche à objets, les modules principaux correspondent aux *objets concrets ou abstraits du domaine de l'application ('business objects' ou 'objets métier')*. Par exemple, dans la bibliothèque on peut trouver les objets (modules) Livre, Bibliothécaire, Emprunt, Réservation, Catalogue, etc. Les objets *regroupent données et traitements*. Ce sont des entités autonomes qui *collaborent* pour réaliser le système global.

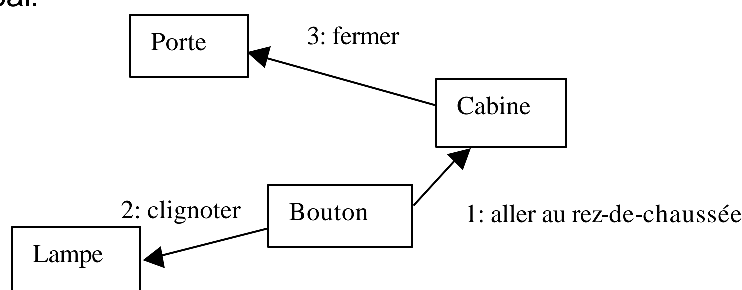


Figure 3 : décomposition à base d'objets qui collaborent

Les objets similaires sont décrits dans des classes dans la plupart des langages à objets (C++, java, ….). L'architecture de l'application est alors décrite au niveau des classes. La relation de base est la relation *d'utilisation (par appel de service)*, qui constitue un *réseau quelconque*. On trouve aussi des relations de décomposition (classe à classe composante ex : voiture à moteur, voiture à carrosserie, etc.) et d'héritage entre classes (d'une classe générale à une classe spécialisée ex : véhicule à voiture, véhicule à camion, etc.). Nous étudierons en détail ces concepts avec UML. *L'architecture n'est plus liée aux seuls traitements mais aux objets du domaine* (données+traitements). Les évolutions sont *plus locales,* donc plus faciles.

Dans tous les cas, on cherche à diviser un système en composants qui *peuvent être conçus indépendamment*. Il faut dans ce cas que la nature de la relation d'utilisation, c'est à dire la nature des ressources et services qu'un module procure soit spécifiée explicitement et précisément : c'est son *interface*. La manière dont ces services sont réalisés est son *implantation qui est cachée*. On sépare ainsi la vue abstraite d'un module telle qu'elle est nécessaire par ses clients (le 'contrat' passé entre le module

et ses clients), de la vue de son implantation. *On peut programmer un module en ne connaissant que les interfaces des autres modules,* c'est à dire *en manipulant des abstractions.* En général, l'interface décrit des ressources (constantes, variables, types) et des fonctions ; mais il peut aussi contenir d'autres types d'informations (ex : un temps de réponse garanti pour les systèmes temps-réel).

Un module bien conçu doit montrer le moins possible de choses à travers son interface. Le reste de l'information est *caché (encapsulé)* dans l'implantation. *La partie cachée peut être modifiée, sans aucun impact sur les modules clients à partir du moment où l'interface ne change pas.*

Les *langages modulaires* (comme ADA) et les *langages à objets* permettent de tirer pleinement profit de ces notions d'interface et d'encapsulation. Aujourd'hui les *approches par composants* tentent d'aller plus loin. L'application peut être construite dynamiquement via un outil qui permet de configurer et connecter des composants. Ceux ci explicitent leurs E/S pour permettre les connexions et leurs propriétés pour permettre leur configuration. C'est le cas des *Java Beans* qui sont des classes avec des conventions d'écriture sur les méthodes et les événements ; l'outil de connexion utilise l'API reflection qui permet d'interroger une classe à l'exécution sur ses propriétés (cf. classe Class).

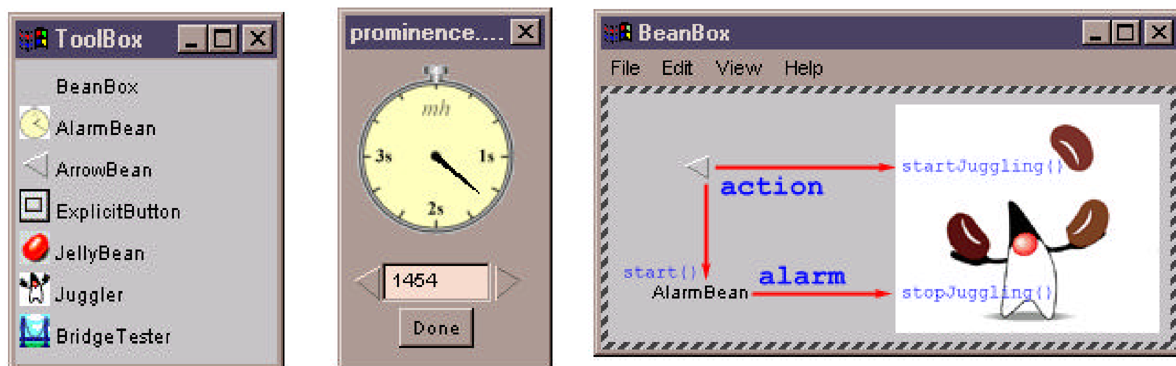


Figure 4. Une palette de beans, un bean (alarme) et la composition de beans.

