Remerciement

Ce travail n'aurait certainement jamais vu le jour sans l'aide, le soutient et le dévouement de certains personnes que nous tenons vivement à

Remercier.

Nous tenons d'abord à remercier notre encadreur pour son aide, ses encouragements

et sa présence tout au long de ce travail.

Nous lui sommes reconnaissants et particulièrement pour la confiance qu'il nous a

accordé et l'autonomie qu'il nous a donnée.

Nous tenons à remercier chaleureusement les membres du jury pour nous avoir accordée une partie de leur temps et pour leurs remarques

Constructives.

A toute l'administration et touts les enseignants de l'Université de Tlemcen qui nous ont encadrés durant notre cycle de formation.

Enfin, nous tenons à remercier nos parents, nos familles et nos amis pour leurs soutiens et leur encouragement.

Dédicaces

Au dieu tout puissant mon créateur.

A mon père, en signe d'amour, de reconnaissance et de gratitude pour les soutiens et les sacrifices dont il a fait preuve a mon égard.

A ma mère, ma raison d'être, ma raison de vivre.

A mes Amie, et à tous mes proches.

Je voudrais dédier ce travail a mes enseignants et encadreurs qui on su patiemment m'orienter vers mes recherches, me conseiller de puis de long mois.

Table des matières

Remerciement ....
Dédicace..
Résumé ....
Table de matière ....
Introduction général..

CHAPITRE 1 Etat d'art de l'architecture distribuée

Introduction..1
1. **L'architecture client-serveur** 1
1.1 Définition 1
1.2 Les principes généraux 2
1.3 La répartition des tâches 2
1.4 Les différents modèles de client-serveur 3
1.4.1 Le client-serveur de donnée 3
1.4.2 Client-serveur de présentation 3
1.4.3 Le client-serveur de traitement 3
1.4.4 Une synthèse des différents cas 3
1.5 Les différentes architectures 6
1.5.1 L'architecture un tiers..6
1.5.1.1 Présentation..6
1.5.1.2 Les solutions sur site central (mainframe) 6
1.5.1.3 Les applications un tiers déployées 7
1.5.1.4 Limitation 8
1.5.2 L'architecture 2 tiers 8
1.5.3 L'architecture 3 tiers 10
1.5.4 L'architecture n-tiers 11
1.6 Les middleware 12
1.6.1 Présentation 12
1.6.2 Les services des middlewares 13

1.6.3 Exemples de Middleware 13
1.6.4 Les Middleware objet 13
2. Le cas de l'Internet 14
2.1 Répartition des tâches 14
2.2 Le client universel.. 15
2.3 Les technologies coté client ou serveur 16
2.4 Le futur... 16
3. Le rôle de l'approche objet 16
3.1 L'approche objet 16
3.2 Les objets métier.. 17
4. La communication entre objets 18

CHAPITRE 2 *Ingénierie des composantes*

Introduction... 20
1. Concept de composants 21
1.1. Notions et définitions 21
1.2 Les trois dimensions d'un composant.. 23
1.3 La réutilisation d'un composant 25
2. Cycle de vie d'un composant 25
3. Représentation d'un composant... 26
4. Conclusion... 27

CHAPITRE 3 *Plate-forme JEE*

1. Introduction à J2EE 28
1.1 Principes de J2EE 28
1.2 Technologies de composants utilisées dans J2EE 30
1.2.1 Clients J2EE 32
1.2.2 Composants web 33
1.2.3 Composants Enterprise JavaBeans 35

2. Topologie(s) d'une application J2EE 35
3. Topologie(s) J2EE 35
3. 1 Remote Method Interface (RMI) 36
3. 2 JavaBeans 37
3.3 Java Naming and Directory Interface (JNDI) 37
3.4 Java DataBase Connectivity (JDBC) 38
3.5 Servlets 40
3.6 Java Server Pages 40
3.7 Enterprise Java Beans 41

CHAPITRE 4 *Les entreprises java beans*

1. Introduction **43**
2. L'historique des EJB **43**
3. Les nouveaux concepts et fonctionnalités utilisés 44
3.1 L'utilisation de POJO et POJI 45
3.2 L'utilisation des annotations 47
4. EJB 2.x vs EJB 3.0 .... **49**
5. Les types des ejb **50**
5.1 Les ejb de type Session 50
5.1.1 L'interface distante et /ou local 51
5.1.2 Les beans de type stateless 51
5.1.3 Les beans de type stateful 54
5.2 Les ejb de type Entity 55
5.2.1 La création d'un bean entity 56
5.2.2 La persistance des entités 59
5.3 Les EJB de type MessageDriven 60
5.3.1 L'annotation @ javax.ejb.MessageDriven 60
5.3.2 L'annotation @ javax.ejb.ActivationConfigProperty 61

CHAPITRE 5 ***Modélisation d'une application e-commerce***

1. Introduction 62
2. Les applications e-commerces 62
2.1 Définition d'un site e-commerce 62
2.2 Objectifs d'un site e-commerce 63
3. Description de notre application 63
4. Processus de développement de l'application 64
5. La démarche de développement 64
5.1 Les étapes de développement d'un logiciel 64
5.2 Modélisation 65
Diagramme de cas d'utilisation 65
Diagramme de séquence 68
Diagramme de classes d'analyse 70
Diagramme de classe participantes 72
Diagramme d'activités 74
Diagramme de classes de conception préliminaire 75
Diagramme de classes de conception détaillée 79
6. Conclusion 80
Conclusion générale 81
Liste des figures 82
Liste des abréviations 83
Bibliographie 84

Etre présent sur internet et devenue une réalité fréquente de nos jour.de ce fait toute les entreprise qui ce respecte a assurer sa présence sur réseau mondial, pour cella web représente une source de données et d'information interrogée par un grand nombre d'internaute.les requérants du web sont de profils très varies et ont donc des objectifs différent, et qui partagent de nombreux services disponibles sur le web par exemple : les site e-commerce, les bibliothèque numériques, les musée virtuelles, les journaux numérique…Ets.

Depuis quelques années, la modélisation objet avec le langage UML est devenue incontournable sur la plupart des projets informatique. Alors pourquoi ne pas appliquer ce qui marche pour les projets classique aux projets de site marchands ? Les applications web sont justement des candidates idéales à la modélisation et à l'application d'un processus de développement formalisé.

Dans cet esprit, notre objectif est la modélisation d'un site web marchand. Vente des matérielles informatique en ligne facile a comprendre et suffisamment représentatif des projets e-commerce

Chapitre 1

Etat de l'art des architectures distribuées

Introduction

Ces vingt dernières années ont vues une évolution majeure des systèmes d'information, à savoir le passage d'une architecture centralisée à travers de grosses machines (des Mainframe) vers une architecture distribuée basée sur l'utilisation de serveurs et de postes clients grâce à l'utilisation des PC et des réseaux.

Cette évolution a été possible essentiellement grâce à 2 facteurs qui sont :

- la baisse des prix de l'informatique personnelle
- le développement des réseaux.

1. L'architecture client-serveur

1.1 Définition

L'architecture client-serveur est un modèle de fonctionnement logiciel qui peut se réaliser sur tout type d'architecture matérielle (petites ou grosses machines), à partir du moment ou ces architectures peuvent être interconnectées.

On parle de fonctionnement logiciel dans la mesure où cette architecture est basée sur l'utilisation de deux types de logiciels, à savoir un logiciel serveur et un logiciel client s'exécutant normalement sur 2 machines différentes. L'élément important dans cette architecture est l'utilisation de mécanismes de communication entre les 2 applications.

Le dialogue entre les applications peut se résumer par :

- Le client demande un service au serveur
- Le serveur réalise ce service et renvoie le résultat au client

Un des principes fondamental est que le serveur réalise un traitement pour le client.

1.2 Les principes généraux

Il n'y a pas véritablement de définition exhaustive de la notion de client-serveur, néanmoins des principes régissent ce que l'on entend par client-serveur :

- **Service.**

Le serveur est fournisseur de services. Le client est consommateur de services.

- **Protocole.**

C'est toujours le client qui déclenche la demande de service. Le serveur attend passivement les requêtes des clients.

- **Partage des ressources**.

Un serveur traite plusieurs clients en même temps et contrôle leurs accès aux ressources.

- **Localisation.**

Le logiciel client-serveur masque aux clients la localisation du serveur.

- **Hétérogénéité.**

Le logiciel client-serveur est indépendant des plates-formes matérielles et logicielles.

- **Redimensionnement.**

Il est possible d'ajouter et de retirer des stations clientes. Il est possible de faire évoluer les serveurs.

- **Intégrité.**

Les données du serveur sont gérées sur le serveur de façon centralisée. Les clients restent individuels et indépendants.

- **Souplesse et adaptabilité**.

On peut modifier le module serveur sans toucher au module client. La réciproque est vraie. Si une station est remplacée par un modèle plus récent, on modifie le module client (en améliorant l'interface, par exemple) sans modifier le module serveur.

1.3 La répartition des tâches

Dans l'architecture client-serveur, une application est constituée de trois parties :

- l'interface utilisateur
- la logique des traitements
- la gestion des données.

Le client n'exécute que l'interface utilisateur (souvent un interfaces graphique) ainsi que la logique des traitements (formuler la requête), laissant au serveur de bases de données la gestion complète des manipulations de données.

La liaison entre le client et le serveur correspond à tout un ensemble complexe de logiciels appelé middleware qui se charge de toutes les communications entre les processus.

1.4 Les différents modèles de client-serveur

En fait, les différences sont essentiellement liées aux services qui sont assurés par le serveur.

On distingue couramment :

1.4.1 Le client-serveur de donnée.

Dans ce cas, le serveur assure des tâches de gestion, stockage et de traitement de données. C'est le cas le plus connu de client-serveur est qui est utilisé par tous les grands SGBD.

La base de données avec tous ses outils (maintenance, sauvegarde …) est installée sur un poste serveur.

Sur les clients, un logiciel d'accès est installé permettant d'accéder à la base de données du serveur.

Tous les traitements sur les données sont effectués sur le serveur qui renvoie les informations demandées (souvent à travers une requête SQL) par le client.

1.4.2 Client-serveur de présentation

Dans ce cas la présentation des pages affichées par le client est intégralement prise en charge par le serveur. Cette organisation présente l'inconvénient de générer un fort trafic réseau.

1.4.3 Le client-serveur de traitement

Dans ce cas, le serveur effectue des traitements à la demande du client. Il peut s'agir de traitement particulier sur des données, de vérification de formulaires de saisie, de traitements d'alarmes …

Ces traitements peuvent être réalisés par des programmes installé sur des serveurs mais également intégrés dans des bases de données (triggers, procédures stockées), dans ce cas, la partie donnée et traitement sont intégrés.

1.4.4 Une synthèse des différents cas

Cette synthèse s'illustre par un schéma du Gartner Group qui représente les différents modèles ainsi que la répartition des tâches entre serveur et client.

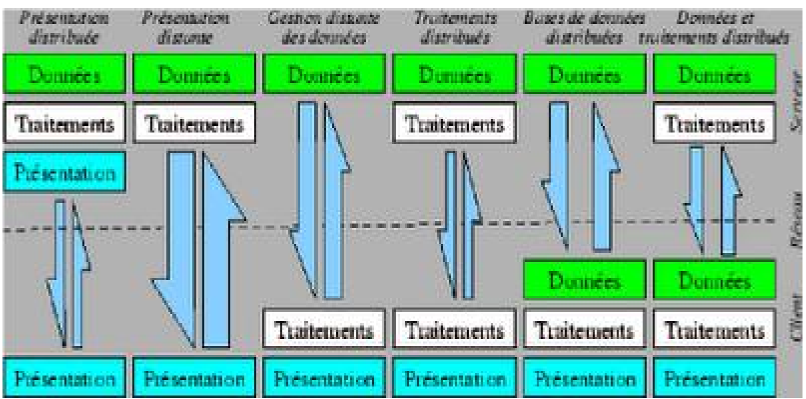


Figure 1.1: Les différents types de client-serveur selon la classification du Gartner Group

Sur ce schéma, le trait horizontal représente le réseau et les flèches entre client et serveur, le trafic réseau généré par la conversation entre client et serveur.

Nous verrons par la suite que la vision du Gartner Group, en ne prenant en compte qu'un découpage en deux niveaux, est quelque peu limitatif.

Le Gartner Group distingue les types de client-serveur suivants, en fonction du type de service déporté du cœur de l'application :

- **Présentation distribuée :** Correspond à l'habillage ``graphique' de l'affichage en mode caractères d'applications fonctionnant sur site central. Cette solution est aussi appelée revamping. La classification ``client-serveur' du revamping est souvent jugée abusive, du fait que l'intégralité des traitements originaux est conservée et que le poste client conserve une position d'esclave par rapport au serveur.

- **Présentation distante :** Encore appelée client-serveur de présentation. L'ensemble des traitements est exécuté par le serveur, le client ne prend en charge que l'affichage. Ce type d'application présentait jusqu'à présent l'inconvénient de générer un fort trafic réseau et de ne permettre aucune répartition de la charge entre client et serveur.

S'il n'était que rarement retenu dans sa forme primitive, il connaît aujourd'hui un très fort regain d'intérêt avec l'exploitation des standards Internet.

- **Gestion distante des données :** Correspond au client-serveur de données, sans doute le type de client-serveur le plus répandu. L'application fonctionne dans sa totalité sur le client, la gestion des données et le contrôle de leur intégrité sont assurés par un SGBD centralisé.

Cette architecture, de part sa souplesse, s'adapte très bien aux applications de type info centre, interrogeant la base de façon ponctuelle. Il génère toutefois un trafic réseau assez important et ne soulage pas énormément le poste client, qui réalise encore la grande majorité des traitements.

- **Traitement distribué :** Correspond au client-serveur de traitements. Le découpage de l'application se fait ici au plus près de son noyau et les traitements sont distribués entre le client et le(s) serveur(s).

Le client-serveur de traitements s'appuie, soit un mécanisme d'appel de procédure distante, soit sur la notion de procédure stockée proposée par les principaux SGBD du marché.

Cette architecture permet d'optimiser la répartition de la charge de traitement entre machines et limite le trafic réseau. Par contre il n'offre pas la même souplesse que le client-serveur de données puisque les traitements doivent être connus du serveur à l'avance.

- **Bases de données distribuées :** Il s'agit d'une variante du client-serveur de données dans laquelle une partie de données est prise en charge par le client. Ce modèle est intéressant si l'application doit gérer de gros volumes de données, si l'on souhaite disposer de temps d'accès très rapides sur certaines données ou pour répondre à de fortes contraintes de confidentialité.

Ce modèle est aussi puissant que complexe à mettre en œuvre.

- **Données et traitements distribués**. Ce modèle est très puissant et tire partie de la notion de composants réutilisables et distribuables pour répartir au mieux la charge entre client et serveur.

C'est, bien entendu, l'architecture la plus complexe à mettre en œuvre

1.5 Les différentes architectures

1.5.1 L'architecture un tiers

1.5.1.1 Présentation

Dans une application un tiers, les trois couches applicatives sont intimement liées et s'exécutent sur le même ordinateur. On ne parle pas ici d'architecture client-serveur, mais d'informatique centralisée.

Dans un contexte multi-utilisateurs, on peut rencontrer deux types d'architecture mettant en œuvre des applications un tiers :

- des applications sur site central,
- des applications réparties sur des machines indépendantes communiquant par partage de fichiers.

1.5.1.2 Les solutions sur site central (mainframe)

Historiquement, les applications sur site central furent les premières à proposer un accès multiutilisateurs.

Dans ce contexte, les utilisateurs se connectent aux applications exécutées par le serveur central (le mainframe) à l'aide de terminaux passifs se comportant en esclaves. C'est le serveur central qui prend en charge l'intégralité des traitements, y compris l'affichage qui est simplement déporté sur des terminaux passifs

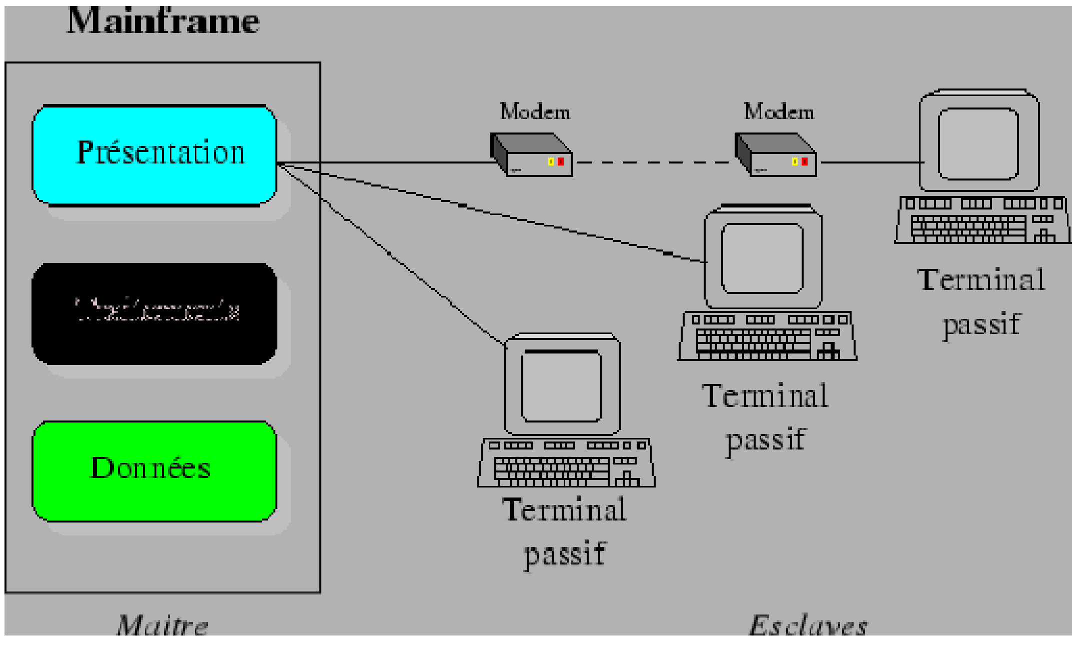


Figure 1.2: Architecture d'une application sur site central

Ce type d'organisation brille par sa grande facilité d'administration et sa haute disponibilité.
Il bénéficie aujourd'hui d'une large palette d'outils de conception, de programmation et d'administration ayant atteint un niveau de maturité et de fiabilité leur permettant encore de soutenir la comparaison avec des solutions beaucoup plus modernes. De plus, la centralisation de la puissance sur une seule et même machine permet une utilisation optimale des ressources et se prête très bien à des traitements par lots (en *batch*).
L'émergence des interfaces utilisateur de type *Windows - Macintosh*, démocratisées par les outils bureautiques sur micro-ordinateur, a fortement démodé les applications *mainframe*.
Ces dernières exploitaient exclusivement une interface utilisateur en mode caractère jugée obsolète par les utilisateurs, qui réclamaient alors massivement une convergence entre la bureautique et le système d'information de l'entreprise.
Le ré-habillage d'application centralisée, ou *revamping*, permet de rajeunir ces dernières en leur faisant profiter d'une interface graphique moderne (*GUI*), mais au prix d'une telle lourdeur qu'il doit être considéré comme une simple étape vers une architecture plus moderne.

1.5.1.3 Les applications un tiers déployées

Avec l'arrivée dans l'entreprise des premiers PC en réseau, il est devenu possible de déployer une application un tiers sur plusieurs ordinateurs indépendants.
Ce type de programme est simple à concevoir et à mettre en œuvre.
Il existe pour cela de nombreux environnements de développement (*dBase, Ms Access, Lotus Approach, Paradox...*) qui sont souvent intégrés aux principales suites bureautiques. L'ergonomie des applications mises en œuvre, basée sur celle des outils bureautiques, est très riche.
Ce type d'application peut être très satisfaisant pour répondre aux besoins d'un utilisateur isolé et sa mise en œuvre dans un environnement multi-utilisateur est envisageable.
Dans ce contexte, plusieurs utilisateurs se partagent des fichiers de données stockés sur un serveur commun. Le moteur de base de données est exécuté indépendamment sur chaque poste client. La gestion des conflits d'accès aux données doit être prise en charge par chaque programme de façon indépendante, ce qui n'est pas toujours évident.
Lors de l'exécution d'une requête, l'intégralité des données nécessaires doit transiter sur le réseau et on arrive vite à saturer ce dernier. De plus, la cohabitation de plusieurs moteurs de

base de données indépendants manipulant les mêmes données peut devenir assez instable. Il n'est pas rare de rencontrer des conflits lors de la consultation ou de la modification Simultanée d'un même enregistrement par plusieurs utilisateurs. Ces conflits peuvent altérer l'intégrité des données. Enfin, il est difficile d'assurer la confidentialité des données.
Ce type de solution est donc à réserver à des applications non critiques exploitées par de petits groupes de travail (une dizaine de personnes au maximum).

1.5.1.4 Limitations

On le voit, les applications sur site central souffrent d'une interface utilisateur en mode caractères et la cohabitation d'applications micro exploitant des données communes n'est pas fiable au delà d'un certain nombre d'utilisateurs.
Il a donc fallu trouver une solution conciliant les avantages des deux premières :

- ✓ la fiabilité des solutions sur site central, qui gèrent les données de façon centralisée,
- ✓ l'interface utilisateur moderne des applications sur micro-ordinateurs.

