REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE
UNIVERSITE MENTOURI DE CONSTANTINE
FACULTE DES SCIENCES DE L'INGENIEUR

DEPARTEMENT D'INFORMATIQUE

Projet de fin d'études pour l'obtention du Diplôme d'Ingénieur d'état en Informatique

Option : Systèmes d'Information

Thème

Conception et Réalisation d'une Interface Utilisateur en Java Interagissante avec des Bases de Données Distantes

Présenté par :

M. BENDRAOU REDA

M. BELKHODJA YACINE

Dirigé par :

Dr ZIZETTE BOUFAIDA

Promotion : juin-2000

Sommaire

Introduction..... 01

Chapitre 1 : Présentation générale de Java

1. Introduction.....03
2. Caractéristiques principales..... 03

2.1. L'indépendance..... 03
2.1.1. Au niveau source 03
2.1.2. Au niveau binaire..... 03
2.3. La suppression des pointeurs.....05
2.4. La Gestion automatique de la mémoire..... 05
2.5. Multiprocessus et Java.....06
2.6. Le chargement dynamique du code..... 06
2.7. Remplacement de l'héritage multiple par la notion d'interface..... 06
2.8. La nécessité d'un typage fort..... 06
2.9. Java et les standards.....06
2.10. Familiarité de La syntaxe..... 07

3. Ses atouts pour le World Wide Web.....07

3.1. Robustesse..... 07
3.2. Portabilité et neutralité de l'architecture..... 08
3.3. Sûreté..... 08

4. Java et la programmation distribuée..... 08
5. Architecture de la plate forme Java.....10

5.1. Description de la couche "Système".....11
5.2. Description de la couche "Horizontale" 12
5.3. Description de la couche "Verticale".....12

6. Acquisition d'un environnement de développement Java..... 13
7. Notion d'Applet et d'Application.....14
8. Développer avec Java.....15

8.1. Dans le cas d'une Application.....15
8.1.1. Création du fichier source.....15
8.1.2. Compilation du fichier source..... 15
8.1.3. Interprétation..... 15
8.2. Dans le cas d'une Applet Java.....16
8.2.1. Compilation du fichier source..... 18
8.2.2. Inclusion de l'applet dans une page Web..... 18
8.2.3. Visualisation de l'Applet..... 19

9. Outils pour la distribution des applets.....19
10. Conclusion..... 20

Chapitre 2 : L'accès aux Bases de données avec JDBC

1. Introduction.....21
2. Présentation de JDBC..... 21

2.1. Principales Fonctionnalités..... 21
2.2. JDBC et les standards..... 22
2.2.1. JDBC et ODBC..... 22
2.2.2. JDBC et SQL.....22

3. Architecture de JDBC..... 23

3.1. L'API JDBC..... 23
3.2. JDBC Driver API.....24
3.2.1. Choix du pilote JDBC..... 25
3.2.2. Type de bases de données exploitables par L'API JDBC.....25

4. Principaux modules de JDBC..... 26

4.1. DriverManager..... 26
4.2. Connection.....27
4.3. Requêtes SQL dans JDBC avec l'interface *Statement*..... 27
4.3.1. *Statement*..... 27
4.3.2. *PreparedStatement (extension de Statement*)..... 28
4.3.3. CallabelStatement (extension de Statement)..... 28
4.4. ResultSet 28
4.4.1. Résultat d'un SELECT..... 29
4.4.2. Résultat d'un ordre UPDATE, INSERT ou DELETE..... 30
4.4.3. Cas de valeurs nulles..... 30

5. Mise en œuvre de JDBC.....31

4.1. DriverManager..... 26
5.1. 1ére étape : Etablir la connexion avec la base de données.....32
5.2. 2éme étape : Exécution des requêtes..... 36
5.2.1. Requête simple..... 36
5.2.2. Requêtes de mise à jour.....38
5.2.3. Utilisation d'une requête précompilées : l'objet *PreparedStatement*. 39
5.2.4. Utilisation de procédures stockées : l'objet *CallableStatement*..... 41
5.3. 3éme étape : Libération des ressources..... 42

6. Sécurité..... 44

6.1. Sécurité des Applets.....44
6.2. Sécurité des Drivers (pilotes) 44

7. Conclusion.....45

Chapitre 3 : Conception d'un Système d'information relatif à la Gestion du PORTEFEUILLE

1. Introduction……46
2. Présentation de l'organisme d'accueil……46

2.1. Organigramme structurel ……46
2.2. Présentation de la Subdivision du Portefeuille…… 47
2.3. Structure de la subdivision du Portefeuille…… 47

3. Exposition du problème……48

3.1. Résultats attendus…… ….48
3.2. Choix de la Méthode……48

4. Règles de gestion…… 48
5. Le Modèle Conceptuel des Données…… 49

5.1. Dépendances Fonctionnelles…… 49
5.2. Le Graphe des Dépendances Fonctionnelles……..49
5.3. Dictionnaire de Données……..50
5.4. Règles de calcul et d'intégrité…… 51
5.5. Construction du MCD Relatif à la Gestion du Portefeuille…… 52

6. Modèle Conceptuel des Traitements…….. 53

6.1. Description des taches et Diagramme de flux…… 53
6.2. Construction du Modèle Conceptuel des Traitements…… 59

7. Modèle Organisationnel des Traitements……62
8. Modèles Externes……67
9. Validation entre le MCD et les Modèles Externes……72
10. Modèle Logique des Données……..73
11. Conclusion……..73

Chapitre 4 : Réalisation

1. Introduction…….. 74
2. Objectifs…… 74
3. Réalisation……74

3.1. 1ère Version : Une Applet…… 74
3.1.1. Architecture logicielle…… 74
3.1.2. Outils de conception utilisés……78
3.1.3. Etapes suivies dans la Réalisation…… 79
3.2. 2ème Version : Une Application…… 85
3.2.1. Architecture……85
3.2.2. Etapes suivies dans la réalisation……...86

4. Conclusion……87

Conclusion 88

Bibliographie et Ressources Internet 89

Dédicaces

Ce travail est dédié à nos très chers parents qui nous ont tant aidés et soutenus tout au long de nos études et sans qui on ne serait pas ici aujourd'hui, en espérant qu'un jour on pourra en faire autant pour eux.