Les *Entreprise Java Beans* (EJB) étendent cette approche à la construction d'application réparties : le développeur écrit des 'objets métiers' (Enterprise Beans) en respectant certaines règles qui permettent à des serveurs (EJB Containers) de le décharger de tous les problèmes liés à la répartition, aux transactions, à la persistance –via JDBC-, à la sécurité, etc.

3. Illustration de l'approche fonctionnelle

Il s'agit de structurer l'application en modules fonctionnels, généralement selon une hiérarchie de fonctions. SADT, SASD qui regroupe Structured Analysis (SA) et Structured Design (SD) sont des exemples bien connus de méthodes fonctionnelles d'analyse et conception. Ces méthodes datent des années 70.

Le système peut d'abord être décomposé en modules de haut niveau ou sous-systèmes. Puis chaque sous-système est à son tour et indépendamment décomposé, et ce à plusieurs niveaux, par exemple jusqu'à ce que les modules soient réalisables par une personne seule. Ceci décrit une approche *descendante*.

Quand on construit tout d'abord des modules de base en essayant de réutiliser des modules préexistants, l'approche est dite *ascendante*, puisqu'on cherche à les combiner pour réaliser un (sous-)système. Souvent, les processus de conception mêlent étapes descendantes et étapes ascendantes.
La relation de décomposition donne naissance à la *notion de "couche"* : un module de niveau i est implanté par les modules du niveau i+1 qui lui sont liés.

Un exemple de notation : les *diagrammes de structure* (ou 'structure chart') de la méthode SD de Yourdon. Ils reflètent une organisation hiérarchique et fonctionnelle des systèmes. Les rectangles correspondent aux modules. Les grandes flèches correspondent à la relation d'appel entre modules (du type appel de sous programme; il peut éventuellement y avoir plusieurs appels à l'exécution pour une flèche donnée et certaines notations distinguent les appels simples, les appels conditionnels et les appels répétitifs). Les petites flèches correspondent aux flux de données (paramètres, données retournées). Un exemple est donné par la figure 5. Elle se lit de gauche à droite.

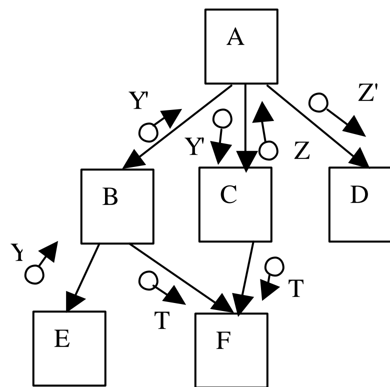


Figure 5 : un diagramme de structure

La relation d'appel entre modules donne des indications sur la qualité de la modularité. Dans une bonne structure modulaire, le nombre d'appels sortants ('fan-out') doit rester relativement faible (pas plus 7, sinon le module est probablement trop complexe) alors que le nombre d'appels entrants ('fan-in') peut être élevé, ce qui prouve que le module est souvent réutilisé (cf. Figure 6).

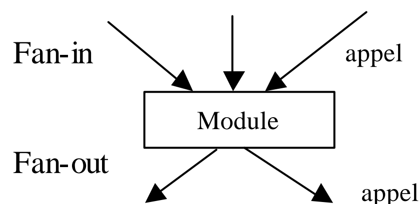


Figure 6 : fan in - fan out

Les modules peuvent être classées en types entrée ('afférent'), sortie ('efférent'), transformation et coordination (cf. Fig. 7).