Pour obtenir cette synthèse, il a fallu scinder les applications en plusieurs parties distinctes et coopérants :

- ✓ gestion centralisée des données,
- ✓ gestion locale de l'interface utilisateur.

Ainsi est né le concept du client-serveur

1.5.2 L'architecture 2 tiers

Dans une architecture deux tiers, encore appelée client-serveur de première génération ou client-serveur de données, le poste client se contente de déléguer la gestion des données à un service spécialisé. Le cas typique de cette architecture est une application de gestion fonctionnant sous Windows ou Linux et exploitant un SGBD centralisé.
Ce type d'application permet de tirer partie de la puissance des ordinateurs déployés en réseau pour fournir à l'utilisateur une interface riche, tout en garantissant la cohérence des données, qui restent gérées de façon centralisée.
La gestion des données est prise en charge par un SGBD centralisé, s'exécutant le plus souvent sur un serveur dédié. Ce dernier est interrogé en utilisant un langage de requête qui, plus souvent, est SQL. Le dialogue entre client et serveur se résume donc à l'envoi de requêtes et au retour des données correspondant aux requêtes.

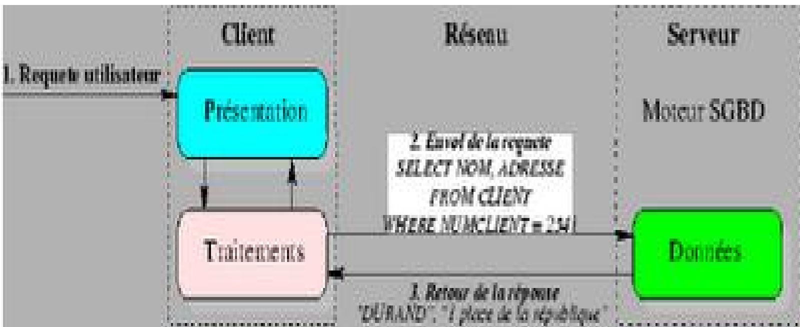


Figure 1.3 : architecture deux tiers

Cet échange de messages transite à travers le réseau reliant les deux machines. Il met en œuvre des mécanismes relativement complexes qui sont, en général, pris en charge par un middleware.

L'expérience a démontré qu'il était coûteux et contraignant de vouloir faire porter l'ensemble des traitements applicatifs par le poste client. On en arrive aujourd'hui à ce que l'on appelle le client lourd, avec un certain nombre d'inconvénients :

On ne peut pas soulager la charge du poste client, qui supporte la grande majorité des traitements applicatifs,

Le poste client est fortement sollicité, il devient de plus en plus complexe et doit être mis à jour régulièrement pour répondre aux besoins des utilisateurs,

Les applications se prêtent assez mal aux fortes montées en charge car il est difficile de modifier l'architecture initiale,

La relation étroite qui existe entre le programme client et l'organisation de la partie serveur complique les évolutions de cette dernière,

Ce type d'architecture est grandement rigidifié par les coûts et la complexité de sa maintenance.

Malgré tout, l'architecture deux tiers présente de nombreux avantages qui lui permettent de présenter un bilan globalement positif :

Elle permet l'utilisation d'une interface utilisateur riche,

Elle a permis l'appropriation des applications par l'utilisateur,

Elle a introduit la notion d'interopérabilité.

Pour résoudre les limitations du client-serveur deux tiers tout en conservant ses avantages, on a cherché une architecture plus évoluée, facilitant les forts déploiements à moindre coût. La réponse est apportée par les architectures distribuées.

1.5.3 L'architecture 3 tiers

Les limites de l'architecture deux tiers proviennent en grande partie de la nature du client utilisé :

Le frontal est complexe et non standard (même s'il s'agit presque toujours d'un PC sous Windows),

Le middleware entre client et serveur n'est pas standard (dépend de la plate-forme, du SGBD …).

La solution résiderait donc dans l'utilisation d'un poste client simple communicant avec le serveur par le biais d'un protocole standard.

Dans ce but, l'architecture trois tiers applique les principes suivants :

Les données sont toujours gérées de façon centralisée,

La présentation est toujours prise en charge par le poste client,

La logique applicative est prise en charge par un serveur intermédiaire.

Cette architecture trois tiers, également appelée client-serveur de deuxième génération ou client-serveur distribué sépare l'application en 3 niveaux de services distincts, conformes au principe précédent :

Premier niveau : l'affichage et les traitements locaux (contrôles de saisie, mise en forme de données...) sont pris en charge par le poste client,

- Deuxième niveau : les traitements applicatifs globaux sont pris en charge par le service applicatif,
- Troisième niveau : les services de base de données sont pris en charge par un SGBD.

Figure 1.4 : architecture trois tiers

Tous ces niveaux étant indépendants, ils peuvent être implantés sur des machines différentes, de ce fait :

- Le poste client ne supporte plus l'ensemble des traitements, il est moins sollicité et peut être moins évolué, donc moins coûteux,

- Les ressources présentes sur le réseau sont mieux exploitées, puisque les traitements applicatifs peuvent être partagés ou regroupés (le serveur d'application peut s'exécuter sur la même machine que le SGBD),
- La fiabilité et les performances de certains traitements se trouvent améliorées par leur centralisation,
- Il est relativement simple de faire face à une forte montée en charge, en renforçant le service applicatif.

Dans l'architecture trois tiers, le poste client est communément appelé client léger ou Thin Client, par opposition au client lourd des architectures deux tiers. Il ne prend en charge que la présentation de l'application avec, éventuellement, une partie de logique applicative permettant une vérification immédiate de la saisie et la mise en forme des données.

Le serveur de traitement constitue la pierre angulaire de l'architecture et se trouve souvent fortement sollicité. Dans ce type d'architecture, il est difficile de répartir la charge entre client et serveur. On se retrouve confronté aux épineux problèmes de dimensionnement serveur et de gestion de la montée en charge rappelant l'époque des mainframes.

De plus, les solutions mises en œuvre sont relativement complexes à maintenir et la gestion des sessions est compliquée.

Les contraintes semblent inversées par rapport à celles rencontrées avec les architectures deux tiers : le client est soulagé, mais le serveur est fortement sollicité.

1.5.4 L'architecture n-tiers

L'architecture n-tiers a été pensée pour pallier aux limitations des architectures trois tiers et concevoir des applications puissantes et simples à maintenir. Ce type d'architecture permet de distribuer plus librement la logique applicative, ce qui facilite la répartition de la charge entre tous les niveaux.

Cette évolution des architectures trois tiers met en œuvre une approche objet pour offrir une plus grande souplesse d'implémentation et faciliter la réutilisation des développements.

Théoriquement, ce type d'architecture supprime tous les inconvénients des architectures précédentes :

- ✓ Elle permet l'utilisation d'interfaces utilisateurs riches,
- ✓ Elle sépare nettement tous les niveaux de l'application,
- ✓ Elle offre de grandes capacités d'extension,
- ✓ Elle facilite la gestion des sessions.

L'appellation ``n-tiers' pourrait faire penser que cette architecture met en œuvre un nombre indéterminé de niveaux de service, alors que ces derniers sont au maximum trois (les trois niveaux d'une application informatique). En fait, l'architecture n-tiers qualifie la distribution d'application entre de multiples services et non la multiplication des niveaux de service.

Cette distribution est facilitée par l'utilisation de composants ``métier', spécialisés et indépendants, introduits par les concepts orientés objets (langages de programmation et middleware). Elle permet de tirer pleinement partie de la notion de composants métiers réutilisables.

Ces composants rendent un service si possible générique et clairement identifié. Ils sont capables de communiquer entre eux et peuvent donc coopérer en étant implantés sur des machines distinctes.

La distribution des services applicatifs facilite aussi l'intégration de traitements existants dans les nouvelles applications. On peut ainsi envisager de connecter un programme de prise de commande existant sur le site central de l'entreprise à une application distribuée en utilisant un middleware adapté.

Ces nouveaux concepts sont basés sur la programmation objet ainsi que sur des communications standards entre application. Ainsi est né le concept de Middleware objet.

1.6 Les middleware

1.6.1 Présentation

On appelle middleware (ou logiciel médiateur en français), littéralement ``élément du milieu', l'ensemble des couches réseau et services logiciel qui permettent le dialogue entre les différents composants d'une application répartie. Ce dialogue se base sur un protocole applicatif commun, défini par l'API du middleware.

Le Gartner Group définit le middleware comme une interface de communication universelle entre processus. Il représente véritablement la clef de voûte de toute application client-serveur.

L'objectif principal du middleware est d'unifier, pour les applications, l'accès et la manipulation de l'ensemble des services disponibles sur le réseau, afin de rendre l'utilisation de ces derniers presque transparente.

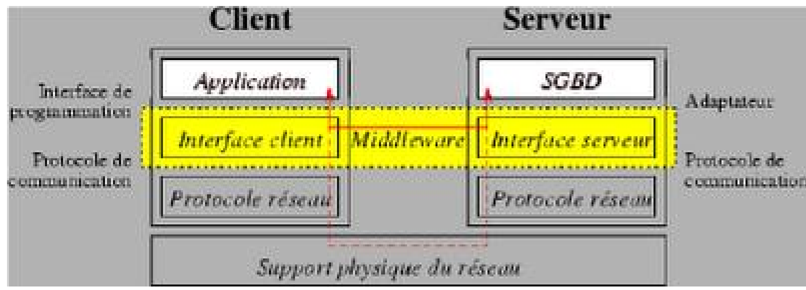


Figure 1. 5 : positionnement du middeware.

1.6.2 Les services des middlewares

Un middleware est susceptible de rendre les services suivants :

- **Conversion** : Service utilisé pour la communication entre machines mettant en œuvre des formats de données différents
- **Adressage** : Permet d'identifier la machine serveur sur laquelle est localisé le service demandé afin d'en déduire le chemin d'accès. Dans la mesure du possible
- **Sécurité** : Permet de garantir la confidentialité et la sécurité des données à l'aide de mécanismes d'authentification et de cryptage des informations.
- **Communication** : Permet la transmission des messages entre les deux systèmes sans altération. Ce service doit gérer la connexion au serveur, la préparation de l'exécution des requêtes, la récupération des résultats et la déconnexion de l'utilisateur.

Le middleware masque la complexité des échanges inter-applications et permet ainsi d'élever le niveau des API utilisées par les programmes. Sans ce mécanisme, la programmation d'une application client-serveur serait complexe et difficilement évolutive.

1.6.3 Exemples de Middleware

- **SQL*Net :** Interface propriétaire permettant de faire dialoguer une application cliente avec une base de données Oracle. Ce dialogue peut aussi bien être le passage de requêtes SQL que l'appel de procédures stockées.
- **ODBC :** (Object Data Base Connexion) Interface standardisée isolant le client du serveur de données. C'est l'implémentation par Microsoft d'un standard défini par le SQL Access Group. Elle se compose d'un gestionnaire de driver standardisé, d'une API s'interfaçant avec l'application cliente (sous Ms Windows) et d'un driver correspondant au SGBD utilisé.

- **DCE:** (Distributions Computing environment) Permet l'appel à des procédures distantes depuis une application. Correspond à RPC (Remote Procedure Call) qui permet d'exécuter des procédures distantes.

Le choix d'un middleware est déterminant en matière d'architecture, il joue un grand rôle dans la structuration du système d'information.

Pour certaines applications devant accéder à des services hétérogènes, il est parfois nécessaire de combiner plusieurs middlewares. On en vient à la notion de client lourd.

1.6.4 Les Middleware objet

Pour permettre la répartition d'objets entre machines et l'intégration des systèmes non objets, il doit être possible d'instaurer une communication entre tous ces éléments. Ainsi est né le concept de middleware objet qui a donné naissance à plusieurs spécifications, dont l'architecture CORBA 5common Object Request Broker Architecture) préconisée par l'OMG (Object Management Group) et DCOM développée par Microsoft.

Ces middlewares sont constitués d'une série de mécanismes permettant à un ensemble de programmes d'inter opérer de façon transparente. Les services offerts par les applications serveurs sont présentés aux clients sous la forme d'objets. La localisation et les mécanismes mis en œuvre pour cette interaction sont cachés par le middleware.

La communication entre objets gomme la différence entre ce qui est local ou distant. Les appels de méthodes d'objet à objet sont traités par un mécanisme se chargeant d'aiguiller les messages vers les objets (locaux ou distants).

2. Le cas de l'Internet

Dans le cadre d'un Intranet ou d'un Extranet, le poste client prend la forme d'un simple navigateur Web, le service applicatif est assuré par un serveur HTTP et la communication avec le SGBD met en œuvre les mécanismes bien connus des applications client-serveur de la première génération.

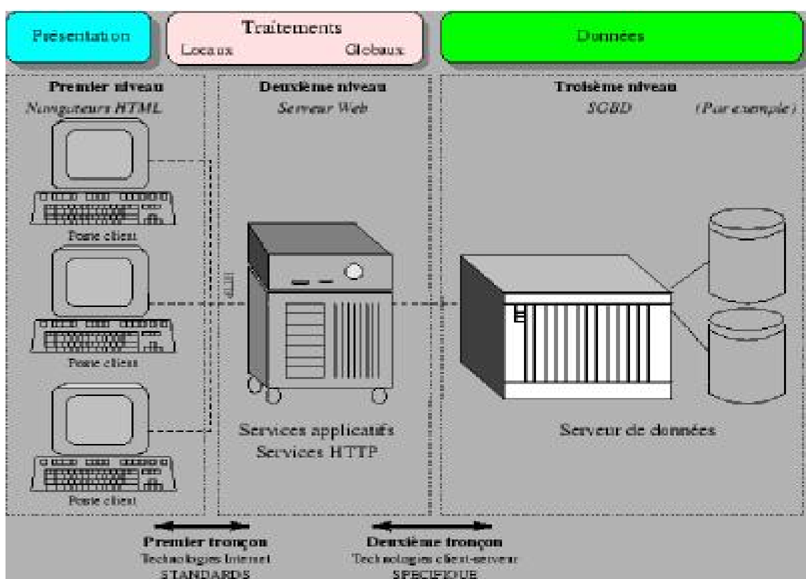
2.1 Répartition des tâches

Ce type d'architecture fait une distinction nette entre deux tronçons de communication indépendants et délimités par le serveur HTTP :

- le premier tronçon relie le poste client au serveur Web pour permettre l'interaction avec l'utilisateur et la visualisation des résultats. Ce premier tronçon

n'est composé que de standards (principalement HTM L et HTTP) et est basé sur un simple navigateur Web. Le serveur Web tient le rôle de ``frontal HTTP',

- le deuxième tronçon permet la collecte des données. Les mécanismes utilisés sont comparables à ceux mis en œuvre pour une application deux tiers. Ils ne franchissent jamais la façade HTTP et, de ce fait, peuvent évoluer sans influence sur la configuration des postes clients.



2.2 Le client universel

Ce type d'architecture est définie par certain comme le client-serveur universel dans la mesure où il s'appuie sur des standards existant sur toutes les plates-formes.

De plus, dans la même approche, on peut dire au vu de l'explosion des architectures Internet que le navigateur Web est le client universel, puisque de plus en plus utilisé pour tout type de développement, ou d'interface.

L'avantage de ce client universel est qu'il est aujourd'hui présent sur tout type de machine :

- Postes de bureau
- Portables
- Pockets PC
- Téléphones portables

De plus, tous les constructeurs de produits embarqués embarquent dans leurs applicatifs un serveur http qui permet d'administrer facilement à distance leurs machines.

2.3 Les technologies coté client ou serveur

Pour revenir aux architectures 3 tiers de l'Internet, cette solution peut s'appuyer sur toutes les nouvelles technologies, et permet de répartir en fonction des besoins les traitements coté client ou coté serveur.

Les technologies coté client font appel à :

- du JavaScript
- des applets JAVA
- des composants ACTIVEX (Microsoft)

Coté serveur, toutes les techniques permettant de faire du Web dynamique à savoir :

- des scripts CGI (Perl, C, C++
- des servlets écrit en JAVA : JSP (Java Server Pages)
- des traitements en ASP (Active Server Pages) - technologie Microsoft
- du code PHP

2.4 Le futur

Le futur appelé par certains Web 2.0 s'oriente néanmoins vers des clients plus riche afin de pouvoir bénéficier d'interfaces plus conviviaux, tout en gardant le coté universel du poste client. Un exemple en est donné aujourd'hui à travers certains clients de messagerie (Yahoo, MSN).

3. Le rôle de l'approche objet

3.1 L'approche objet

Les évolutions successives de l'informatique permettent de masquer la complexité des mécanismes mis en œuvre derrière une approche de plus en plus conceptuelle.

Ainsi, les langages de programmation ont tout d'abord été très proches de la machine (langage machine de bas niveau), puis procéduraux et, enfin, orientés objets.

Le succès du langage Java a véritablement popularisé ce mode de programmation.

Les protocoles réseau ont suivi le même type d'évolution. Ils furent d'abord très proches de la couche physique, avec les mécanismes de sockets orientés octets. Ensuite, la notion de RPC a permis de faire abstraction des protocoles de communication et, ainsi, a facilité la mise en place d'application client-serveur. Aujourd'hui, l'utilisation d'ORB7.1 permet une totale transparence des appels distants et permet de manipuler un objet distant comme s'il était local.

Le flux d'informations fut donc initialement constitué d'octets, puis de données et, enfin, de messages.

Les méthodes de conception orientées objet telles qu'*UML* ou *OMT* permettent une modélisation plus concrète des besoins et facilitent le passage de la conception à la réalisation. Aucune de ces évolutions ne constitue en soit une révolution, mais elles rendent économiquement réalisables des architectures qui n'étaient jusqu'à présent que techniquement envisageables.

3.2 Les objets métier

Les Java Beans

Selon les spécifications de Sun, un Java Beans est un composant logiciel réutilisable qui peut être manipulé par un outil d'assemblage. Cette notion assez large englobe aussi bien un simple bouton qu'une application complète. Concrètement, les Java Beans sont des classes Java utilisant des interfaces particulières.

Un Java Beans peut être utilisé indépendamment, sous la forme d'une simple applet, ou intégré à un développement Java. Il peut dévoiler son comportement à ses futurs utilisateurs à l'aide :

- des propriétés qu'il expose et rend accessible à l'aide d'accesseurs,
- des méthodes qu'il permet d'invoquer, comme tout objet Java,
- des événements qu'il peut générer pour avertir d'autres composants.

Le mécanisme d'introspection permet une analyse automatique du composant qui, ainsi, s'auto-décrit.

Les Java Beans proposent deux modes de fonctionnement :

- le mode de configuration permettant de paramétrer l'intégration du composant à l'aide d'un outil d'assemblage. Dans ce mode, le Java Beans propose une interface graphique de configuration,

Le mode d'exécution utilisé par l'application intégrant le Java Beans.

- Les Java Beans sont gérés comme tout objet Java, notamment ce qui concerne son cycle de vie. Ils sont livrés sous forme de fichiers JAR7.2.

Les EJB (Entreprise Java Beans)

Les Enterprise Java Beans sont des Java Beans destinés à s'exécuter côté serveur :

- ils ne comportent pas forcément de partie visible,
- ils prennent en charge des fonctions de sécurité, de gestion des transactions et d'état,

- ils peuvent communiquer avec des Java Beans côté client de façon indépendante du protocole (IIOP7.3, RMI7.4, DCOM7.5),
- ils s'exécutent dans un environnement ``Beanstalk'' s'appuyant sur l'API Java Server et offrant des fonctionnalités de gestion de la charge d'exécution, de la sécurité d'accès, de communication, d'accès aux services transactionnels.

Microsoft OLE-COM-ActiveX

Le modèle de communication COM permet de mettre en place une communication orientée objet entre des applications s'exécutant sur une même machine.

DCOM est une extension de ce modèle pour les architectures distribuées.

Il repose sur le modèle DCE7.6 défini par l'OSF7.7 et met en œuvre un serveur d'objets situé sur chaque machine.

Les contrôles ActiveX, anciennement dénommés OCX, sont des composants logiciels basés sur le modèles COM. Ils peuvent être intégrés à des applications où à des documents sous Windows.

4. La communication entre objets

Principes

Pour permettre la répartition d'objets entre machines et l'intégration des systèmes non objets, il doit être possible d'instaurer une communication entre tous ces éléments.

Ainsi est né le concept de middleware objet qui a donné naissance à plusieurs spécifications, dont l'architecture CORBA7.8 préconisée par l'OMG7.9 et DCOM développée par Microsoft.

Ces middlewares sont constitués d'une série de mécanismes permettant à un ensemble de programmes d'interopérer de façon transparente. Les services offerts par les applications serveurs sont présentés aux clients sous la forme d'objets. La localisation et les mécanismes mis en œuvre pour cette interaction sont cachés par le middleware.

La communication entre objets gomme la différence entre ce qui est local ou distant. Les appels de méthodes d'objet à objet sont traités par un ORB7.10 se chargeant d'aiguiller les messages vers les objets (locaux ou distants).

Il est possible d'encapsuler des applications ``non objet'' existantes pour les rendre accessibles via le middleware objet. Ainsi, on préserve l'existant sans alourdir les nouveaux développements, qui ne voient que l'interface de l'application encapsulée. La vision du système s'effectue ainsi de manière unifiée, chaque composant étant accessible via le

middleware. Un client peut désormais accéder de la même façon à un EJB ou à une transaction sur mainframe.