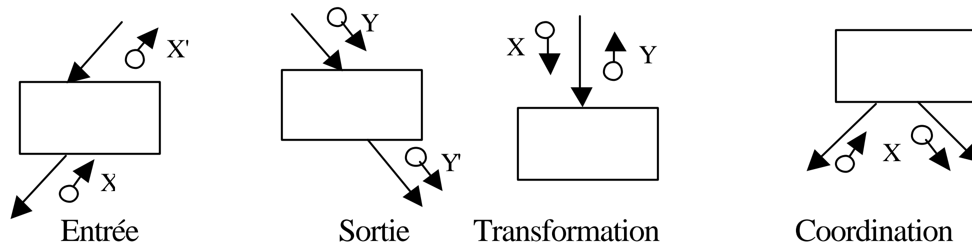
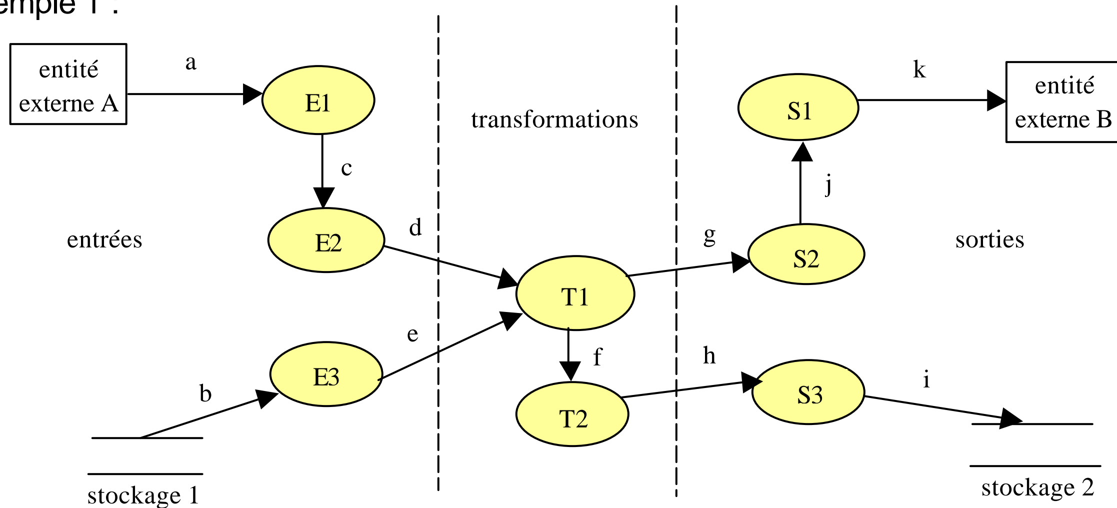


Figure 7 : typologie des modules fonctionnels

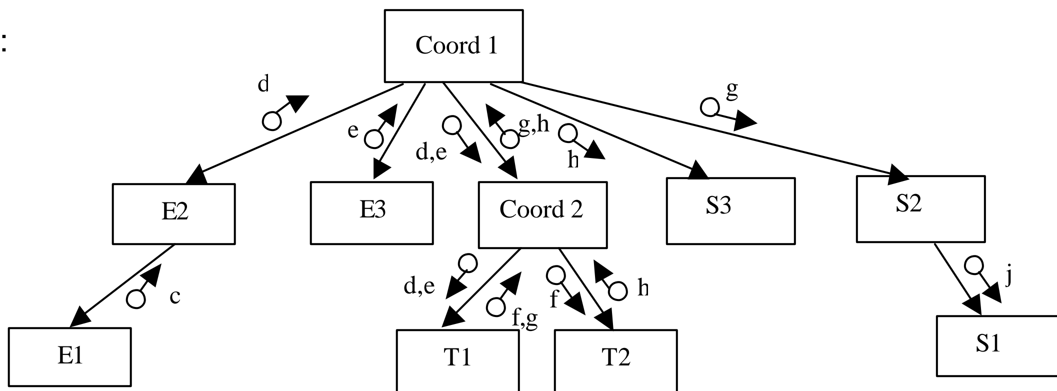
On utilise très souvent les diagrammes de structure en complément des diagrammes de flots de données (DfD de la méthode SA). Le passage peut se 'systématiser' (méthode SA-SD) en particulier quand les flots peuvent être analysés comme des suites de transformations. Il faut :

- classer les fonctions en entrées, transformations et sorties,
- introduire des fonctions de coordination pour dispatcher les flots vers les fonctions relevant d'un même sous système.

Exemple 1 :



donne :



Si E2 est analysée comme incluant plutôt une transformation 'secondaire' TE2 du flux d'entrée (T1 et T2 étant appelées 'transformations centrales') on peut préférer :

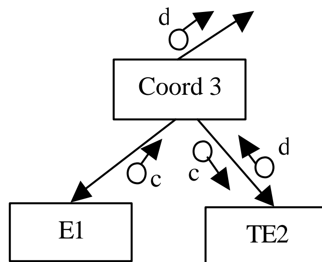


Figure 8 : une transformation SA-SD

Exemple 2 : soit un vérificateur d'orthographe qui cherche chaque mot d'un document dans un dictionnaire. Les mots trouvés sont considérés comme corrects. Les mots non trouvés apparaissent à l'écran et l'utilisateur prend une décision : soit ils sont corrects et sont ajoutés au dictionnaire, soit ils sont incorrects et ajoutés avec la correction à un fichier des mots à corriger. Le DfD suivant peut être construit.

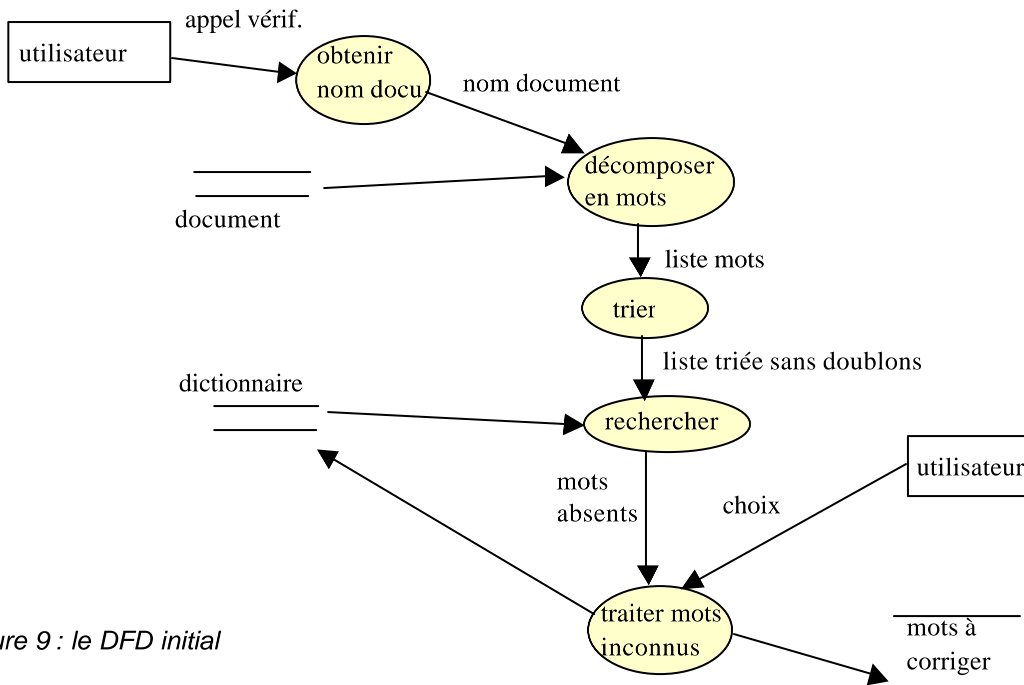
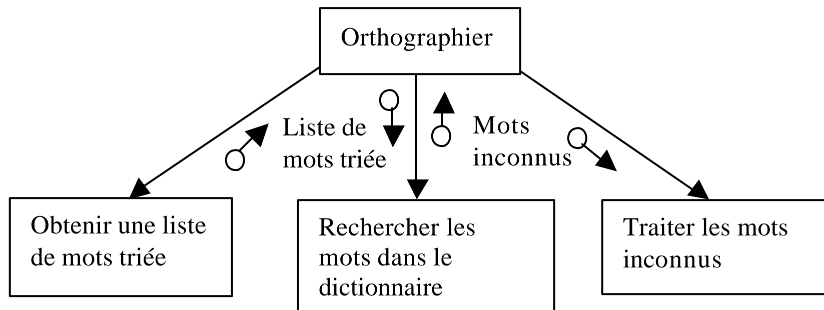


Figure 9 : le DFD initial

La transformation centrale est 'rechercher dans le dictionnaire'. C'est là ou les données deviennent des résultats. D'où la première décomposition :



L'entrée 'obtenir une liste de mots triée' peut se décomposer à deux niveaux pour obtenir :