L'appel de procédure distante (RMI)

RMI permet à une application Java, s'exécutant sur une machine virtuelle, d'invoquer les méthodes d'un objet hébergé par une machine distante. Pour cela, le client utilise une représentation locale de l'interface de l'objet serveur. Cette représentation locale est appelée stub et représente l'interface de l'objet serveur, appelée skeleton.

Un objet distribué se caractérise par son interface et son adresse (URL). La mise en relation des objets est assurée par un serveur de noms.

Le modèle CORBA

Le système CORBA permet, au travers du protocole IIOP, l'utilisation d'objets structurés dans un environnement hétérogène.

Cette communication, orchestrée par l'ORB, est indépendante des contraintes systèmes des différentes plates-formes matérielles.

Une application accède à un objet distant en utilisant une télécommande locale, appelée proxy.

Ce proxy lui permet de déclencher les méthodes de l'objet distant à l'aide de primitives décrites avec le langage IDL7.11.

L'OMG effectue toute une série de recommandations connues sous le nom de CORBA services visant à proposer des interfaces génériques pour chaque type de service usuel (nommage, transaction, cycle de vie, ...).

Chapitre 2

Ingénierie des composantes

Introduction

Aujourd'hui, nous vivons une nouvelle évaluation dans l'art de concevoir des systèmes complexes. Après les technologies objets qui ont modifié profondément l'ingénierie des systèmes logiciels améliorant ainsi leur analyse, leur conception et leur développement, nous Entrons dans une nouvelle ère de conception de systèmes, l'orienté composant, qui a émergé au sein de la communauté de recherche « Architectures logicielles ». L'orienté composant consiste à concevoir et à développer des systèmes par assemblage de composants réutilisables à l'image, par exemple, des composants électroniques ou des composants mécaniques. Plus précisément, il s'agit de :

- ✓ Concevoir et développer des systèmes à partir de composants préfabriqués, préconçus et prétestés ;
- ✓ Réutiliser ces composants dans d'autres applications ;
- ✓ Faciliter leur maintenance et leur évolution ;
- ✓ Favoriser leur adaptabilité et leur configurabilité pour produire de nouvelles fonctionnalités et de nouvelles caractéristiques.

Cette nouvelle approche tente de répondre à la complexité croissante des systèmes informatiques, à leur évolution plus rapide, et suscite un intérêt accru pour le développement de logiciels à base de composants. Cet intérêt est motivé principalement par la réduction des coûts et des délais de développement des applications. En effet, on prend moins de temps à acheter, et donc à réutiliser, un composant que de concevoir, le coder, le tester, le debogguer et le documenter.

Cependant, plusieurs problèmes doivent encore être résolus avant qu'on parvienne à l'ubiquité du composant logiciel. En effet, l'utilisation de l'approche « composants » dans le développement d'applications « grandeur réelle » relève certaines insuffisances. Par exemple : quelle est la définition d'un composant ? Quelle est sa granularité, sa portée ? Comment peut –on distinguer et rechercher les composants de manière rigoureuse ? Comment peut-on les manipuler, assembler, installer dans des contextes matériels et logiciels variant dans le temps. Comment les administrer, les faire ? Etc.

De ce fait, la situation des langages à base de composants rappelle celle des langages à objets au début des années soixante-dix, c'est –à-dire une syntaxe disparate qui renvoie à une sémantique ambiguë.

Si l'approche « composants » est avant tout une façon de concevoir un système logiciel, elle sera à la base de la conception des générations futures de logiciels et constituera sans doute la seule réponse crédible à l'évolution nécessaire et permanente des architectures logicielles de demain.

Aujourd'hui ; un certain nombre de propositions commence à émerger sur les composants.

Ils sont pour la plupart orientés très fortement sur l'aspect implémentation (bas niveau) (EJB, COM/DCOM, NET, CCM, etc.). D'un autre côte les composants haut-niveau préconisés par les ADL (Architectures Description Langages) comme ACME, WRIGHT, RAPIDE, etc., perdent leurs atouts dès qu'ils sont projetés au niveau implémentation. Dans tous les cas, à terme, les enjeux sont l'amélioration de la qualité et la productivité, et l'industrialisation de la production de logiciels.

Aussi, devant une telle diversité de proposition et de solution, il était indispensable de faire le point sur la recherche et l'application concernant l'ingénierie des composants.

1. Concept de composants

1.1. Notions et définitions

Il existe différentes définitions du concept de composant dont on cite ci-dessous les plus répandues :

Definition 1: A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Cette définition est. valable à différents niveaux de compréhension. Elle décrit les caractéristiques d'un composant logiciel ainsi que ses dépendances avec l'environnement extérieur (définition de Szyperki [Szyp98]).

Une autre définition d'un composant est donnée par Meyer [Meye99)

Definition 2: A software component is a program element with the following two properties:

1. it may be used by other program elements or clients.
2. The clients and their authors do not be known to the component's anthers.

Enfin, Harris [HaEe99] a donné également une définition bien acceptée dans la communauté « architecture logicielle » :

Définition 3: un composant est un morceau de logiciel assez petit que l'on peut créer et maintenir, et assez grand pour que l'on puisse l'installer et en assurer le support .De plus،il est doté d'interfaces standards pour pouvoir interopérer.

A partir des définitions précédentes ،on déduit les propriétés suivantes،à savoir un composant est:

- **Autodescriptif :** capable de disposer d'un mécanisme d'introspection permettant de Connaître et de modifier dynamiquement ses caractéristiques ;
- **composable :** considéré comme une unité de composition،il est censée être connectable avec d'autres composants ;
- **configurable***:* Il doit être paramétrable via ses propriétés configurables selon un contexte d'exécution particulier;
- **réutilisation : i**l doit être une unité de réutilisation.une étape d'adaptation sera sans doute nécessaire selon le contexte d'exécution;
- **autonome :** il peut être déployé (c'est-à-dire diffusé et installé) et exécuté indépendamment des autres composants [Ouss03].

Il s'avère important de présenter les différents concepts pour la description des systèmes à base de composants:

- les composants sont définis comme des unités de composition qui assurent une fonction bien précise ils peuvent être déployés indépendamment comme ils peuvent être composés avec d'autres composants ILS possèdent ainsi un statut de réutilisation. Chaque composant possédés une ou plusieurs interfaces pour interagir avec son environnement:
- les interaction ou connecteurs :il existe différentes formes d'interaction entre composants comme les appels de procédures،les événements،....qui sont des exemples d'interaction simple،quant aux interactions complexes comme les protocoles،les connexions avec des bases de données etc.;Ils sont représentés par des connecteurs;
- les propriétés:elles représentent les informations sémantiques des composants et leur interactions;

Les contraintes /contrats:représentent les moyens permettant à un modèle d'architecture de rester valide durant sa durée de vie et de prendre en compte l'évolution et le remplacement des composants logiciels dans cette architecture ;

- **Une architecture :** elle définit un modèle pour décrire les composants et leurs interactions;

- **la composition⁄assembla :** permet de construire des applications complexes à partir de composants simples ،la composition permet de lier un composant demandant des services à un composant offrant ces services.

1.2 Les trois dimensions d'un composant

Un composant peut être de deux natures :

- ✓ Produit : il s'agit d'une entité « building block » autonome passive (entité logicielle ou entité conceptuelle) qu'il est possible d'adapter. Les composants logiciels et les bibliothèques de fonction mathématique en font partie ;
- ✓ Processus : un composant processus correspond à une suite d'actions qu'il faut réutiliser pour obtenir un produit final. Ces actions sont souvent encapsulées dans un processeur (ou unité de traitement). Un composant processus correspond en général à des fragments de démarche.

Au regard des travaux existants, un composant produit ou processus doit refléter trois dimensions comme le montre la figure 2.1 :

- ✓ Son niveau d'abstraction ;
- ✓ Son mode d'expression ;
- ✓ Son domaine ;

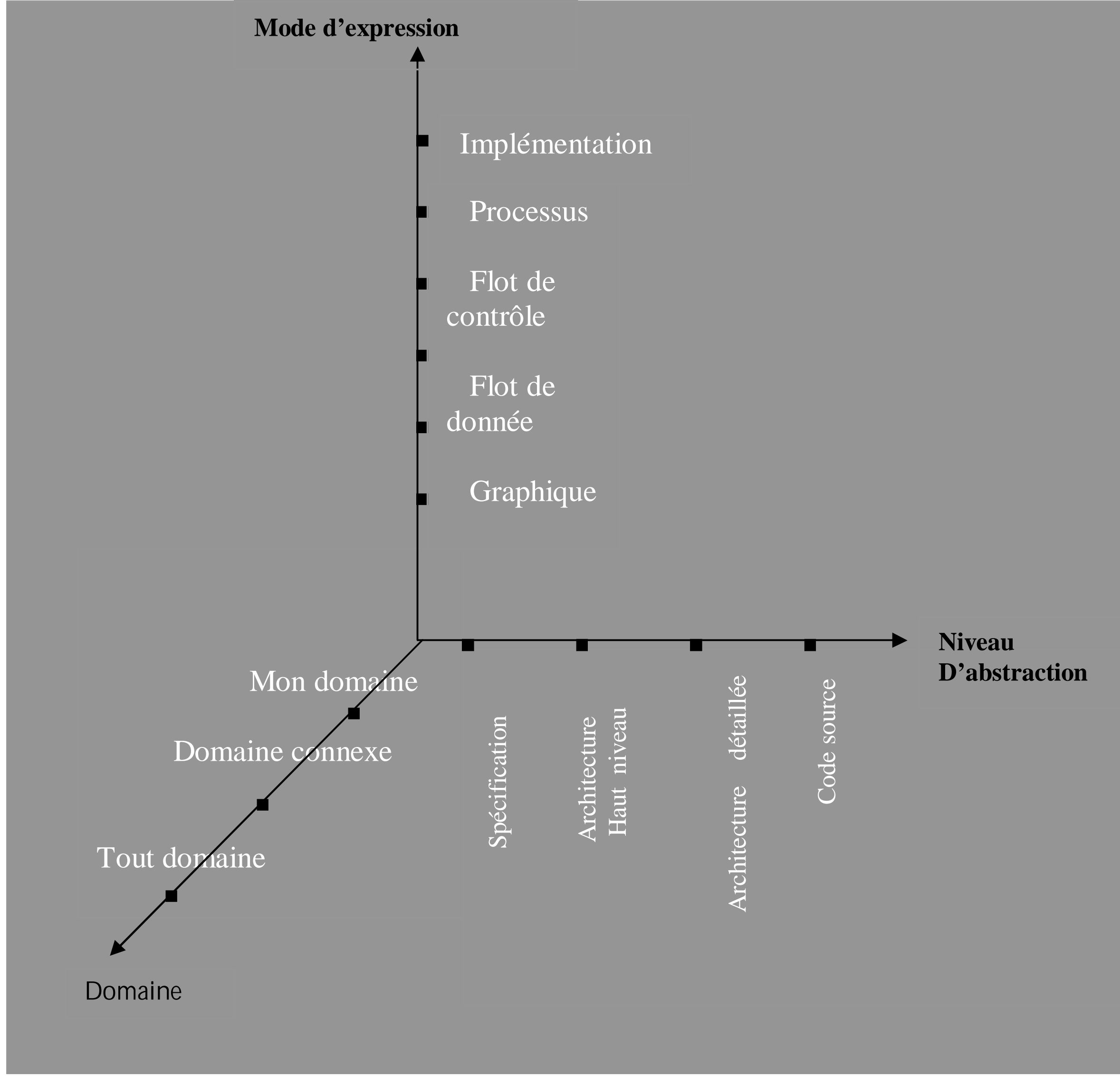


Figure 2.1 les trois dimensions d'un composant

La première dimension, ou niveau d'abstraction, permet d'exprimer les degrés de raffinement d'un composant.

La deuxième dimension, ou mode d'expression, permet de décrire le différent model de représentation d'un composant (représentation textuelle, graphique, flot de donnée, implémentation).

La troisième dimension, ou domaine, reflète les différents domaines structurés en couche et sur lesquels repose un système donné. Cette structuration permet à un composant appartenant

à une couche domaine particulière de ne pouvoir interagir qu'avec seulement les composants de la même couche ou d'une couche adjacente.

Les composants de « tout domaine » constituent la couche de plus bas niveau et sont indépendants du domaine d'application, c'est le cas par exemple des systèmes d'exploitation, de compilateurs, de systèmes de bases de donnée, etc. au niveau plus haut « domaine connexe », on trouve des composants appartenant à des domaines connexes à l'application en cours, par exemple des interfaces utilisateurs, des simulateurs. Enfin de la hiérarchie, les composants de « mon domaine » sont spécifiques au domaine d'application choisi, par exemple les composants de circuits intégrés.

1.3 La réutilisation d'un composant

L'idée de réutilisation d'un composant est simple du fait qu'on crée des applications à partir d'éléments existants. Ainsi, la réutilisation d'un composant permet de capitalises sur l'existant en réutilisant un maximum de composants, de même qu'elle favorise l'accélération du coût de développement de l'application.

2. Cycle de vie d'un composant

Le cycle de vie d'un composant suit les étapes de définition et d'analyse des besoins, de conception, d'implantation, de paquetage et diffusion des implantations, d'assemblage/composition, de déploiement et d'exécution.

- Définition et analyse des besoins en termes de composant

Cette étape est essentielle dans le processus de construction de logiciel puisqu'elle a comme objectif de produire les composants réutilisables.

- **Conception des types de composants**

La conception consiste à spécifier de composant à partir de l'étape précédente, cette étape est concernée par la représentation, l'organisation, l'accès, l'adaptation et l'intégration, l'utilisation d'un langage est importante. Ce langage servira à spécifier les trois aspects du composant (interfaces, propriétés, modes d'interaction).

- **Implantation des types de composants**

L'implantation d'un type de composant se résume à l'implantation des parties fonctionnelles. Elle doit se réaliser en respectant les spécifications du type de composant visé, en utilisant un langage de programmation qui varier d'un modèle de composant à un autre (EJB utilise java, etc.).

- **Paquetage et diffusion des implantations de composants**

La diffusion d'une implantation d'un composant se fait grâce au paquetage de cette implantation .une fois les composants implantés, ils sont d'abord archivés dans des paquetages. Un paquetage permet de se regrouper dans une même unité. La diffusion d'un composant nécessite de normaliser le format et le contenu des paquetages comme exemple (dans EJB et CCM, les paquetages sont respectivement de format JAR et ZIP)

- **Assemblage/composition**

L'assemblage consiste à assembler tous les composants d'une application, dans une même unité, qui peut être à son tour empaquetée et réutilisée.

- **Déploiement**

Le déploiement d'un composant consiste à extraire le composant de son paquetage et à l'installer dans son environnement, le processus de déploiement se base en général sur les descripteurs du composant contenus dans le paquetage pour instancier le composant.

- **Exécution utilisation**

Correspond à l'exploitation effective du composant, les acteurs potentiels de cette phase sont les utilisateurs finaux qui vont exploiter ces services.

3. Représentation d'un composant

Un composant est généralement représenté par les éléments suivants :

- ✓ **Sont type :** la définition abstraite du composant ;
- ✓ Sont implémentation : la mise en œuvre des aspects fonctionnels et non fonctionnels de son type ;
- ✓ **Son instance :** une entité exécutable dans un système définie par une référence unique ;

Un type de composant est caractérisé par deux éléments : ses interfaces avec leurs modes de connexion et ses propriétés configurables ou non.

- ✓ **Interface :** l'interface sert à déclarer les services fournis et requis et permet de faire le lien avec les autres composants. C'est la seule partie visible d'un composant.

Une interface est composée de deux éléments :

1. **Points de connexion**

Le point de connexion est le point d'interaction entre le composant et l'extérieur de composant.

2. services

Les services permettent d'exprimer la sémantique des fonctionnalités fournies et requises par le composant. On peut regrouper les services par catégorie, cela permet de faciliter la recherche de composant. Un service peut utiliser plusieurs points de connexion pour exécuter sa tache.

4. Conclusion

L'objectif de l'approche composant est de construire des systèmes bien structurés, qui soient compréhensibles de manière indépendante et qui constituent des blocs de construction réutilisables.

La conception et le développement d'un modèle de composants doivent répondre aux critères suivants :

- **l'adéquation :** le modèle doit être capable de représenter les concepts de base existant dans la plupart des outils de développement de composants logiciels tels que ActiveX, JavaBeans, etc.
- **l'expressivité :** les systèmes à base de composants peuvent être caractérisés par leur comportement, c'est -à-dire le type et l'organisation de leurs composants et des échanges d'information mais aussi l'évolution de leur état et de leur structure, les modèle de composant devraient être capable d'exprimer adéquatement l'ensemble de ces aspects, sans pour autant imposer des contraintes drastiques non réalistes sur les composants à représenter.
- **La clarté :** les modèle de composants doivent reposer sur un nombre minimal de concepts de base en définissant clairement les relations existantes entre eux, ce qui devrait faciliter la compréhension , la communication et l'application de ces modèles.

- **L'ubiquité :** les modèles de composants doivent fournir des mécanismes pour prendre en compte le développement d'application distribuées.

Chapitre 3

Plate-forme JEE

1. Introduction à J2EE

J2EE (Java 2 Entreprise Edition) est une norme proposée par la société Sun, portée par un consortium de sociétés internationales, visant à définir un standard de développement d'applications d'entreprises multi-niveaux, basées sur des composants.

On parle généralement de «plate-forme J2EE» pour désigner l'ensemble constitué des services (API) offerts et de l'infrastructure d'exécution. J2EE comprend notamment :

• Les spécifications du serveur d'application, c'est-à-dire de l'environnement d'exécution : J2EE définit finement les rôles et les interfaces pour les applications ainsi que l'environnement dans lequel elles seront exécutées. Ces recommandations permettent ainsi à des entreprises tierces de développer des serveurs d'application conformes aux spécifications ainsi définies, sans avoir à redévelopper les principaux services.

• Des services, au travers d'API, c'est-à-dire des extensions Java indépendantes permettant d'offrir en standard un certain nombre de fonctionnalités. Sun fournit une implémentation minimale de ces API appelée J2EE SDK (J2EE Software Development Kit).

Dans la mesure où J2EE s'appuie entièrement sur le Java, il bénéficie des avantages et inconvénients de ce langage, en particulier une bonne portabilité et une maintenabilité du code.

De plus, l'architecture J2EE repose sur des composants distincts, interchangeables et distribués, ce qui signifie notamment :

• Qu'il est simple d'étendre l'architecture.

• Qu'un système reposant sur J2EE peut posséder des mécanismes de haute disponibilité, afin de garantir une bonne qualité de service.

• Que la maintenabilité des applications est facilitée

1.1 Principes de J2EE

L'architecture J2EE est une architecture d'application distribuée à base de composants. Elle identifie et donne les spécifications des composants de l'application :

- ✓ composants logiciels ou *beans* (EJB),
- ✓ conteneur
- ✓ serveurs
- ✓ clients

Les conteneurs isolent les beans du client et d'une implémentation spécifique du serveur.Les beans sont installés dans la partie serveur d'une application J2EE.

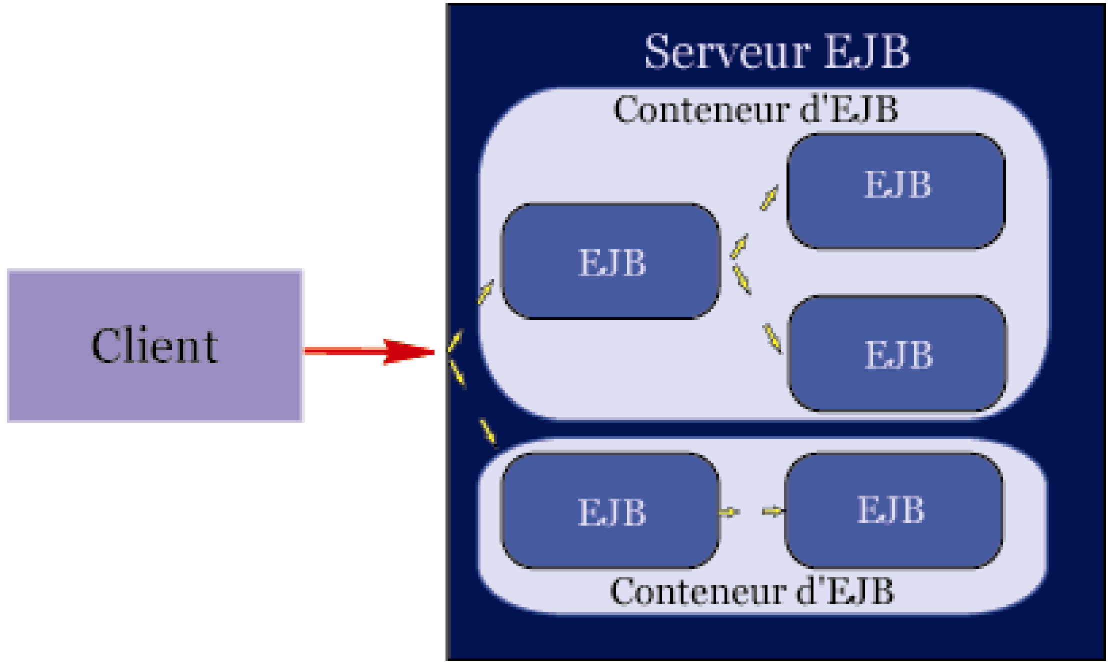


Figure 3.1 : Communications au sein d'une application J2EE - vue générale

Les conteneurs et serveurs implémentent les mécanismes un bas niveau utilisé par les applications :

- ✓ Transactions,
- ✓ Persistance,
- ✓ Gestion de la mémoire,
- ✓ Sécurité

Les spécifications J2EE s'intéressent aux activités d'une application liées :

- ✓ au développement,
- ✓ au déploiement,
- ✓ à l'exécution

1.2 Technologies de composants utilisées dans J2EE

Un composant est une unité logicielle de niveau applicatif. En plus des JavaBeans, qui font partie du J2SE, J2EE supporte les types de composants suivants :

- applets,
- application clientes,
- composants Enterprise JavaBeans (EJB),
- composants Web,
- composants adaptateurs de ressource

Les applets et applications clientes sont exécutées sur le poste du client tandis que les composants EJB, Web et adaptateurs de ressources fonctionnent sur le serveur.