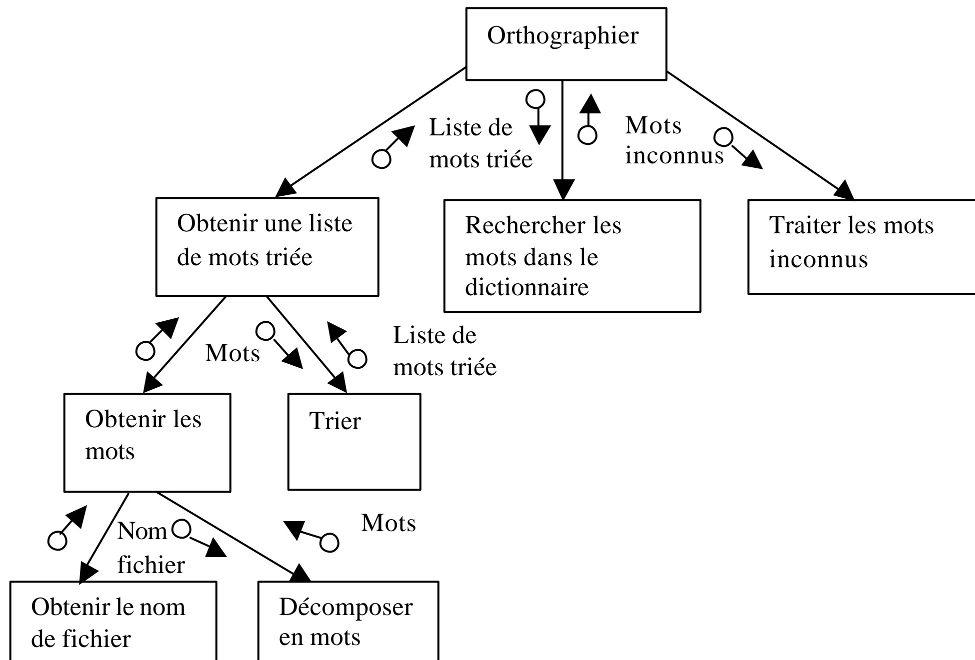


Figure 10 : décompositions hiérarchiques

4. L'approche objets et les « patterns »

Nous étudierons l'approche objet en détail avec la notation UML ainsi que les démarches associées (chapitre 6). La notion de 'pattern' (patron) s'est imposée depuis quelques années pour diffuser les « bonnes pratiques » sous forme de schémas de classes réutilisables pour répondre à des *situations de conception* qui apparaissent de manière *répétitive* dans beaucoup de projets. Il est souvent fait analogie entre l'apprentissage des échecs et celui de la conception informatique. Pour jouer aux échecs, il faut d'abord apprendre les règles (pièces, mouvements, ...), puis les principes (valeur relative des pièces, emplacements stratégiques, ...) puis enfin étudier les *parties des maîtres* qui donnent des idées réutilisables (débuts/fins de partie, sacrifices de pièces ...). En conception, les règle sont les algorithmes, les structures de données, les langages de programmation. Les principes sont ceux de la programmation modulaire, de la programmation à objets ... Les 'design patterns' remplacent les parties des maîtres et comme elles, méritent d'être étudiées.

On distingue :

- les *patterns architecturaux* (structure générale d'un système en sous systèmes),
- les *patterns de conception* ('design patterns' – structure d'un composant en sous composants dans un contexte donné),
- les *idiomes* (patterns de bas niveau dans un langage de programmation donné),
- les *frameworks* ou 'cadres' (sous système prêt à être instancié avec des possibilités bien définies d'adaptation et d'extension).

Parmi les patterns architecturaux, qui se rattachent à la problématique de la conception architecturale, on peut prendre en exemple, le pattern 'Layer' qui décrit la structuration en couches d'un système.

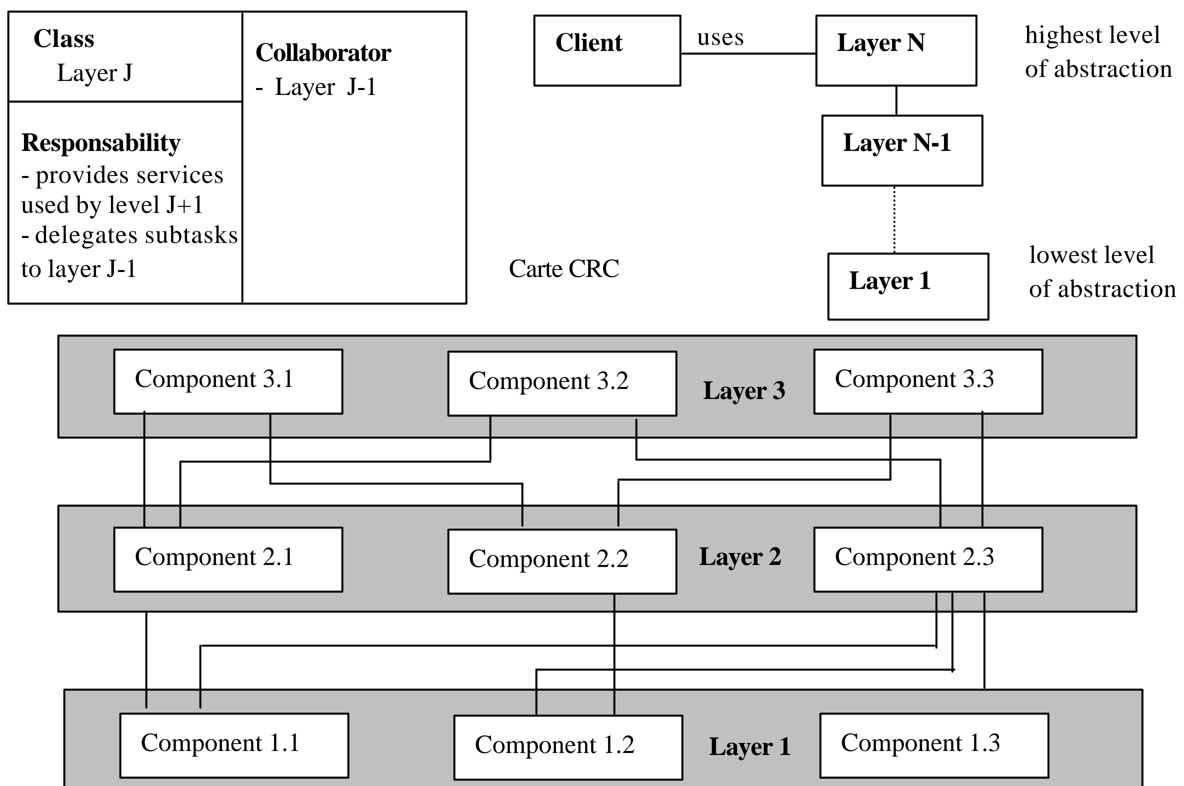


Figure 11 : le pattern architectural Layer

On trouve des utilisations de cette architecture en télécom (le célèbre modèle OSI à 7 couches – physique, liaison, réseau, transport, session, présentation, application) et dans beaucoup d'applications à objets avec les 3 couches 'présentation', 'logique applicative' et 'services de base' (persistance, distribution, ...).

Parmi les autres patterns architecturaux on peut citer les 'pipes and filters' (architectures de type 'flux de données' à travers des tubes et des filtres – cf. exercice 4.2), le 'blackboard' (architecture avec des modules spécialisés qui partagent leurs connaissances pour construire une solution), le 'broker' (architecture distribuée autour d'un composant de routage des requêtes), le 'MVC' (architecture qui sépare les vues externes, les objets applicatifs et les contrôleurs qui lient événements sur les vues et services sur les objets), la 'réflection' (architecture auto descriptive, modifiable dynamiquement à l'exécution), etc.

Concernant les design patterns, nous verrons l'utilisation d'un certain nombre d'entre eux dans l'étude de cas faisant suite à UML ('singleton', 'observer', 'factory').

5. Les architectures physiques

Les systèmes sont de plus en plus souvent distribués sur un réseau. Il se rajoute donc une phase de *conception de la l'architecture physique de l'application*. Les différents modules (liés à la présentation, la logique applicative ('business logic'), aux services de base, comme la gestion des données) sont répartis sur ces architectures physiques. Sans entrer dans les détails, on peut rappeler quelques architectures classiques :

- *système centralisé* (1 niveau ou 1 'tier') : tout le système est installé sur la même machine; il n'y a pas de distribution.

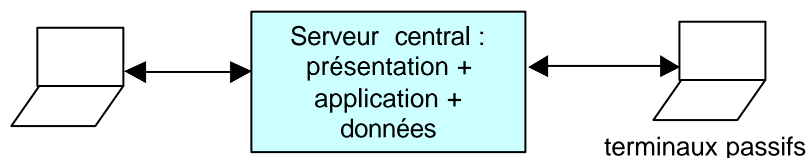


Figure 12 : système centralisé

- *système client/serveur* (2 niveaux ou 2 'tiers') : le serveur gère les données (BD et gestionnaire de BD); les clients gèrent la présentation et la logique des traitements; ils adressent des requêtes au serveur et reçoivent des résultats (ex: requêtes SQL vers Oracle, SQL server, …); on parle de "clients lourds" ('fat clients').

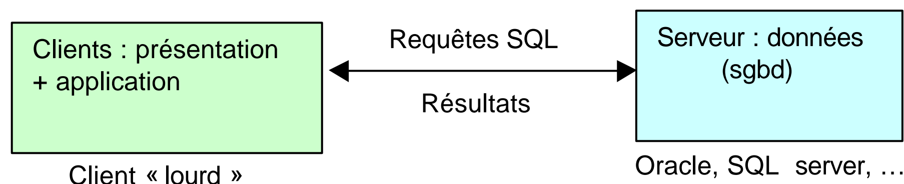


Figure 13 : système client-serveur

- *systèmes 3 niveaux ou 3 'tiers'* : les clients légers ne s'occupent que de la présentation; la logique de l'application est gérée sur un "serveur d'application" qui lui même adresse des requêtes au "serveur des données" (BD + gestionnaire de BD). Il peut arriver que les 2 serveurs soient sur la même machine physique.

Exemple d'architecture 3 niveaux :

- Client léger = navigateur Web (applet, javascript, ...)
- Serveur d'application = serveur http + CGI, servlets, JSP,ASP, PHP, …
- Serveur de données = Oracle, SQL Server, MySQL, SGBDOO, ...

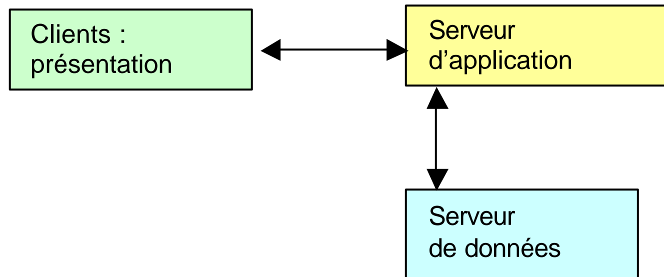


Figure 14 : système 3 niveaux

(même machine ou non)

- *systèmes n niveaux ou 'n tiers' :* le serveur d'application fait appel aux services de plusieurs autres serveurs (d'applications ou de données).