A l'éxception des adaptateurs de ressources, les concepteurs et développeurs d'application développents les composants d'une application J2EE.

Les adaptateurs de ressources et logiciels associés sont en général vendu par les fournisseurs de systèmes d'information de l'entreprise et ensuite déployés sur les serveurs pour accéder aux données.

Tous les composants J2EE dépendent à l'exécution d'une entité système baptisée conteneur (*container*).

Les conteneurs fournissent aux composants des services de bases comme la gestion du cycle de vie, la sécurité, le déploiement et l'exécution en thread. Comme c'est le conteneur qui gère ces services, la plupart des paramètres de configuration de ces services peuvent être configurés lors du déploiement des composants en fonction de la plateforme d'accueil.

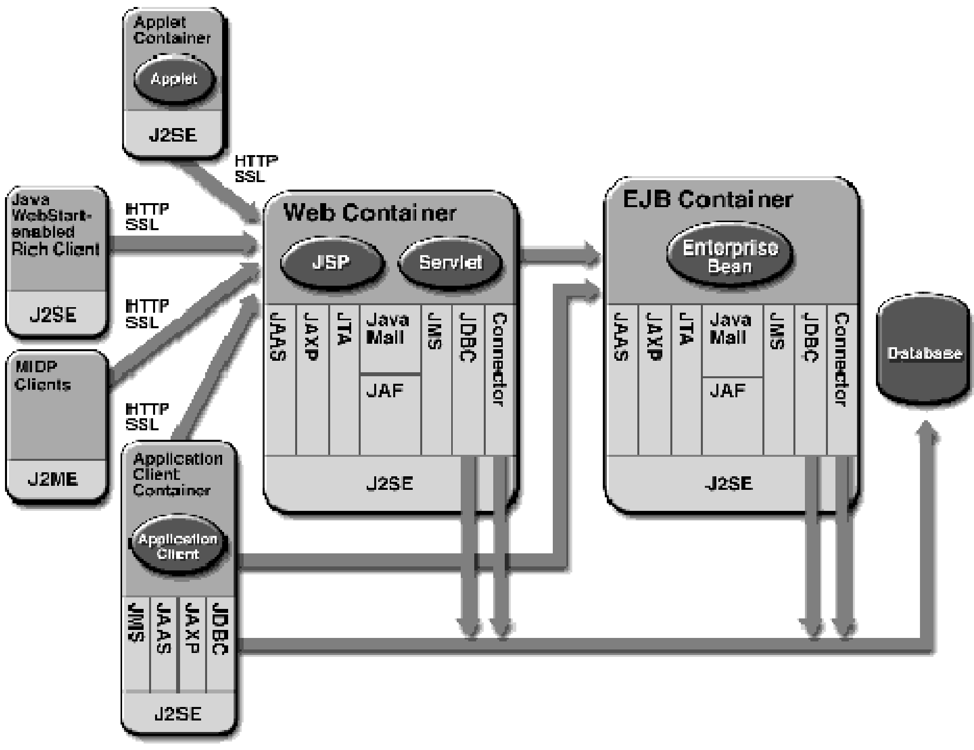


Figure 3.2 : Composants J2EE

1.2.1 Clients J2EE

La plateforme J2EE prévoit que plusieurs types de clients puissent accéder à une même application et interagir avec les composants côté serveur.

- **Applets**

Les Applets sont des clients Java qui s'exécutent habituellement dans un navigateur web et qui ont accès à toutes les possibilités du langage Java. Les applications J2EE peuvent utiliser des clients applets pour avoir des interfaces utilisateurs plus puissantes que celles connues en HTML. Les applets communiquent avec le serveur par HTTP.

- **Applications clientes**

Des *applications clientes* s'exécutent dans leur propre conteneur client. (le conteneur client est un jeu de librairies et d'API qui supportent le code client). Les applications clients ont des interfaces utilisateurs qui peuvent directement interagir avec le tier EJB en utilisant RMI-IIOP. Ces clients ont un accès complet. aux services de la plateforme J2EE comme les services de nommage JNDI, l'envoi de messages et JDBC. Le conteneur client gère l'accès à ces services et les communications RMI-IIOP.

- **Applications clientes Java Web Start**

Les *applications clients Java Web Start* sont des applications autonomes reposant sur les JFC et Swing et capables d'utiliser les services de la plateforme J2EE par l'intermédiaire de la technologie Java WebStart. Ces applications peuvent être installées par le web et communiquent avec le serveur en utilisant du XML encapsulé dans du HTTP(S).

- **Clients sans fil**

Les *clients sans fil* sont basés sur la technologie Mobile Information Device Profile (MIDP), en conjonction avec Connected Limited Device Configuration (CLDC), qui fournissent un environnement J2ME complet pour les dispositifs sans fil.

1.2.2 *Composants web*

Un composant web est une entité logicielle qui fournit une réponse à une requête.Les composants web génèrent habituellement l'interface utilisateur d'une application web. La plateforme J2EE définit deux types de composants web : les servlets et les JavaServer Pages (JSP). La section suivante donne un aperçu de ces composants qui sont détaillés ultérieurement.

- **Servlets**

Une servlet est un composant qui étend les fonctionnalités d'un serveur web de manière portable et efficace.

Un serveur web héberge des classes Java servlets qui sont exécutées à l'intérieur du container web. Le serveur web associe une ou plusieurs URLs à chaque servlet et lorsque ces URLs sont appelées via une requête HTTP de l'utilisateur la servlet est déclenchée.Quand la servlet reçoit une requête du client, elle génère une réponse, éventuellement en utilisant le logique métier contenu dans des EJBs ou en interrogeant directement une base de données. Elle retourne alors une réponse HTML ou XML au demandeur.

Un développeur de servlet utilise l'API servlet pour :

- Initialiser et finaliser la servlet
- Accéder à l'environnement de la servlet
- Recevoir ou rediriger les requêtes et envoyer les réponses
- Interagir avec d'autres serlvets ou composants
- Maintenir les informations de sessions du client
- Filtrer avant ou après traitement les requêtes et les réponses
- Implémenter la sécurité sur le tier web

JavaServer Pages

La technologie JavaServer Pages (JSP) fournit un moyen simple et extensible pour générer du contenu dynamique pour le client web.

Une page JSP est un document texte qui décrit comment traiter la requête d'un client et comment créer une réponse.

Une page JSP contient :

Des informations de formatage (modèle) du document web, habituellement en HTML ou XML. Les concepteurs web peuvent modifier cette partie de la page sans affecter les parties dynamiques. Cette approche permet de séparer la présentation du contenu dynamique.

Des éléments JSP et de script pour générer le contenu dynamique du document Web. La plupart des pages JSP utilisent aussi des JavaBeans et/ou des Entreprise JavaBeans pour réaliser les opérations complexes de l'application. Les JSP permettent en standard d'instancier des beans, de modifier ou lire leurs attributs et de télécharger des applets. La technologie JSP est extensible en utilisant des balises personnalisées qui peuvent être encapsuler dans des bibliothèques de balises personnaliées (*taglibs*)

Conteneur de composants web

Les composants webs sont hébergés dans des conteneurs de servlets, conteneurs de JSP et conteneurs web.

En sus des fonctionnalités normales d'un conteneur de composants, un conteneur de servlets (*servlets container*) fournit les services réseaux par les quels les requêtes et réponses sont émises.Il décode également les requêtes et formate les réponses dans le format approprié. Tous les conteneurs de servlets doivent supporter le protocole HTTP et peuvent aussi supporter le protocole HTTPS.

Un conteneur de JSP (*JSP container*) fournit les mêmes services qu'un conteneur de servlets. Ces conteneurs sont généralement appelés conteneurs web (*Web containers*).

1.2.3 Composants Enterprise JavaBeans

L'architecture Entreprise JavaBeans est une technologie côté serveur pour développer et déployer des composants contenant la logique métier d'une application d'entreprise.Les composants Entreprise JavaBeans, aussi appelé Entreprise Beans, sont scalables, transactionnel et supporte l'accès concurrent.

Il y a 3 types d'entreprise beans : les beans de sessions, d'entité et de messages.

2. Topologie(s) d'une application J2EE

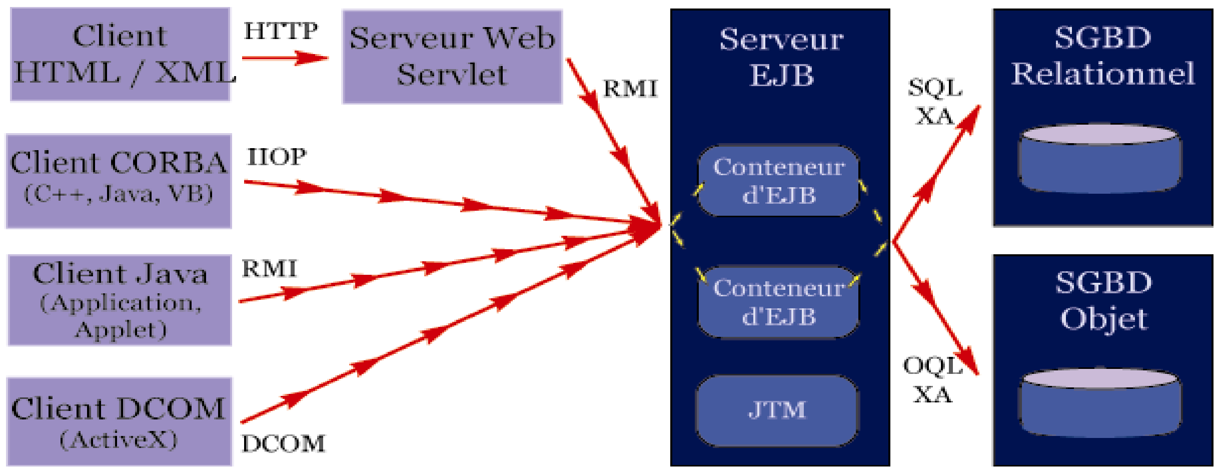


Figure 3.3 : Schéma général

3. Technologies J2EE

La plateforme J2EE comme la plateforme J2SE incluent un grand nombre de bibliothèques de code (API) prédéfinies pour les fonctions de base d'une application.

Les technologies mies en œuvre sont les suivantes :

- L'architecture J2EE Connector est l'infrastructure pour interagir avec une grande variété de systèmes d'information d'entreprise tels que des ERPs, des CRM, et autres progiciels.
- L'API JDBC est utilisée pour accéder à des données relationnelles à partir de programmes Java
- La Java Transaction API (JTA) est utilisée pour gérer et coordonner les transactions entre un ensemble hétérogène de systèmes d'information d'entreprise.

- L'API Java Naming and Directory Interface est utilisée pour accéder aux services de nommage et d'annaire de l'entreprise.
- L'API Java Message Service (JMS) est utilisée pour émettre et recevoir des messages via les systèmes de messagerie d'entreprise comme IBM MQ Series ou TIBCO Rendezvous. Dans l'architecture J2EE les Message Driven Beans fournissent une approche à base de composant pour encapsuler les fonctionnalités de messagerie.
- La JavaMail API est utilisée pour émettre et recevoir des emails.
- Java IDL est utilisé pour appeler des services CORBA
- L'API Java pour XML (JAXP) est utilisée pour l'intégration avec les systèmes et applications existants et pour implémenter les webservices dans la plateforme J2EE.

3.1 Remote Method Interface (RMI)

Interface pour les méthodes distantes.

Le package java correspondant est javax.rmi.

Le principe de RMI est de rendre possible l'accès aux méthodes d'un objet distant en établissant une connexion réseau (socket) entre le client et le serveur où sont stockés les objets.

Le client connaît les méthodes de l'objet distant au moyen des classes stubs qui contiennent les signatures des méthodes distantes.

Les classes javax.rmi sont utilisées pour chercher l'interface home d'un bean de session et l'activer sur le serveur distant.

3.2 JavaBeans

Il est possible d'utiliser la technologie JavaBeans (package Beans.*) entre une page JSP et un bean pour obtenir une meilleure séparation entre Modèle, Vue et Contrôleur (*Model View Controller – MVC*).

Le modèle MVC est un patron de conception (*design pattern*) qui consiste en 3 types d'objets :

- ✓ le Modèle procure la logique métier de l'application,
- ✓ la Vue est la présentation de l'application,
- ✓ le Contrôleur est un objet qui gère les interactions entre l'utilisateur et la Vue.

Un patron de conception décrit un problème récurrent et ses solutions, les solutions ne sont jamais exactement les mêmes pour chaque occurrence du problème mais le patron de conception donne unesolution générale au problème qu'ilsuffitd'adapteri.

3.3 Java Naming and Directory Interface (JNDI)

La technologie JNDI est utilisée dans J2EE pour localiser les objets sur un serveur et accéder aux objets externes à partir des composants J2EE.

Chaque conteneur stocke une référence aux objets qu'il peut créer et instancier ces objets à la demande des clients ou des applications qui fonctionnent sur le serveur.

Le conteneur met aussi à la disposition des composants un jeu de ressources JNDI initial, issu de la configuration du serveur et/ou des applications web (via les descripteurs de déploiement).

Un objet InitialContext est crée par le conteneur lorsqu'une web application est déployée.

Cet objet est accessible par les composants, en lecture seulement.

Dans Tomcat, ces ressources JDNI initiales sont accessibles dans l'espace de nommage java:comp/env.

Exemple d'accès à une ressource JDBC par JNDI :

// Obtain our environment naming context

Context initCtx = new InitialContext ();

```
Context envCtx = (Context) initCtx.lookup ("java:comp/env");

// Look up our data source

DataSource ds = (DataSource)

  envCtx.lookup ("jdbc/EmployeeDB");

// Allocate and use a connection from the pool

Connection conn = ds.getConnection ();

... use this connection to access the database ...

conn.close ();
```

L'API JNDI est définie dans le package javax.naming.

L'autre rôle de JNDI dans une application J2EE est la localisation des interfaces distantes des beans.

3.4 Java DataBase Connectivity (JDBC)

Cette API est développée par Sun en collaboration avec les grands éditeurs de SGBD.

Elle supporte plus de 50 drivers, permettant de se connecter aux bases de 40 éditeurs parmis lesquels Oracle, Informix, Postgres, Ingres, Sybase ...

JDBC supporte le SQL 2 ANSI pour l'écriture des requêtes plus des extensions spécifiques à chaque base de données.

Les tâches assurées par JDBC sont :

- Gestion des connexions et transactions
- Préparation de requêtes SQL
- Accès aisé aux résultats

JDBC est une architecture "bas niveau", qui est utilisée dans les applications J2EE pour assurer les fonctions de persistance des données.

L'utilisation de JDBC est faite soit par le conteneur d'application (*Container Managed Persistence*) soit directement dans le bean (*Bean Managed Persistence*).

L'utilisation typique de JDBC dans une classe java est la suivante :

- ✓ Chargement du driver
- ✓ Connection à la base (classe Connection)
- ✓ Expression d'une requête (classes Statement et PreparedStatement)
- ✓ Analyse du résultat (classe ResultSet)

Ex. de connection JDBC :

```
String url = “jdbc:postgresql://murphy/towns”

Connection c = DriverManager. getConnection( url);

Statement s = c. createStatement();

ResultSet r = s. executeQuery(“ SELECT * FROM VILLES”);

while (r. next())

{  int i = r. getInt(" Population");

  String s = r. getString(" Ville");

  System. out. println(“ Ville “+ s+” a “+ i+” hab.”);

  ......

}
```

3.5 Servlets

Les servlets sont des classes Java exécutées par le serveur web en réponse à une requête du client (en utilisant le protocole http).

Les servlets sont définies dans les packages suivants :

javax.servlet, contient les classes génériques (indépendantes du protocole) des servlets. La classe HTTPServlet utilise la classe *ServletException* de ce package pour indiquer un problème de servlet.

javax.servlet.http, contient la classe de serlvet concue pour le protocole HTTP (classe HttpServlet).

En général les servlets utilisent aussi le package java.io pour l'entrées/sorties système.

La classe HttpServlet utilise la classe IOException de ce package pour signaler les erreurs d'entrée-sortie.

3.6 Java Server Pages

La technologie JavaServer Page (JSP) permet de mettre des fragments de code java dans une page HTML statique.

Lorsque la page JSP est chargée par un navigateur web, le code java est exécuté sur le serveur. Celui-ci crée une servlet correspondante, qui est ensuite compilée et exécutée en tâche de fond.

La servlet retourne une page HTML ou un rapport en XML qui peut alors être transmis au client ou subir d'autres traitements.

Les JSP sont définies dans une classe d'implémentation appelée le package

Une page JSP est un document texte qui décrit comment créer un objet réponse (*response*) à partir d'un objet requête (*request*) pour un protocole donné.

Le traitement d'une page JSP peut entraîner la création et/ou l'utilisation d'autres objets.

Le protocole HTTP est le protocole utilisé par défaut.

3.7 Enterprise Java Beans

Le terme Enterprise Java Bean recouvre deux notions :

C'est le nom générique d'une architecture permettant la programmation répartie en Java

C'est le nom de composants exécutés sur un serveur et appelés par un client distant

Les EJBs n'ont en commun que le nom avec les Javabeans traditionnels, qui sont des composants côté clients utilisés pour obtenir une meilleurs séparation suivant le modèle MVC (model – view – controller).

Les Entreprise Java Beans ont pour but de rendre les applications faciles à développer, à déployer et à administrer.

Les EJBs sont indépendants de la plateforme d'exécution, étant écrits en Java : le déploiement d'un EJB se fait sans recompilation ni modification du code source.

Les spécifications EJB définissent une architecture pour la construction d'applications Java dont la partie serveur est construite à partir de composants Entreprise Beans.

Leurs caractéristiques principales sont les suivantes :

Les composants EB sont « écrits une fois, exécutable partout » (*write once, run anywhere*)

Ces composants sont des composants côté serveurs (analogues aux objets métiers de CORBA)

Les EJBs ne sont pas les seuls composants exploités dans une application J2EE mais ils en sont la partie centrale.

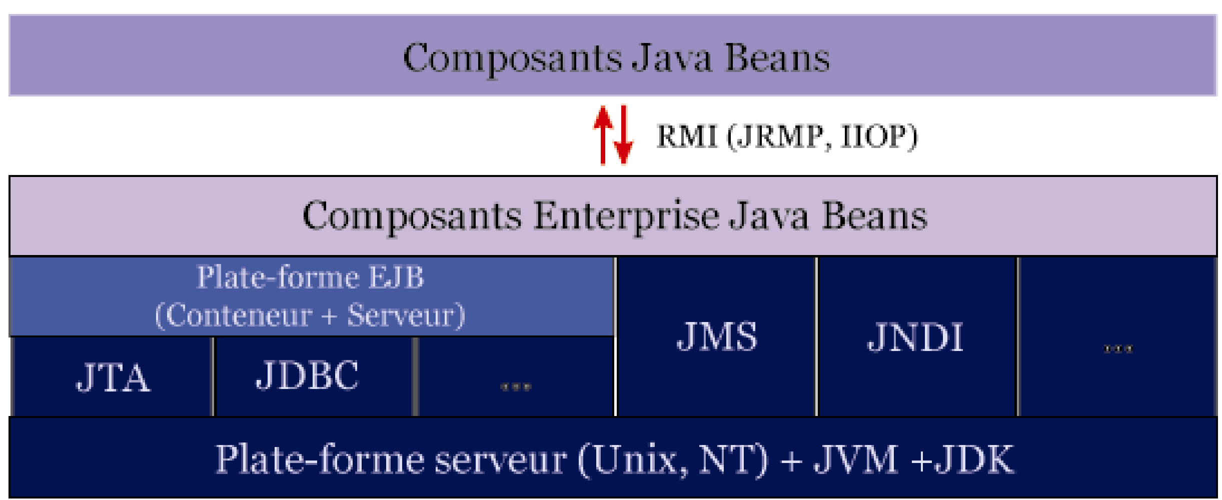


Figure 3.4 : composants EJB

Chapitre 4

Les entreprises java beans

1. Introduction

Les EJB (Entreprise Java Bean) sont un des éléments très important de la plate-forme Java EE pour le développement d'applications distribuées.

La plate-forme Java EE propose de mettre en oeuvre les couches métiers et persistance avec les EJB. Particulièrement intéressants dans des environnements fortement distribués, leur mise en œuvre est assez lourde jusqu'à la version 3 sans l'utilisation d'outils tels que certains IDE ou XDoclet.

La version 3 des EJB vise donc à simplifier le développement et la mise en œuvre des EJB qui sont fréquemment jugés trop complexes et trop lourds à mettre en œuvre.