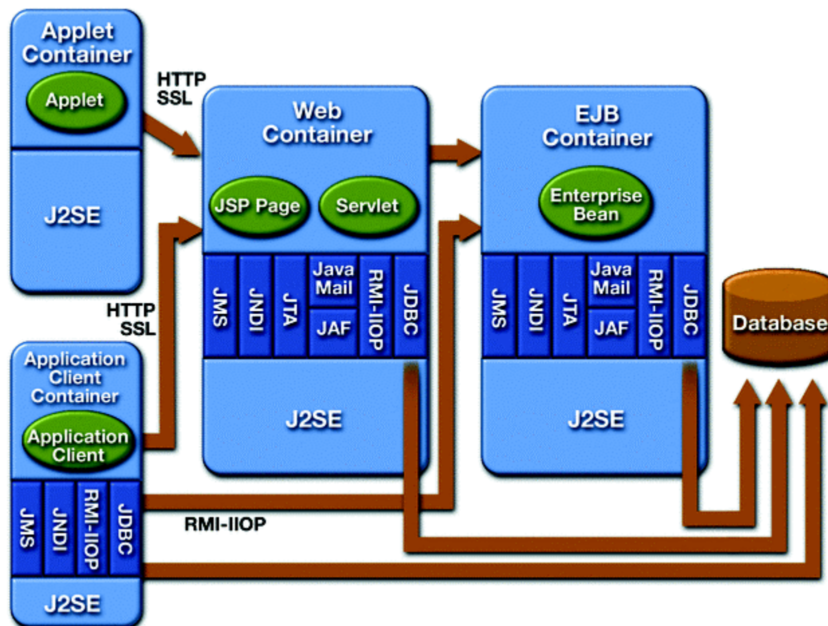


Figure 15 Architecture multi niveaux conforme à l'architecture J2EE.

Le choix de la technologie de communication entre les composants est important. On utilise largement aujourd'hui du *'middleware'*, c'est à dire des technologie de support à la distribution :

- sockets (niveau bas – TCP ou UDP)
- RMI (couche java au dessus des sockets)
- CORBA ("bus logiciel" pour des objets répartis - multi langages)
- appel de procédures à distances en format XML (protocole SOAP). Cette technologie constitue la base des 'Web services'.

Les Web services constituent une technologie permettant de composer des applications à partir d'un ensemble de services distribués sur Internet.
Exemple : une application d'assistance aux voyageurs

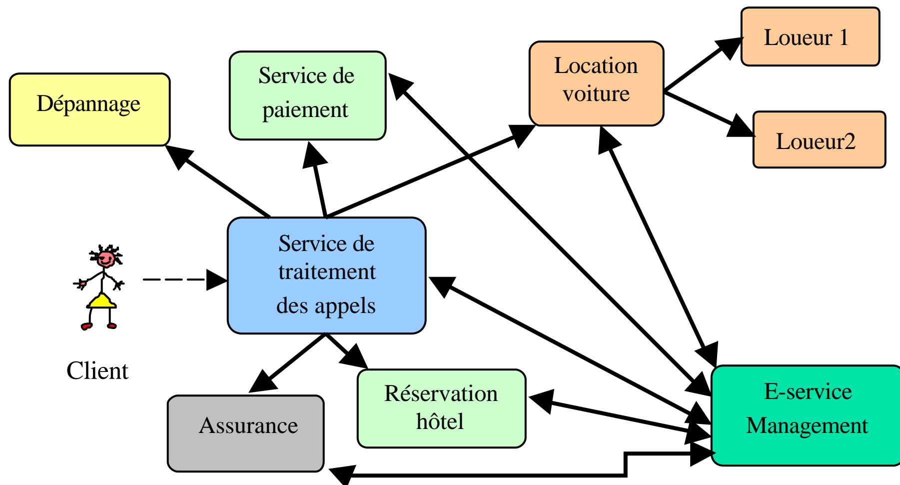


Figure 14 : une application à base de Web services

Les Web services concernent des fournisseurs de services (service providers), des demandeurs de services (service requestors) mis en relation dynamiquement via un annuaire de services (service registry). Tous les protocoles reliant ces acteurs sont à base d'XML (cf. Figure) : SOAP (appel de procédures), WSDL (Web service description language), UDDI (Universal description Discovery Integration).