Cette nouvelle version majeure des EJB propose une simplification de leur développement tout en conservant une compatibilité avec la précédente version des EJB. Elle apporte de très nombreuses fonctionnalités dans le but de simplifier la mise en œuvre des EJB.

Cette simplification est rendue possible notamment par :

L'utilisation des annotations

La mise en oeuvre de valeurs par défaut qui répond à la plupart des besoins (configuration par exception)

Le descripteur de déploiement est facultatif

L'utilisation de POJO et de JPA pour les beans de type entity

L'injection de dépendances côté serveur mais aussi côté client (l'interface Home qui gérait le cycle de vie est abandonnée) qui remplace l'utilisation directe de JNDI Tous ces éléments délèguent une partie du travail du développeur au conteneur d'EJB.

2. L'historique des EJB

EJB 1.1 publié en décembre 1999, intégré dans J2EE 1.2 :

- Session beans (stateless/stateful)
- Entity Beans (CMP / BMP)

- Interface Remote uniquement

EJB 2.0 publié en septembre 2001, intégré à J2EE 1.3 :

- Message-Driven Beans
- Entity 2.x reposant sur EJB QL

Interface Local pour améliorer les performances des appels dans la même JVM

EJB 2.1 publié en novembre 2003, intégré à J2EE 1.4 :

EJB Timer Service

EJB Web Service Endpoints via JAX-RPC

Amélioration du langage EJB QL

EJB 3.0, intégré à Java EE 5 :

- ✓ Utilisation de POJO et POJI, plus d'interface Home
- ✓ Utilisation des annotations, le descripteur de déploiement est optionnel
- ✓ Utilisation de JPA pour les beans de type entity

EJB 3.1, intégré à Java EE 6

3. Les nouveaux concepts et fonctionnalités utilisés

Dans les versions antérieures à la version 3.0 des EJB, le développeur était contraint de créer de nombreuses entités pour respecter l'API EJB (par exemple, l'implémentation d'interfaces engendrant la création de plusieurs méthodes ou le descripteur de déploiement), ce qui rendait leur écriture relativement lourde même avec l'assistance de certains IDE ou outils (XDoclet notamment). Dans la version 3.0, ceci est remplacé par l'utilisation d'annotations.

La mise en oeuvre de l'interface EJBHome n'est plus requise : un EJB de type session est maintenant une simple classe, qui peut implémenter une interface métier.

La seule annotation obligatoire dans un EJB est celle qui précise le type d'EJB (@javax.ejb.Stateless, @javax.ejb.Stateful ou @javax.ejb.MessageDriven).

Les annotations possèdent des valeurs par défaut qui répondent à une majorité de cas typique d'utilisation. L'utilisation de ces annotations n'est alors requise que si les valeurs par défaut ne répondent pas au besoin. Ceci permet de réduire la quantité de code à écrire.

L'utilisation des annotations et de valeurs par défaut pour la plupart de ces dernières rend optionnelle la nécessité de créer un descripteur de déploiement sauf pour des besoins très particuliers.

Le conteneur obtient des informations sur la façon de mettre en oeuvre un EJB par trois moyens :

- Des valeurs par défaut pour la plupart des annotations ce qui évite d'avoir à les utiliser explicitement dans le code
- Les annotations utilisées dans le code
- Le descripteur de déploiement

L'ordre d'utilisation par le conteneur est : le descripteur de déploiement, les annotations, les valeurs par défaut.

L'utilisation des annotations est plus simple à mettre en oeuvre mais le descripteur de déploiement permet de centraliser les informations.

La nouvelle API Java Persistence remplace la persistance assurée par le conteneur : cette API assure la persistance des données grâce à un mapping O/R reposant sur des POJO.

Le conteneur a la possibilité d'injecter des dépendances d'objets gérés par le conteneur.

Les intercepteurs permettent d'offrir certaines fonctionnalités proches de certaines proposées par l'AOP : ceci permet de définir des traitements lors de l'invocation de méthodes des EJB ou d'invoquer certaines méthodes liées au cycle de vie de l'EJB.

3.1. L'utilisation de POJO et POJI

Les classes et les interfaces des EJB 3.0 sont de simples POJO ou POJI : ceci simplifie le développement des EJB. Par exemple, l'interface Home n'est plus à déclarer.

Il est toujours possible d'implémenter les interfaces SessionBean, EntityBean et MessageDrivenBean mais le plus simple est d'utiliser les annotations définies : @Stateless, @Stateful, @Entity ou @MessageDriven

Exemple :

```
1.@Stateless
2.public class HelloWorldBean {
3.  public String saluer(String nom)
4.  {
5.    return "Bonjour "+nom;
6.  }
7.}
```

Il est possible de définir une interface métier pour l'EJB ou de laisser générer cette interface lors du déploiement.

Dans le premier cas, il n'est plus nécessaire qu'elle implémente l'interface EJBObject ou EJBLocalObject mais simplement d'utiliser les annotations définies : @Remote ou @Local.

Exemple :

```
1.@Remote
2.@Stateless
3.public class HelloWorldBean {
4.  public String saluer(String nom)
5.  {
6.    return "Bonjour "+nom;
7.}
8.}
```

Dans le second cas, ces annotations doivent être utilisées dans la classe d'implémentation pour permettre de déterminer l'interface générée.

Il est possible de définir une interface locale et/ou distante pour un même EJB.

Il n'est pas recommandé de laisser les interfaces être générées par le conteneur pour plusieurs raisons :

- les interfaces générées exposent par défaut toutes les méthodes de l'EJB
- l'interface est utilisée par le client pour invoquer l'EJB
- le nom des interfaces générées utilise le nom de l'implémentation de l'EJB

3.2. L'utilisation des annotations

La spécification 3.0 des EJB fait un usage intensif des annotations. Celles-ci sont issues de la JSR 175 et intégrées dans Java SE 5.0 qui constitue la base de Java EE 5.

Les annotations sont des attributs ou méta-données à l'image de celles proposées par XDoclet.

Avec les EJB 3.0, les annotations sont utilisées pour générer des entités et remplacer tout ou partie du descripteur de déploiement.

De nombreuses annotations permettent de simplifier le développement des EJB.

La nature de l'EJB est précisée par une des annotations @Stateless, @Stateful, @Entity et @MessageDriven selon le type d'EJB à définir.

Le type d'accès est précisé par deux annotations

- @Remote : permet un accès à l'EJB depuis un client hors de la JVM
- @Local : permet un accès à l'EJB depuis un client dans la même JVM que celle de l'EJB

Par défaut, l'interface d'appel est locale si aucune annotation n'est indiquée.

Dans le cas d'un accès distant, il est inutile que chaque méthode précise qu'elle peut lever une exception de type RemoteException mais elles peuvent déclarer la levée d'exceptions métier.

Jusqu'à la version 2.1 des EJB, il était obligatoire d'implémenter plusieurs méthodes relatives à la gestion du cycle de vie de l'EJB notamment ejbActivate, ejbLoad, ejbPassivate, ejbRemove, ... pour chaque EJB même si ces méthodes ne contenaient aucun traitement.

Avec les EJB 3.0, l'implémentation de ces méthodes est remplacée par l'utilisation facultative selon les besoins d'annotations sur les méthodes concernées. La signature de ces méthodes doit être de la forme public void nomMethode()

Par exemple, pour que le conteneur exécute automatiquement une méthode avant de retirer l'instance du bean, il faut annoter la méthode avec l'annotation @Remove.

Plusieurs annotations permettent ainsi de définir des méthodes qui interviendront dans le cycle de vie de l'EJB.

Annotation	**Rôle**
@PostConstruct	est invoquée après que l'instance soit créée et que les dépendances soient injectées
@PostActivate	est invoquée après que l'instance de l'EJB ne soit déssérialisée du disque. C'est l'équivalent de la méthode ejbActivate() des EJB 2.x
@Remove	est invoquée avant que l'EJB ne soit retiré du conteneur
@PreDestroy	est invoquée avant que l'instance de l'EJB ne soit supprimée
@PrePassivate	est invoquée avant de l'instance de l'EJB ne soit sérialisée sur disque. C'est l'équivalent de la méthode ejbPassivate () des EJB 2.x

L'utilisation facultative de ces annotations remplace la définition obligatoire des méthodes de gestion du cycle de vie utilisées jusqu'à la version 2.1 des EJB.

Le descripteur de déploiement n'est plus obligatoire puisqu'il peut être remplacé par l'utilisation d'annotations dédiées directement dans les classes des EJB.

Chaque attribut de déploiement possède une valeur par défaut qu'il ne faut définir que si cette valeur ne répond pas au besoin.

Plusieurs annotations sont définies par les spécifications des EJB pour permettre de définir le type de bean, le type de l'interface, des références vers des ressources qui seront injectées, la gestion des transactions, la gestion de la sécurité, ...

Chaque vendeur peut définir en plus ces propres annotations dans l'implémentation de son serveur d'application. Leur utilisation n'est cependant pas recommandée car ils rendent l'application dépendante du serveur d'applications utilisé.

L'utilisation des annotations va simplifier le développement des EJB mais la gestion de la configuration pourra devenir plus complexe puisqu'elle n'est plus centralisée.

4. EJB 2.x vs EJB 3.0

Le développement d'EJB n'a jamais été facile et est même devenu plus complexe au fur et à mesure des nouvelles spécifications.

Avant la version 3.0 des EJB, les EJB sont relativement complexes et lourds à mettre en oeuvre :

- création de plusieurs interfaces et classes (deux interfaces et une classe au minimum)
- implémentation de méthodes callback généralement inutiles
- l'interface de l'EJB doit hériter de EJBObject ou de EJBLocalObject
- chaque méthode de l'EJB doit déclarer pouvoir lever l'exception RemoteException
- le descripteur de déploiement des EJB est complexe
- les EJB Entité de type CMP possèdent plusieurs limitations : complexe à développer et à maintenir, de nombreux problèmes de performance, le langage EJBQL est limité
- le support de la POO pour les EJB est très limité vis-à-vis de l'héritage
- les EJB doivent être testés dans un conteneur ce qui les rend difficile à déboguer
- l'appel d'un EJB par un client nécessite obligatoirement une utilisation de JNDI

La version 3.0 des spécifications des EJB apporte une solution de simplification à tous les points précédemment cités.

- Il n'est plus nécessaire de déclarer d'interfaces (pour des raisons de bonne pratique, la déclaration d'une interface métier contenant les méthodes proposées est cependant fortement recommandée)
- Le descripteur de déploiement est optionnel sauf dans des cas particuliers
- L'utilisation de POJO et POJI
- L'injection de dépendances rend très facile l'obtention d'une instance d'une ressource gérée par le conteneur

Les principales différences entre les EJB 2.x et EJB 3.0 sont donc :

Les EJB sont les descripteurs de déploiement ne sont plus obligatoires grâce à l'utilisation d'annotations et de valeurs par défaut de simples POJO annotés :

Ils n'ont plus besoin d'implémenter une interface de l'API EJB. De fait, il n'est plus nécessaire de définir des méthodes liées au cycle de vie de l'EJB.

Si ces méthodes sont nécessaires, il suffit d'utiliser des annotations dédiées sur une méthode.

Le type de l'interface de l'EJB est précisé avec l'annotation @Local ou @Remote

L'interface métier est une simple POJI

Les EJB de type Entity CMP et BMP sont remplacés par l'utilisation du modèle de persistance reposant sur l'API JPA

5. Les types des ejb

5.1 Les EJB de type Session

Les EJB session sont généralement utilisés comme façade pour proposer des fonctionnalités qui peuvent faire appel à d'autres composants ou entités tels que des EJB session, des EJB Entity, des POJO, ...

La version 3.0 des EJB rend inutile l'implémentation d'une interface spécifique à l'API EJB. Mais même si cela n'est pas obligatoire, il est fortement recommandé (dans la mesure du possible) de définir une interface dédiée à l'EJB qui va notamment préciser son mode d'accès et les méthodes utilisables.

Cette interface est alors une simple POJI.

5.1.1 L'interface distante et/ou locale

Un EJB peut être invoqué :

- en local : le client appelant est exécuté dans la même JVM que celle de l'EJB. Ce type d'appel est le plus performant puisqu'il ne nécessite pas d'échanges réseaux et donc pas de mécanisme pour gérer ces échanges
- à distance : le client appelant est exécuté dans une autre JVM que celle de l'EJB

L'interface distante définit les méthodes qui peuvent être appelées par un client en dehors de la JVM du conteneur. L'interface ou le bean doit être marquée avec l'annotation @Remote implémentée dans la classe javax.ejb.Remote

L'interface locale définit les méthodes qui peuvent être appelées par un autre EJB s'exécutant dans la même JVM que le conteneur. Les performances sont ainsi accrues car les mécanismes de protocoles d'appels distants ne sont pas utilisés (sérialisation/désérialisation, RMI, ...).

L'utilisation de l'interface Local pour des appels à l'EJB dans un même JVM est fortement recommandé par cela améliore les performances de façon dramatique. L'interface Remote met en oeuvre des mécanismes de communication utilisant la sérialisation, ce qui dégrade les performances notamment de façon inutile si l'appel à l'EJB se fait dans une même JVM.

Un client ne dialogue jamais en direct avec une instance de l'EJB : le client utilise toujours l'interface pour accéder au bean grâce à un proxy généré par le conteneur. Même un client local utilise un proxy particulier dépourvu des accès réseau. Ce proxy permet au conteneur d'assurer certaines fonctionnalités comme la sécurité et les transactions.

5.1.2. Les beans de type Stateless

Les beans de type stateless sont les plus simples et les plus véloces car le conteneur gère un pool d'instances qui sont utilisées au besoin, ce qui évite des opérations d'instanciation et de destruction à chaque utilisation. Ceci permet une meilleure montée en charge de l'application.

L'annotation @javax.ejb.Stateless permet de préciser qu'un EJB session est de type stateless. Elle s'utilise sur une classe qui encapsule un EJB et possède plusieurs attributs :

Attribut	Rôle
String name	Nom de l'EJB (optionnel)
String mappedName	Nom sous lequel l'EJB sera mappé. La valeur par défaut est le nom non qualifié de la classe (optionnel)
String description	Description de l'EJB (optionnel)

Il faut définir l'interface de l'EJB avec l'annotation précisant le mode d'accès

L'annotation @javax.ejb.Remote permet de préciser que l'EJB pourra être accédé par des clients distants. Elle s'utilise sur une classe qui encapsule un EJB ou l'interface qui décrit les fonctionnalités de l'EJB utilisables à distance. Cette annotation ne peut être utilisée que pour des EJB sessions.

Elle possède un seul attribut :

Attribut	Rôle
Class[] value	Préciser la liste des interfaces distantes de l'EJB. Son utilisation est obligatoire si la classe de l'EJB implémente plusieurs interfaces différentes java.io.Serializable, java.io.Externalizable, ou une des interfaces du package javax.ejb (optionnel)

Exemple :

```
1.import javax.ejb.Remote;
2.
3.@Remote
4.public interface CalculRemote {
5.  public long additionner(int valeur1, int valeur2);
6.}
```

Cette interface est marquée avec l'annotation @Remote pour permettre un appel distant et définit la méthode additionner.

Remarque : l'utilisation de l'annotation rend inutile l'utilisation de la clause throws RemoteException des versions antérieures des EJB.

L'annotation @javax.ejb.Local permet de préciser que l'EJB pourra être accédé par des clients locaux de la JVM. Elle s'utilise sur une classe qui encapsule un EJB ou l'interface qui décrit les fonctionnalités de l'EJB utilisables en local dans la JVM. Cette annotation ne peut être utilisée que pour des EJB sessions.

Elle possède un attribut :

Attribut	Rôle
Class[] value	Préciser la liste des interfaces distantes de l'EJB. Son utilisation est obligatoire si la classe de l'EJB implémente plusieurs interfaces différentes java.io.Serializable, java.io.Externalizable, ou une des interfaces du package javax.ejb (optionnel)

Cette interface est marquée avec l'annotation @Local pour permettre un appel local et définit la méthode additionner.

Il faut ensuite définir la classe de l'EJB qui va contenir les traitements métier.

Exemple :

```
1.import javax.ejb.*;
2.
3.@Stateless
4.public class CalculBean implements CalculRemote, CalculLocal {
5.  public long additionner(int valeur1, int valeur2) {
6.    return valeur1 + valeur2;
7.  }
8.}
```

Cette classe est marquée avec l'annotation @Stateless et implémente les interfaces distante et locale précédemment définies.

Il est préférable lorsque cela est possible d'utiliser l'interface Local car elle est beaucoup plus performante. L'interface Remote est à utiliser lorsque le client n'est pas dans la même JVM.

Les annotations @Local et @Remote peuvent être utilisées directement sur l'EJB mais il est préférable de définir une interface par mode d'accès et d'utiliser l'annotation adéquate sur chacune des interfaces.

La classe de l'EJB ne doit plus implémenter l'interface javax.ejb.SessionBean qui était obligatoire avec les EJB 2.x. Maintenant, les EJB session de type stateless peuvent utiliser les callbacks d'évènements marqués avec les annotations suivantes :

- @PostConstruct
- @PreDestroy

5.1.3. Les beans de type stateful

Les beans de type stateful sont capables de conserver leur état durant toute leur utilisation par le client. Cet état n'est cependant pas persistant : les données sont perdues à la fin de son utilisation ou à l'arrêt du serveur. Un exemple type d'utilisation de ce type de bean est l'implémentation d'un caddie pour un site de vente en ligne.

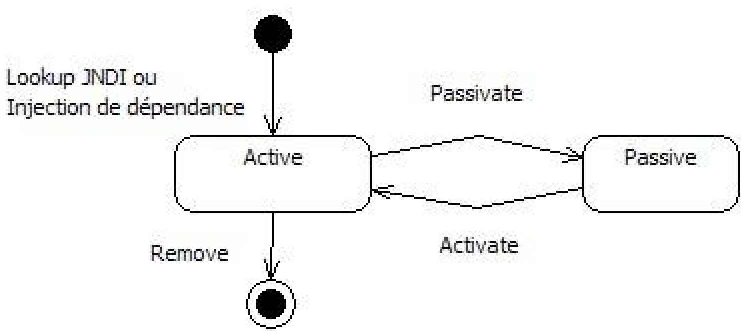
L'annotation @javax.ejb.Stateful permet de préciser qu'un EJB session est de type stateful. Elle s'utilise sur une classe qui encapsule un EJB.

Elle possède plusieurs attributs :

Attribut	Rôle
String name	Nom de l'EJB (optionnel)
String mappedName	Nom sous lequel l'EJB sera mappé. La valeur par défaut est le nom non qualifié de la classe (optionnel)
String description	Description de l'EJB (optionnel)

Le conteneur EJB a la possibilité de serialiser/dessérialiser des EJB de type Stateful notamment dans le cas ou la JVM du conteneur commence à manquer de mémoire. Dans ce

cas, le conteneur peut serialiser des EJB (passivate) qui ne sont pas en cours d'utilisation sur disque. Dès qu'un de ces EJB sera sollicité, le conteneur va le desserialiser (activate) du disque pour le remettre en mémoire et pouvoir l'utiliser.



Ils n'ont plus à implémenter l'interface javax.ejb.SessionBean comme c'était le cas dans les versions antérieures aux EJB 3.0. Maintenant, les EJB session de type stateful peuvent utiliser les callbacks d'évènements marqués avec les annotations suivantes :

- @PostConstruct
- @PostActivate
- @PreDestroy
- @PrePassivate
- @Remove

5.2. Les EJB de type Entity

Dans les versions antérieures des EJB, les EJB de type Entity avaient la charge de la persistance des données. Les EJB de type Entity CMP (Container Managed Persistence) doivent simplement requérir un fichier de description.

Les EJB 3.0 proposent d'utiliser l'API Java Persistence pour assurer la persistance des données dans les EJB : ils utilisent un modèle de persistance léger standard en remplacement des entity beans de type CMP.

JPA repose sur des beans entity qui sont de simples POJO enrichis d'annotations qui permettent de mettre en oeuvre les concepts de POO tels que l'héritage ou le polymorphisme.

Jusqu'à la version 3.0 des EJB, les Entity beans sont des composants qui dépendent pleinement du conteneur d'EJB du serveur d'applications dans lequel ils s'exécutent. L'utilisation de POJO avec l'API Java Persistence permet de rendre les beans entity indépendants du conteneur. Ceci possède plusieurs avantages dont celui de pouvoir facilement tester les beans puisqu'ils ne requièrent plus de conteneur pour leur exécution.

Avec la version 3.0 des EJB, les beans entity sont donc des POJO qui n'ont donc pas besoin d'implémenter une interface spécifique aux EJB et qui doivent posséder un constructeur sans argument et implémenter l'interface Serializable.

Les attributs persistants sont déclarés via des annotations soit au niveau de l'attribut soit au niveau de son getter/setter. De ce fait, ils peuvent être utilisés directement comme objets du domaine ; il n'y a plus l'obligation de définir un DTO.

5.2.1. La création d'un bean Entity

Les beans de type Entity sont dans la version 3.0 des spécifications de simple POJO utilisant les annotations de l'API Java Persistence (JPA) pour définir le mapping.

Les informations de mapping entre une table et un objet peuvent être définies grâce aux annotations mais aussi via un fichier de mapping qui permet d'externaliser ces informations du POJO. Il est possible de mixer les deux (annotations et fichiers de mapping) mais les données incluses dans le fichier sont prioritaires par rapport aux annotations.