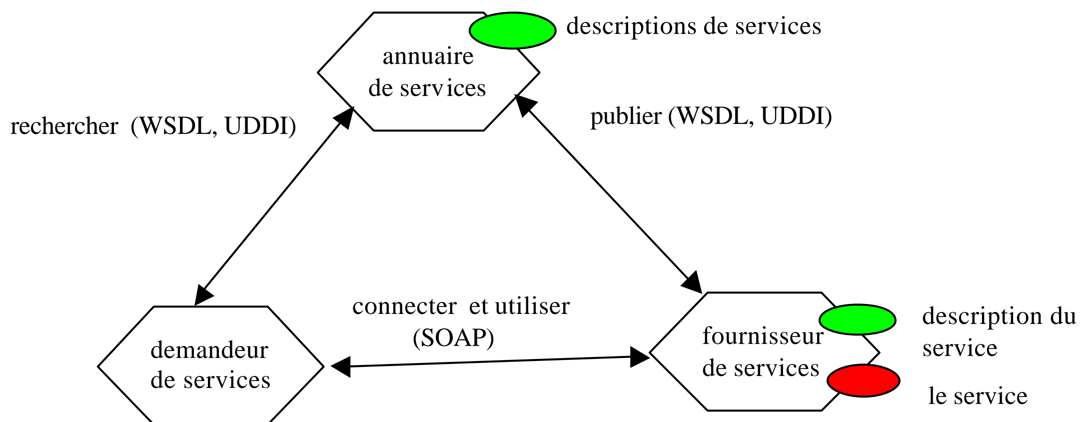


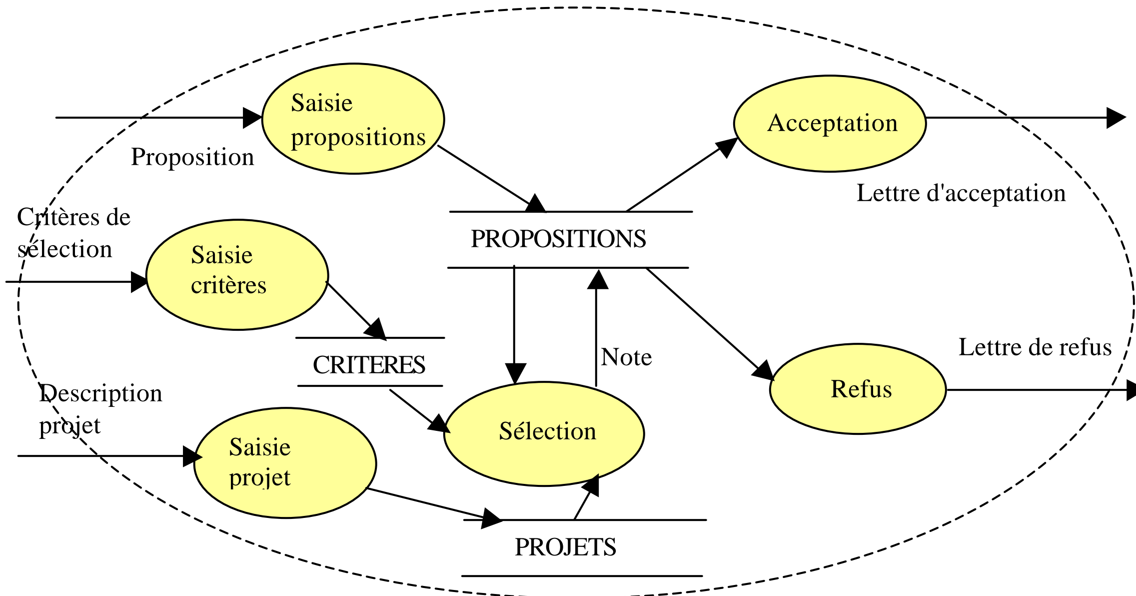
Figure 15 : acteurs et protocoles des Web services

6. Autres aspects de la conception

En plus de la conception de l'architecture de l'application en modules, d'autres aspects doivent être considérés par les concepteurs. C'est le cas de la *conception des interfaces utilisateurs, de la conception des algorithmes, de la conception des bases de données*, etc. Ces aspects relèvent de cours spécialisés sur les interfaces, l'algorithmique et les bases de données.

Exercices

Exercice 4.1 *Diagramme de structure*



A partir du DfD construit au chapitre précédent pour la sélection d'une proposition suite à un appel d'offres, proposer un diagramme de structure.

Exercice 4.2 *Pattern 'pipe and filter'*

Class Filter	***Collaborators***
Responsibility • Gets input data. • Performs a function on its input data. • Supplies output data.	• Pipe

Class Pipe	***Collaborators***
Responsibility • Transfers data. • Buffers data. • Synchronizes active neighbors.	• Data Source • Data Sink • Filter

Class Data Source	***Collaborators***
Responsibility • Delivers input to processing pipeline.	• Pipe

Class Data Sink	***Collaborators***
Responsibility • Consumes output.	• Pipe

Commenter le pattern ci dessus. Donner un exemple d'utilisation.