Le bean entity doit être annoté avec l'annotation @Entity implémentée dans la classe javax.persistence.Entity.

L'annotation @Table implémentée dans la classe javax.persistence.Table permet de préciser le nom de la table vers lequel le bean sera mappé. L'utilisation de cette annotation est facultative si le nom de la table correspond au nom de la classe.

Pour mapper un champ de la table avec une propriété du bean, il faut utiliser l'annotation @Column implémentée dans la classe javax.persistence.Column sur le getter de la propriété. L'utilisation de cette annotation est facultative si le nom du champ correspond au nom de la propriété.

Le champ correspondant à la clé primaire de la table doit être annoté avec l'annotation @Id implémenté dans la classe javax.persistence.Id. L'utilisation de cette annotation est obligatoire car un identifiant unique est obligatoire pour chaque occurrence et l'API n'a aucun moyen de déterminer le champ qui encapsule cette information.

Il peut être pratique pour un bean de type entity d'implémenter l'interface Serializable : le bean pourra être utilisé dans les paramètres et la valeur de retour des méthodes métiers d'un EJB. Le bean peut ainsi être utilisé pour la persistance et le transfert de données.

Exemple :

```
01.package com.jmd.test.domaine.entity;
02.
03.import java.io.Serializable;
04.import javax.persistence.Column;
05.import javax.persistence.Entity;
06.import javax.persistence.Id;
07.import javax.persistence.NamedQueries;
08.import javax.persistence.NamedQuery;
09.import javax.persistence.Table;
10.
11.@Entity
12.@Table(name = "PERSONNE")
13.@NamedQueries({@NamedQuery(name = "Personne.findById",
14.  query = "SELECT p FROM Personne p WHERE p.id = :id"),
15.             @NamedQuery(name = "Personne.findByNom",
16.  query = "SELECT p FROM Personne p WHERE p.nom = :nom"),
17.             @NamedQuery(name = "Personne.findByPrenom",
18. query = "SELECT p FROM Personne p WHERE p.prenom = 
:prenom")})
19.public class Personne implements Serializable {
20.    private static final long serialVersionUID = 1L;
21.    @Id
22.    @Column(name = "ID", nullable = false)
```

```
23.  private Integer id;
24.  @Column(name = "NOM")
25.  private String nom;
26.  @Column(name = "PRENOM")
27.  private String prenom;
28.
29.  public Personne() {
30.  }
31.
32.  public Personne(Integer id) {
33.      this.id = id;
34.  }
35.
36.  public Integer getId() {
37.      return id;
38.  }
39.
40.  public void setId(Integer id) {
41.      this.id = id;
42.  }
43.
44.  public String getNom() {
45.      return nom;
46.  }
47.
48.  public void setNom(String nom) {
49.      this.nom = nom;
50.  }
51.
52.  public String getPrenom() {
53.      return prenom;
54.  }
55.
```

```
56.    public void setPrenom(String prenom) {
57.        this.prenom = prenom;
58.    }
59.
60.    @Override
61.    public String toString() {
62.        return "com.jmd.test.domaine.entity.Personne[id=" + id + "]";
63.    }
64.
65.}
```

5.2.2. La persistance des entités

La version 3.0 propose une refonte complète des EJB entités afin de simplifier leur développement. Cette simplification est assurée en grande partie par la mise en oeuvre de JPA qui permet :

la standardisation du mapping O/R

L'utilisation de POJO annotés avec support de l'héritage et du polymorphisme

La possibilité d'utiliser les EJB entités en dehors du conteneur d'EJB qui permet notamment la mise en oeuvre de tests unitaires automatisés.

La persistance d'objets avec JPA repose sur plusieurs fonctionnalités :

Un ensemble d'entités annotées qui représente le modèle objet du domaine

Une API contenue dans le package javax.persistence

Un cycle de vie pour les entités

La classe EntityManager est responsable de la gestion des opérations sur une entité notamment grâce à plusieurs méthodes :

- ✓ persist ()
- ✓ remove ()
- ✓ merge ()
- ✓ flush ()
- ✓ find ()
- ✓ refresh ()

5.3. Les EJB de type MessageDriven

Les EJB de type MessageDriven permettent de réaliser des traitements asynchrones exécutés à la réception d'un message dans une queue JMS.

Ils ne proposent pas d'interface locale ou distante et ne peuvent pas être utilisés via un service web. Pour connecter le bean à une queue JMS, il faut que le bean implémente l'interface javax.jms.MessageListener.

Cette interface définit la méthode onMessage(Message).

5.3.1 L'annotation @ javax.ejb.MessageDriven

L'annotation @ javax.ejb.MessageDriven permet de préciser qu'un EJB est de type MessageDriven. Elle s'utilise sur une classe qui encapsule un EJB.

L'annotation @MessageDriven possède plusieurs attributs optionnels :

Attribut	Rôle
ActivationConfigProperty[] activationConfig	Préciser les informations de configuration (type de endpoint, destination (queue ou topic), mode d'aquittement des messages, ...) sous la forme d'un tableau d'annotations de type @javac.ejb.ActivationConfigProperty (optionnel)
String description	Description de l'EJB (optionnel)
String mappedName	Nom sous lequel l'EJB sera mappé. Peut aussi être utilisé pour désigner le nom JNDI de la destination utilisée (optionnel)

Class messageListenerInterface	Préciser l'interface de type message Listener. Il faut utiliser cet attribut si l'EJB n'implémente pas d'interface ou implémente plusieurs interfaces différentes de java.io.Serializable, java.io.Externalizable, ou une ou plusieurs interfaces du package javax.ejb. La valeur par défaut est Object.class (optionnel)
String name	Nom de l'EJB. La valeur par défaut c'est le nom non qualifié de la classe (optionnel)

5.3.2. L'annotation @javax.ejb.ActivationConfigProperty

Les paramètres nécessaires à la configuration de l'EJB notamment le type et la destination sur laquelle le bean doit écouter doivent être précisés grâce à l'attribut activationConfig. Cet attribut est un tableau d'objets de type ActivationConfigProperty.

L'annotation @javax.ejb.ActivationConfigProperty permet de préciser le nom et la valeur d'une propriété de la configuration des EJB de type MessageDriven. Elle s'utilise dans la propriété activationConfig d'une annotation de type javax.ejb.MessageDriven.

Elle possède plusieurs attributs :

Attribut	Rôle
String propertyName	Préciser le nom de la propriété (obligatoire)
String propertyValue	Préciser la valeur de la propriété (obligatoire)

Exemple :

```
1.@MessageDriven(mappedName = "jms/MonEJBQueue", activationConfig = {
2.  @ActivationConfigProperty(propertyName="acknowledgeMode",          propertyValue="Auto-
acknowledge"),
3.  @ActivationConfigProperty(propertyName="destinationType",
propertyValue="javax.jms.Queue")
4.})
```

Chapitre 5

Modélisation d'une application e-commerce

1. Introduction

Après avoir présenté les concepts fondamentaux de développement des composants EJB. Cette partie traite d'une étude détaillée du développement et d'implémentation d'une application de commerce électronique. (E-commerce). Depuis la spécification jusqu'a la conception détaillée.

Avant de présenter un chapitre consacré à la partie implémentation .ce premier chapitre introduit de manière générale les applications e-commerce leur place actuelle dans les applications web.

Aussi .nous présentons notre démarche suivie dans cette étude ainsi que les différentes technologies choisies et mises en œuvre. Ensuite nous détaillons notre modélisation qui s'est basée sur UML en explicitant les différente étapes de modélisation chacune finalisée par un modèle.

2. Les applications e-commerces

2.1 Définition d'un site e-commerce

On appelle commerce électronique l'utilisation des réseaux informatique notamment Internet pour la réalisation de transaction commerciales la plupart du temps il s'agit de la vente de produits sur Internet. Mais le terme e-commerce englobe aussi les mécanismes d'achat par Internet un consommateur peut alors visiter un site e-commerce et choisir des produits.

Le e-commerce ne se limite pas à la vente en ligne mais il englobe également

- ✓ la mise à disposition d'un catalogue électronique.
- ✓ la réalisation de devis en ligne.
- ✓ le paiement en ligne.
- ✓ le suivi de livraison.
- ✓ la gestion en temps réel de la disposition des produits.
- ✓ le service après vente.

2.2 Objectifs d'un site e-commerce

L'objectif principal d'un site e-commerce est de présenter les différents produits d'une entreprise dans un site web et d'offrir la faire des commandes et de consulter ces produits.

Un site e-commerce a aussi pour objectif de permettre aux clients de consulter toutes les informations concernant les produits qui sont stockés dans une base de données.

Afin de réaliser ces objectifs.la conception de doit être modulaire extensible et bien documentée .ce qui justifie l'utilisation des diagrammes UML.des design patterns et l'intégration des Framework existantes.

3. Description de notre application

Notre application consiste à réalisé un site e-commerce d'une entreprise virtuelle doté des fonctions de base existantes dans les sites e-commerces à savoir la recherche des produits. La gestion du panier ainsi que la gestion des commandes.

Pour cela nous avons choisi comme type d'application la vente de matériels informatique (PC portable .Imprimante. Lecteur.CD/DVD. Ecrans. Disques durs).quand un internaute accède à notre site .une page d'accueil qui contient les nouveaux produit de l'entreprise s'affiche .Ainsi .un client peut effectuer des recherche rapides ou pas catégorie. À l'affichage du résultat de la recherche .les produit peuvent être classés selon différents critères le client à la possibilité de voir la fiche détaillée d'un produit qui peut être ajouté dans un panier.

Le client peut voir l'état de son panier en temps réel .il a aussi la passibilité de modifier la quantité des produit dans le panier et voir le montant total afin de lui donner une idée sur le paiement .le client peut aussi vider son panier.

Quand un client termine la manipulation de son panier .il peut passer à l'étape de commande cette étape exige l'inscription du client s'il n'est pas inscrit. Sinon. Il aura la possibilité de modifier son compte.

Après authentification. Le client remplit un bon de commande pour qu'il passe à l'étape de paiement.

4. Processus de développement de l'application

Un processus définit une séquence d'étapes en partie ordonnées qui concourent à l'obtention d'un système existant.

L'objectif d'un processus de développement est de produire des logiciels de qualité qui répondent aux besoins de leurs utilisateurs dans des temps et des coûts prévisibles.

L'ors de la réalisation de notre application .on s'est basé sur un processus de développement simple conduit par les cas d'utilisation et centré sure l'architecture le processus que nous allons présenter et appliquer tout au long de ce projet est fondé sur l'utilisation d'un sous ensemble de diagrammes UML nécessaire et suffisant.

5. La démarche de développement

5.1 Les étapes de développement d'un logiciel

Avant d'entamer l'étude de notre application .il est nécessaire de présenter les différentes étapes de développement d'un logiciel.

1. **Capture des besoins:** Cette étape consiste à spécifier les fonctionnalités du système en fonction de l'information recueillies et spécifiées dans le cahier de charge .elle permet de recenser les cas d'utilisation (use cases en UML) cette phase est importante pour le démarrage des activités d'analyse et conception du système.
2. **Analyse des besoins**: Elle donne une description détaillée des fonctionnalisées du système à l'aide de diagrammes de séquences et met en évidence les objets et leurs liens Au niveau de cette phase l'accent n'est pas mis sur la façon dont le système fonctionne.
3. **Conception générique :** Cette étape permet de formaliser le comment du système c'est – à-dire de spécifier la façon dont sera conçu le système et comment il fonctionnera
4. **Conception détaillée:** Cette étape consiste à définir précisément chaque fonctionnalité du système. Elle précède la phase de codage .A ce niveau .toutes les questions relatives aux

détails de la solution doivent être modélisées. Ainsi les interrogations restantes concernent la bonne utilisation des langages et outils de développement.

5.2 Modélisation

Diagramme de cas d'utilisation

Les cas d'utilisation décrivent les interactions de l'utilisateur avec le système (application) le diagramme UML des cas d'utilisation est composé des acteurs et des d'utilisations.

Un acteur représente un rôle joué par une entité externe du système (utilisateur humain .dispositif matériel ou autre système) qui interagit directement avec le système [RoVa04].

1. **Identification des acteurs:** Notre application e-commerce a comme acteur principal l'internaute.

2. **Identification des d'utilisation :** les cas d'utilisation sont identifiés à partir des acteurs du système .ainsi les cas d'utilisation principaux sont:

- ✓ **Rechercher son produit.**
- ✓ **Gérer son panier.**
- ✓ **Effectuer une commande**.

D'où le modèle UML ci-après:

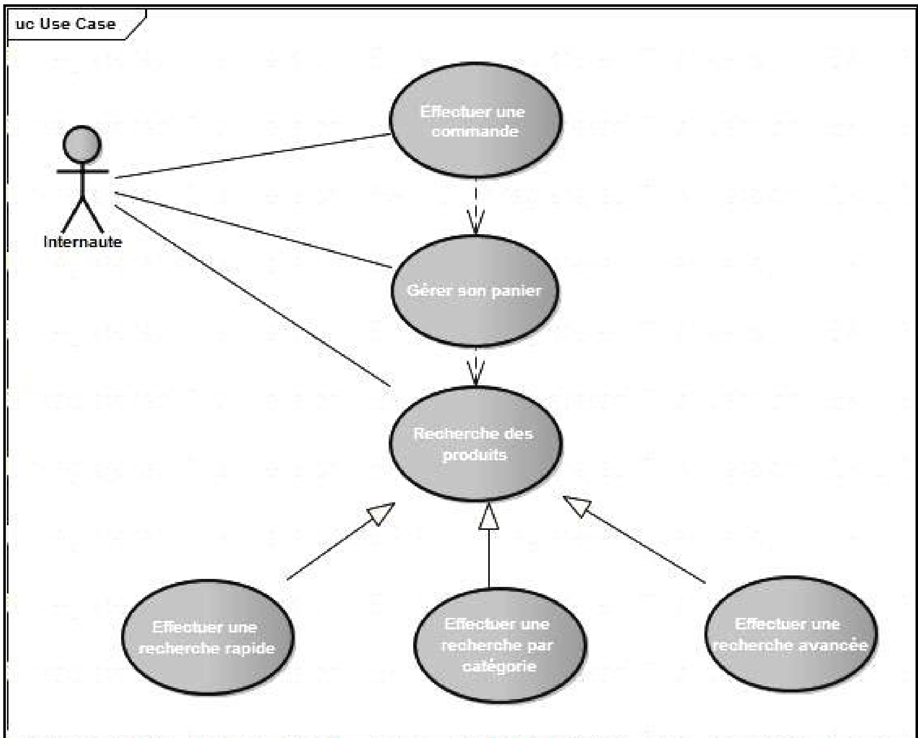


Figure 5.1 diagramme de cas d'utilisation

Ce modèle est accompagné d'une description textuelle de chaque cas d'utilisation afin d'obtenir une expression des besoins précise et décrire les différents scénarios.

1. Rechercher des produits :

a) **Acteur principal : I**nternaute
b) **Objectifs :** l'internaute veut trouver un produit le plus rapidement possible.

Scénario :

1. l'internaute lance la recherche à partir d'un mot clé.
2. le système lui affiche les résultats qui peuvent être triés par prix .On y trouvera d'autres classifications.
3. Sélectionner le produit.
4. le système affiche la fiche détaillée.

5. l'internaute peut éventuellement mettre le produit dans son panier.

Extensions :

1. l'internaute choisit d'effectuer une recherche par catégorie.
2. le système n'a pas trouvé de produit .il signale l'échec et il propose à l'internaute d'effectuer une nouvelle recherche.
3. Au cas où le système trouve beaucoup de produit .le système indique le nombre de produits trouvés .il affiche les produit sur différentes pages.

2. Gérer son panier :

a) **acteur principal:** Internaute
b) **Objectifs :** l'internaute doit avoir la possibilité d'enregistrer un produit dans un panier virtuel.

Scénario :

1. l'internaute enregistre le produit dans le panier.
2. l'internaute demande accès à son panier.
3. le système lui affiche l'état de son panier.
4. l'internaute valide son panier en demandant à effectuer une commande

Extensions :

1. le panier est vide. Le système affiche un message d'erreur à l'internaute et lui propose de revenir à une recherche de produit.
2. l'internaute modifie les qualités des lignes du panier ou en supprime et revalide le panier en demandant le recalcule du total.
3. l'internaute peut effectuer une nouvelle recherche.

3. Effecteur commande :

a) **Acteur principal :** Internaute
b) **Objectifs:** Permette à l'internaute l'accès au formulaire de bon commande

c) Pré conditions :

- Le panier n'est pas vide
- l'internaute doit être abonné.

Scénario :

1. la saisie de l'information nécessaire au paiement et à la livraison.
2. le système afficher un récapitulatif.
3. le système valide la commande.
4. le système envoie la commande au service client.
5. le système confirme la prise de commande de l'internaute.

Extensions :

1. l'internaute est déjà client (il doit s'authentifier par e-mail et mot de passe)
2. S'il n'est pas client. Il doit créer un compte pour pouvoir accéder au bon de commande.

Diagramme de séquence

Nous utilisons le terme diagramme de séquence système pour souligner le fait que le système est considéré comme une boit noir.

Le comportement du système est décrit .vu de l'extérieur .sans préjuger de sa forme de réalisation.

Les cas d'utilisation décrivent les interactions des acteurs avec le site web que nous voulons concevoir

Ces interactions se font par envoi de messages.ces messages sont représentés sous forme de diagramme de séquence UML.

Donc un diagramme de séquence est la description détaillée d'un cas d'utilisation.

Voici les diagrammes de séquence correspondent à chaque cas d'utilisation :

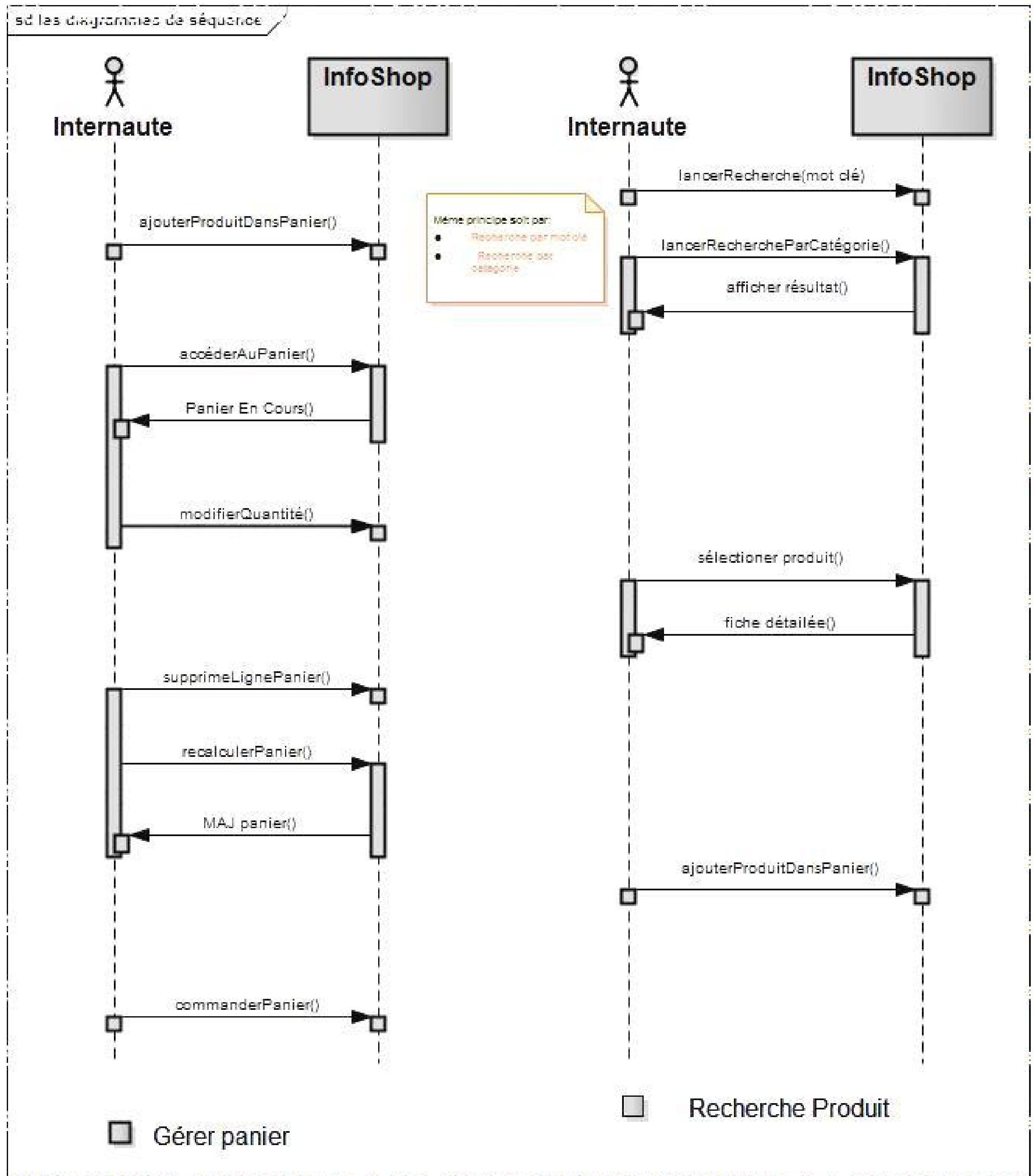


Figure 5.2 Diagramme de séquence (recherché produit & gérer panier)

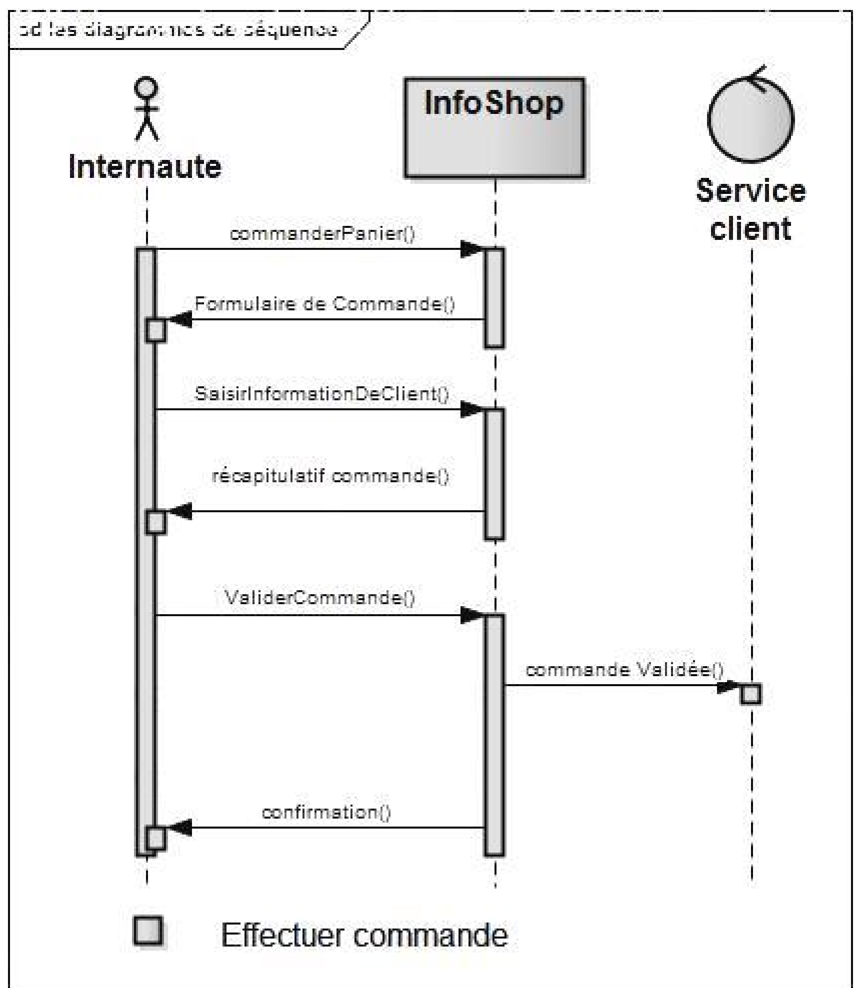


Figure 5.3 Diagramme de séquence (Effectuer commande)

Diagramme de classes d'analyse

Après avoir élaboré le diagramme des cas d'utilisation nous allons établir un diagramme initial de classes d'analyse UML ce dernier contient l'ensemble des classes constituant l'application Avant de donner le modèle il est nécessaire d'identifier les classes du système Pour ce faire nous allons procéder par cas d'utilisation.

1**. Recherche des produits:** Il n'y a qu'une seule concernée qui est « produit ».

2. Gérer son panier : Nous avons comme classe « panier » « « ligne panier » et « produit ».

3. **Effectuer commande :** Nous avons comme classe « commande » « client » « panier » « carte bancaire » et « adresse».

Un panier peut être composé d'une ou plusieurs lignes panier

Les classes pc portable .imprimante .écran. Disque dur lecteur cd et lecteur dvd héritent de la classe produit.

Après avoir combiné les classes entre elles et les relations nous obtenons le modèle suivant :

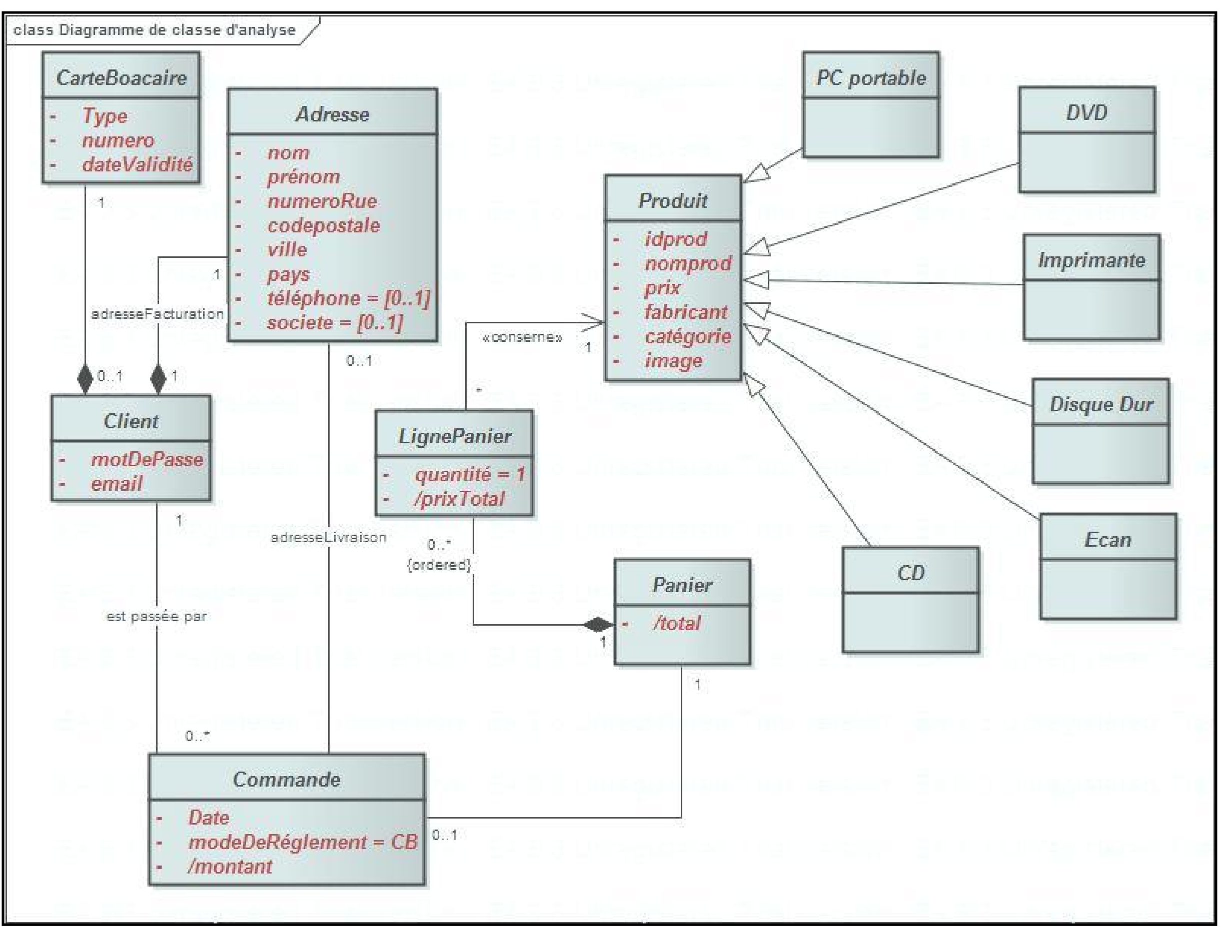


Figure 5.4 Diagramme de classes d'analyse

Diagramme de classe participantes

Pour compléter l'étude de la phrase d'analyse nous allons utiliser un diagramme de classes spécifique appelé diagramme de classes participantes qui consiste à identifier l'ensemble des classes réalisant les traitements vis avis des cas d'utilisation (les principales classes d'IHM (Interface Homme Machine) et les classes qui décrivent l'application)

Démarche : Il existe trois types de classes à identifié :

1. les classes qui permettent l'interaction entre le site Web et les utilisations Elles sont qualifiées de « dialogue »

2. le deuxième type contient la logique de l'application et il est qualifié de « contrôle » il fait la transition entre le dialogue et les entités.

3. le troisième types représente l'ensemble de classes qui décrivent les entités du système il est qualifié de « entité » Ce dernier provient du modèle de classes d'analyse déjà élaboré.

La fusion de trois types de classes leurs relations donnera ce qu'on appelle diagramme de classes participantes. Notre démarche pour élaborer ce modèle est de procéder par cas d'utilisation.

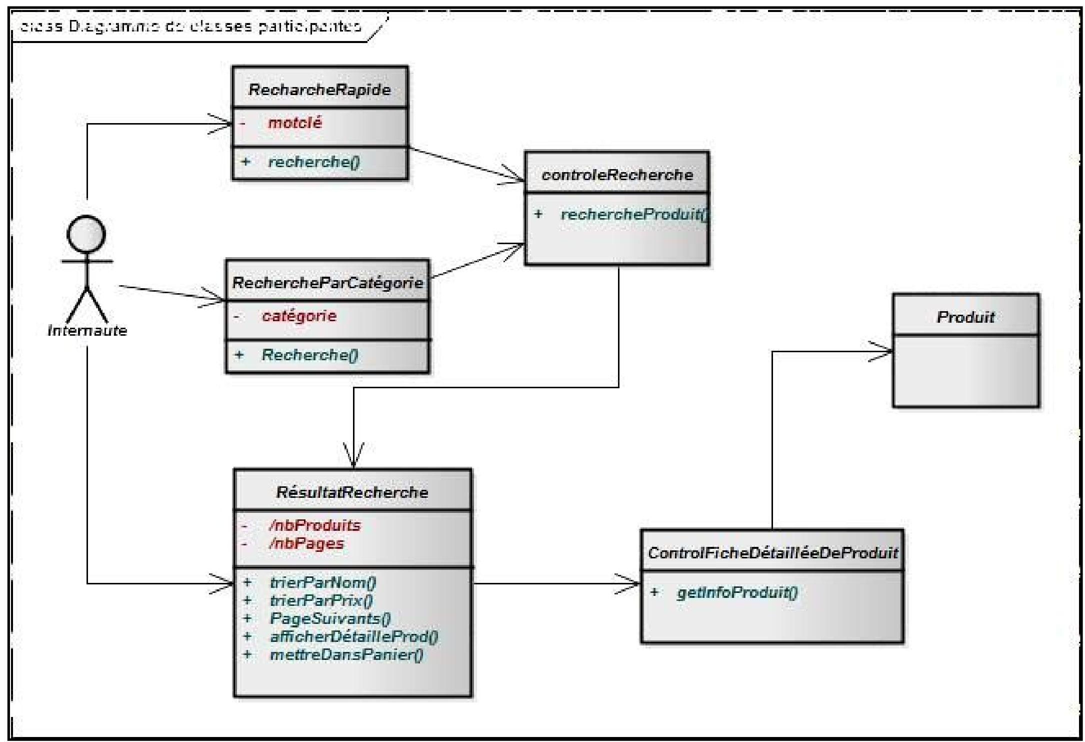


Figure 5.5 Diagramme de classe participantes (recherche produit)

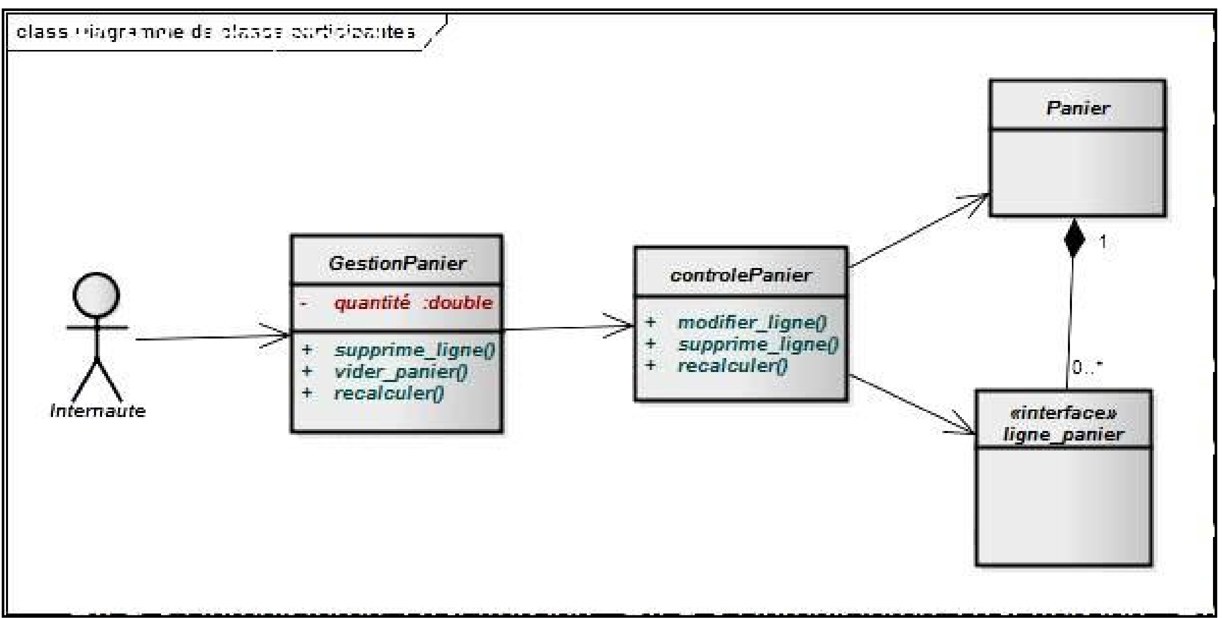


Figure 5.6 Diagramme de classe participantes (gérer panier)

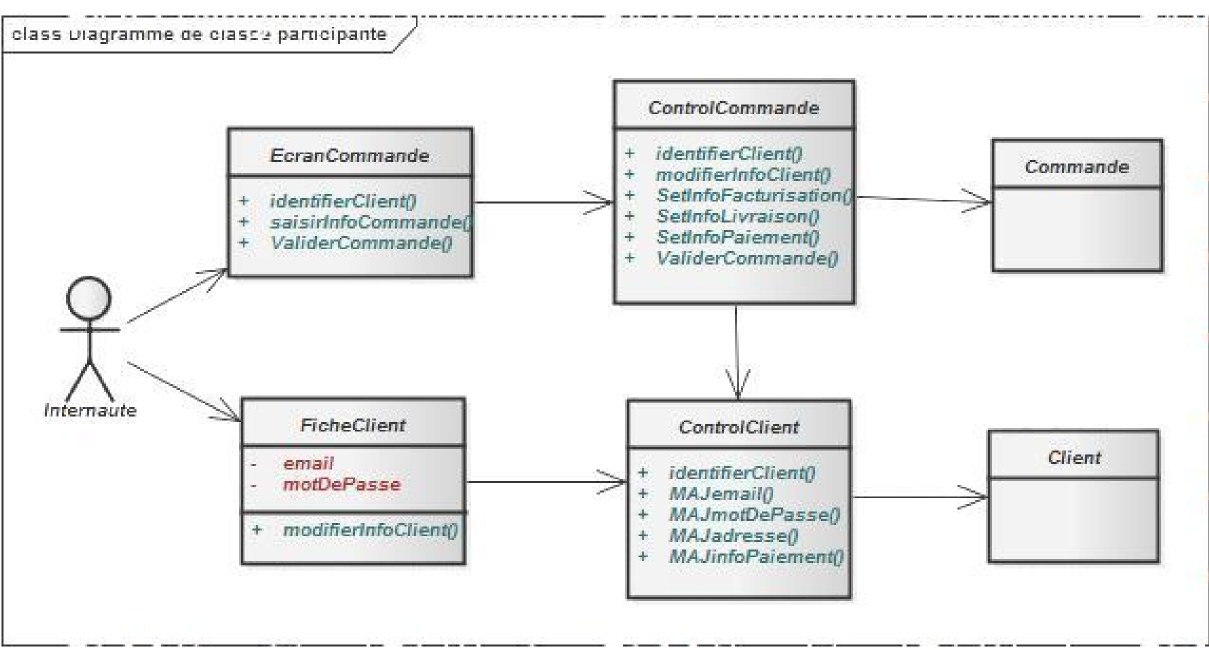


Figure 5.7 Diagramme de classe participantes(effectuer commande)

Diagramme d'activités

Un diagramme d'activités illustre la navigation dans le site web nous allons nous servir d'un certain nombre d'éléments standards pour réaliser ce diagramme en effet un diagramme d'activités contient un nombre restreint d'éléments qui sont

- Des activités
- Des transitions entre activités pouvant porter des conditions
- Des branchements conditionnels
- Un début et une ou plusieurs terminaisons possibles

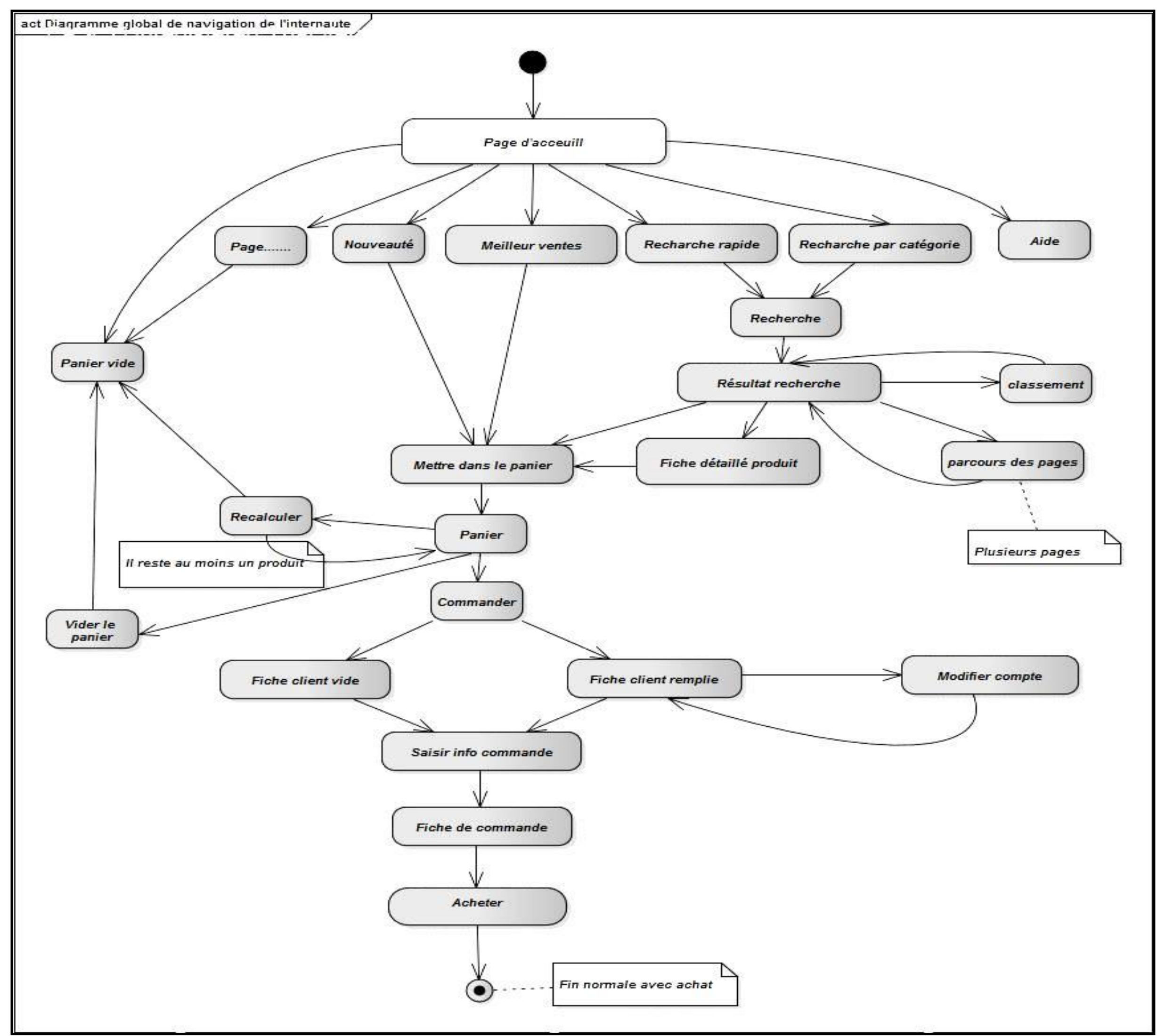


Figure 5.8 Diagramme d'activités

Diagramme de classes de conception préliminaire

Après avoir identifié les cas d'utilisation et poursuivi leur description détaillée grâce aux diagrammes de séquence complétés par des diagrammes des classes participantes on passe maintenant à la conception nous allons répartir tout le comportement du système entre les classes d'analyse Voici les diagrammes des classes de conception préliminaire.

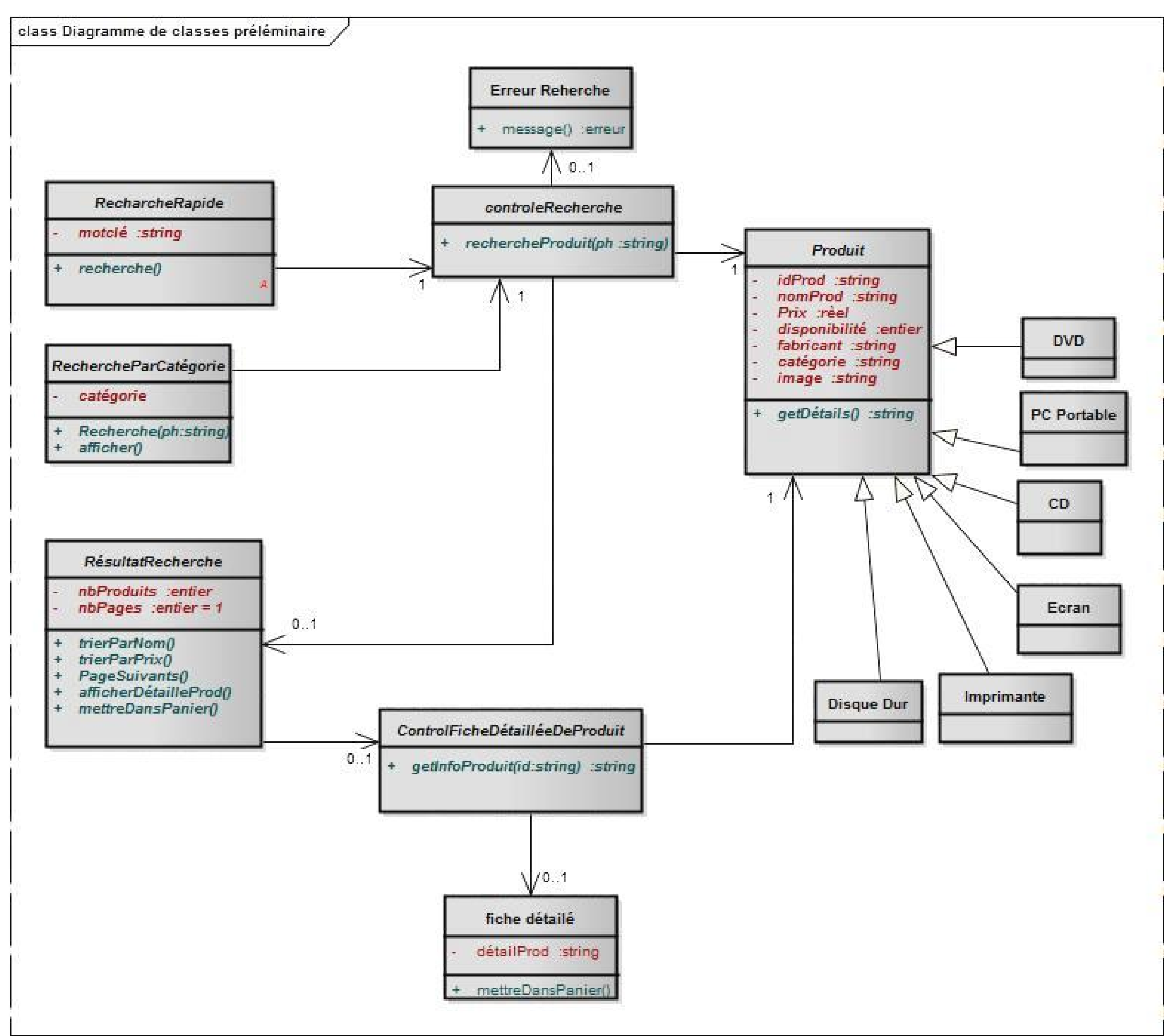


Figure 5.9 Diagramme de classes de conception préliminaire (recherche produit)

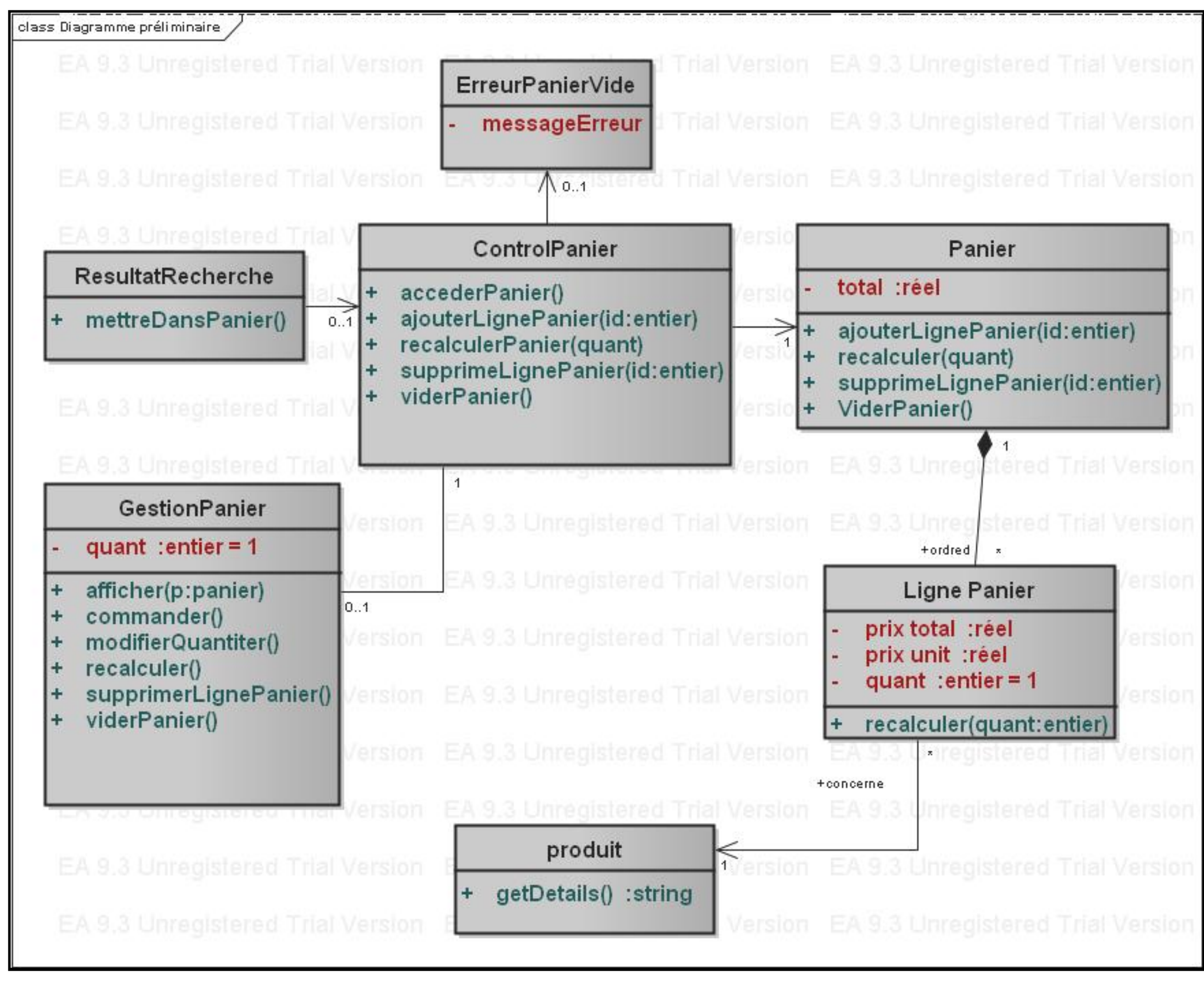


Figure 5.10 Diagramme de classes de conception préliminaire (gérer panier)

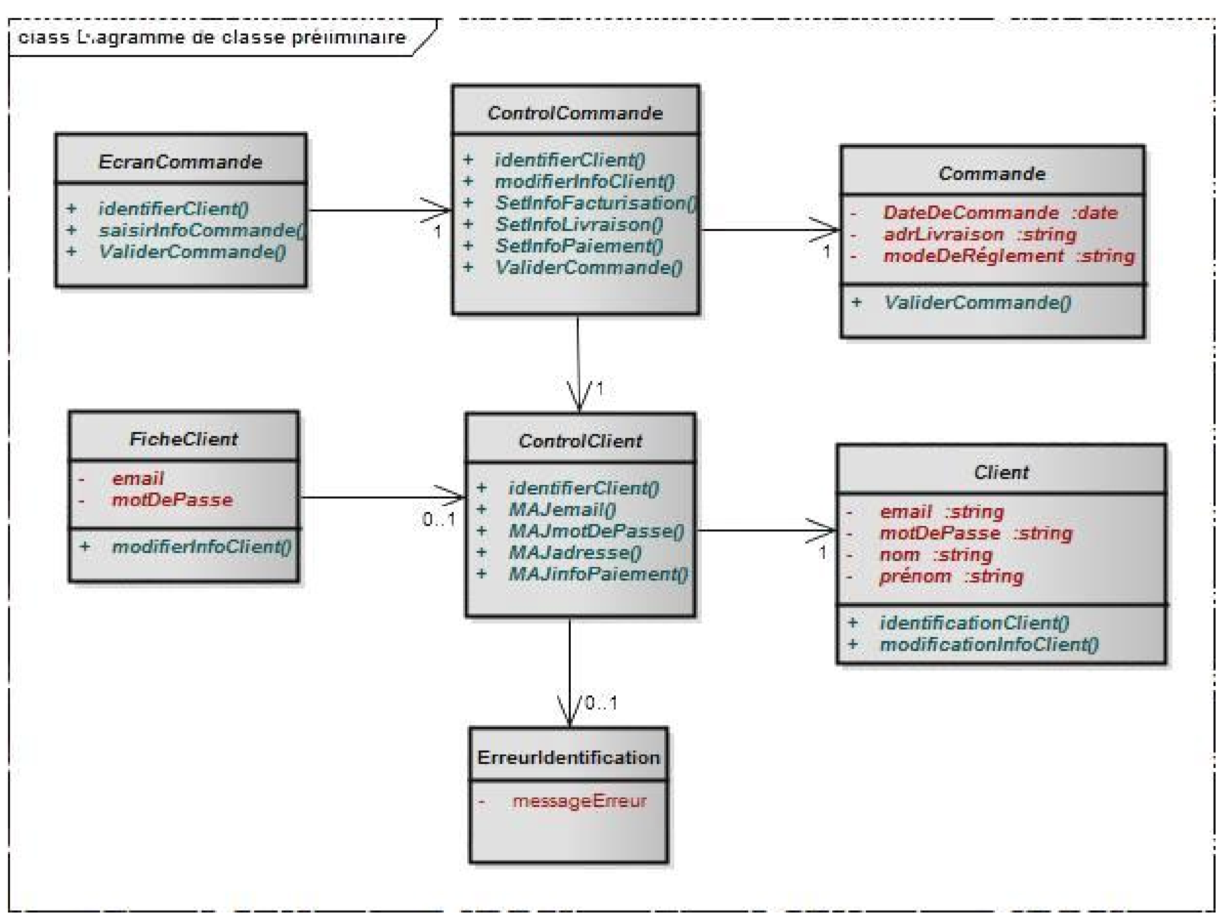


Figure 5.11 Diagramme de classes de conception préliminaire (effectuer commande)

Diagramme de classes de conception détaillée

Restituons cette dernière activité de conception dans l'ensemble du processus décrit dans les cas d'utilisation il s'agit d'affiner ce que nous avons réalisé de façon générique dans le diagramme précédent C'est-à-dire que nous allons maintenant incorporer dans les diagrammes l'architecture et les choix technologiques qui vont modifier les classes de conception préliminaire.

La modification et comme suit :

Les classes (commande, panier, produit, client, ligne panier, DVD, CD, imprimante, pc portable……etc.) devienne des classe entity beans .

Les classes de contrôles (contrôle commande, contrôle client, contrôle panier, control recherche) devienne des servelets.

Les classes (fiche détaillée, écran commande, fiche client, résultat recherche, gestion de panier ….etc.) devienne des pages JSP.

N'oublions pas l'objectif principal du modèle de conception détaillée : il peut être traduit directement en code. Avant de présenter le différent modèle voici l'architecture technique de notre application.

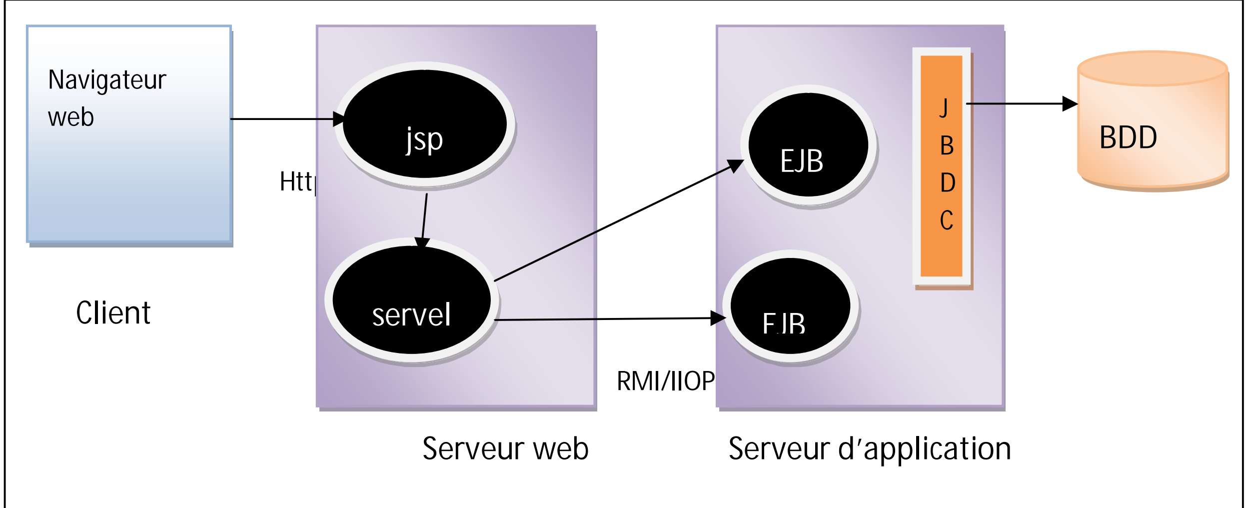


Figure 5.12 L'architecture technique de notre applications.

Le client : c'est un simple navigateur web il peut soumettre des requêtes http vers le serveur web.

Le serveur web : déclenche les traitements côté serveur et génère les résultats sous forme de page HTML les pages web hébergées par le serveur web peuvent être statique (pages HTML) ou dynamique (pages JSP)

Le serveur d'application : c'est l'exécuteur principal du logique métier coté serveur il se concentre sur la mise en œuvre des composants EJB il peut être sollicité par le serveur web à travers le Protocol RMI/IIOP.

Serveur de données (SGBD) : permet de gérer la persistance des données L'accès à ces données peut être réalisé grâce aux API offertes comme JDBC ou en ajoutant une couche intégration pour réaliser le mapping objet relationnel pour notre application nous avons utilisé une base de données relationnelle et l'API JDBC.

6. Conclusion

L'objectif principal de ce chapitre est de présenter l'étape de modélisation de notre application On a essayé de faire la liaison entre les différents diagrammes UML obtenues on a aussi présenté la structure générale de notre application.

Ce chapitre se présente comme une introduction au chapitre suivant dans lequel on va donner les grandes fonctionnalités de notre application en pressentant des exemples de code d'implémentation.

Java et .Net sont les deux principales plates-formes de développement d'applications de différents types (standalone,
Client/serveur, applications web et applications mobiles).

- Java est découpée en plusieurs plate-formes :
- Java SE (J2SE) : la plate-forme standard
- Java EE (J2EE) : la plate-forme pour le développement d'applications d'entreprise
- Java ME (J2ME) : la plate-forme pour le développement d'applications mobiles
- Java Card

.Net est découpée en deux plate-formes :

- Net Framework : pour le développement d'applications standalone et web
- . compact .Net Framework : pour le développement d'applications web

.Net semble plus homogène au niveau des API de ces plate-forme essentiellement car son

Netbeans propose des fonctionnalités permettant le développement d'applications standalone (AWT/Swing), web
(Servlets, JSP, Struts, JSF), mobile (J2ME) ou d'entreprise (J2EE/JEE).

Figure 1.1: Les différents types de client-serveur selon la classification du Gartner Group….4
Figure 1.2: Architecture d'une application sur site central……………………………………6
Figure 1.3 : architecture deux tiers……………………………………………………………9
Figure 1.4 : architecture trois tiers…………………………………………………………...10
Figure 1. 5 : positionnement du middeware…………………………………………………13
Figure 2.1: les trois dimensions d’un composant……………………………………………24
Figure 3.1: Communications au sein d’une application J2EE - vue générale……………..29
Figure 3.2 : Composants J2E…………………………………………………………………..31
Figure 3.3 : Schéma général……………………………………………………………………35
Figure 3.4 : composants EJB…………………………………………………………….....42
Figure 5.1 : diagramme de cas d’utilisation…………………………………………………66
Figure 5.2 : Diagramme de séquence (recherché produit & gérer panier)…………………69
Figure 5.3 : Diagramme de séquence (Effectuer commande)………………………………..70
Figure 5.4 : Diagramme de classes d’analyse………………………………………………..71
Figure 5.5 : Diagramme de classe participantes (recherche produit)………………………...73
Figure 5.6 : Diagramme de classe participantes (gérer panier)……………………………....73
Figure 5.7 : Diagramme de classe participantes (effectuer commande)……………………..74
Figure 5.8 : Diagramme d’activités……………………………………………………………75
Figure 5.9 : Diagramme de classes de conception préliminaire (recherche produit)………..76
Figure 5.10:Diagramme de classes de conception préliminaire (gérer panier)………………77
Figure 5.11 : Diagramme de classes de conception préliminaire (effectuer commande)……78
Figure 5.12 :L’architecture technique de notre application……………………………….....79

A

ADL : *Architectures Description Langages*

ASP : Active Server Pages

C

CMP: Container Managed Persistence

CORBA: Object Request Broker Architecture

D

DCE: Distributions Computing environnement

E

EJB : Enterprise java beans

J

J2EE : java 2 enterprise edition

JDBC : Java DataBase Connectivity

JMS: Java Message Service

JNDI: Java Naming and Directory Interface

JSP : Java Server Pages

JTA: La Java Transaction API

O

OMG: Object Management Group

R

RMI: Remote Method Interface

RPC: Remote Procedure Call

S

SDK: Software Development Kit

Création d'un application JEE avec Alexandre Baillippe, Philippe Lacomme et Raksmey Phan juillet 2010
Développons en java avec Jean Michel 1999-2010
Les cahiers du programmeur Java EE 5 © Groupe Eyrolles, 2007
Les cahiers du programmeur UML modélisé un site e-commerce Pascal Roque avec la contribution de Marline Charlmond
Une introduction aux Entreprise Java Beans Michel Riveill – Université de Nice –Sophia Antipolis

Documentations J2EE

***A Note on Distributed Computing.* J. Waldo, G. Wyant, A. Wollrath, S. Ken-dall**
http://research.sun.com/research/techrep/1994/smli_tr-94-29.pdf

***Cascading Style Sheets Level 2 Specification.* World Wide Web Consortium, May 1998**
http://www.w3.org/TR/REC-CSS2/

***Document Object Model (DOM) Level 2 Core Specification.* World Wide WebConsortium, November 2000**
http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113

***eMobile End-to-End Application Using the Java 2 Platform, Enterprise Edi-tion.* T. Violleau**
http://developer.java.sun.com/developer/technicalArticles/javaone00/eMobileApplet.pdf

***HTML 4.01 Specification.* World Wide Web Consortium, December 1999**
http://www.w3.org/TR/html4/

***HTTP State Management Mechanism.* The Internet Society, February 1997**
http://www.ietf.org/rfc/rfc2109.txt
http://java.sun.com/j2ee/

***Hypertext Transfer Protocol — HTTP/1.1.* The Internet Society, 1999**
http://www.ietf.org/rfc/rfc2616.txt

***Input Verification.* Sun Microsystems, 2001**
http://java.sun.com/j2se/1.3/docs/guide/swing/InputChanges.html

***Java Technology and XML.* T. Violleau. Copyright 2001, Sun Microsystems, Inc**
http://developer.java.sun.com/developer/technicalArticles/xml/JavaTechandXML/

Java Web Start Web site
http://java.sun.com/products/javawebstart/developers.html

Spécifications Enterprise Java Beans
http://java.sun.com/products/ejb/docs.html

***The Eight Fallacies of Distributed Computing.* P. Deutsch. Copyright 2001, Sun Microsystems, Inc**
http://java.sun.com/people/jag/Fallacies.html

The J2EE Tutorial. **S. Bodoff, D. Green, K. Haase, E. Jendrock, M. Pawlan**
http://java.sun.com/j2ee/tutorial/index.html

The Java Tutorial, Third Edition: A Short Course on the Basics. **M. Campione, K. Walrath, A. Huml**
http://java.sun.com/docs/books/tutorial/index.html

The JFC/Swing Tutorial. **M. Campione, K. Walrath. Copyright 2000, Addison-Wesley**
http://java.sun.com/docs/books/tutorial/uiswing/index.html

Webmonkey. **Lycos, 2001**
http://webmonkey.com

Résumé

Ce projet propose de mettre en ouvre la syntaxe UML a la modélisation de site de commerce en ligne, et décline l'analyse réalisé en UML sur un architecture technique JEE, facile a comprendre et suffisamment représentatif des projets e-commerce en utilisant aussi bien des langages objet purs et durs comme java.

Nous nous somme inspirés des fonctionnalités de sites e-commerce existants, comme www. Amazon.fr et, www. Eyrolles.con.

L'objectif fondamental du notre site (vente des matérielle informatique en ligne) est de permettre aux internautes de recherche des matérielles informatique par catégorie, de se constitue un panier virtuel, puis de pouvoir les commander et les payer directement sur le web.