A nos frères et sœurs et à toute notre famille

A nos amis (ies) et camarades de la promotion 2000.

Remerciements

Nous tenons à exprimer gratitudes et remerciements à notre encadreur Mr. Boufaida pour son précieux concours et la pertinence de ses remarques pour mener à bien notre projet.

A tous ceux et celles qui, par un geste, un conseil ou une aide désintéressée ont apporté leurs concours à la réalisation de notre étude.

Introduction

Chapitre 1:

Présentation générale de Java

Chapitre 2:

L'accès aux Bases de Données avec JDBC

Chapitre 3:

Conception d'un Système d'Information relatif à la Gestion du PORTEFEUILLE

Chapitre 4:

Réalisation

Conclusion

Introduction :

Aujourd'hui, nul ne peut nier le succès phénoménal que connaît l'Internet de par ses ressources, ses services et sa capacité à répondre plus au moins à tous les besoins et tous les goûts. Cependant, par souci d'aller toujours de l'avant, les concepteurs ne cessent d'apporter de nouvelles technologies afin de rendre la navigation sur Internet moins lassante, plus efficace et surtout d'introduire peu à peu la notion d'interactivité en mettant en œuvre des applications se basant sur le modèle du Client/Serveur (requêtes, réponses à distance) et qui pourraient s'exécuter sur n'importe qu'elle plate-forme sans souci de portabilité ou de compatibilité.

La réponse de Sun Microsystems aux besoins des entreprises et des développeurs a été le langage Java mais depuis ses débuts, Java a été considéré comme étant un langage dédié à la programmation d'applications destiné au réseau des réseaux (Internet). Cette publicité est due notamment aux fameuses "Applet", qui de nos jours, apparaissent de plus en plus sur les pages Web.

En effet, beaucoup de programmeurs ont développé de simples applications (Applets relatif aux textes défilant, aux jeux, aux montres en temps réel…et bien d'autres). Ces applications avaient pour seul objectif de rendre les pages Web plus attrayantes et plus dynamiques que celles que nous avions l'habitude de voir dans les premiers débuts de l'Internet (pages statiques et monotones).

Toutefois, le langage Java possède des outils et des fonctionnalités qui pourrait étendre son champ d'action et qui le conduiront au-delà de la conception de simples Applets d'animations.

En fait, avec ses API[1] (JDBC[2], RMI[3], AWT[4]) [2,10] ce langage offre aux développeurs la possibilité de développer des applications beaucoup plus complexes : Applications réseaux, Bases de Données et la création d'Interfaces Utilisateurs Graphiques(IUG).

La combinaison des technologies citées ci-dessus présente le support fondamental qui nous permettra d'atteindre notre objectif et qui est de réaliser une interface utilisateur graphique en Java, indépendante des plate-formes sur lesquelles elle sera exécutée et capable d'interagir avec des bases de données (Relationnelles) simultanément via des requêtes SQL.

Il est certain que cela ne se fera pas sans la maîtrise des concepts et des mécanismes propres à Java et que nous essayerons d'expliciter au fur et à mesure de notre étude.

En premier lieu (chapitre 1), nous commencerons par présenter le langage Java [9] tout en soulignant ses principaux caractéristiques, l'architecture de sa plate-forme et la différence qui réside entre une Applet et une Application.

---

(1) API : (*Application Programation Interface*)
(2) JDBC (*Java DataBase Connectivity*) : qui sera étudier en détail dans le 2ème chapitre.
(3) RMI (*Remote Methode Invocation*) : pour l'invocation d'objet à distance (sujet présenté par nos collègues)
(4) AWT (Abstact Windowing Toolkit) : pour la création de l'interface utilisateur graphique (menus, boutons, gestion des événements…etc.

Par la suite (chapitre 2), nous tenterons de définir tous les concepts et les mécanismes propre à l'API JDBC [1,8] et permettant l'interaction avec les bases de données via des requêtes SQL.

L'utilisation d'une base de données cohérente étant nécessaire au bon fonctionnement de notre application, nous avons jugé préférable de la réaliser nous même en concevant un système d'information relatif à la gestion du PORTEFEUILLE au Trésor Public de Constantine (chapitre3), afin de s'assurer de l'intégrité des données.

Enfin (chapitre 4), nous mettrons en évidence les différentes étapes à suivre afin de mettre en œuvre :

- Une application de type Intranet permettant l'accès aux bases de données distantes à travers une Applet

- Une application s'exécutant sur une machine monoposte sans aucune connexion Intranet.

1. Introduction :

De nos jours, il paraît impensable, qu'une personne qui s'intéresse de près ou de loin à l'Internet (que se sois en tant que développeurs ou simple utilisateur) de ne pas connaître ou, au moins, avoir entendu parler de JAVA. Il est souvent mentionné dans le contexte du WWW (World Wide Web) par une de ses fonctionnalités qui a fait sa renommée c'est à dire les Applets (qui seront expliquées ultérieurement) qui servent à créer des animations, des jeux ou d'autres effets interactifs.

Java est un langage de programmation orienté objet développer par *Sun Microsystems* [17] (plus connus pour ces stations de travail UNIX), adapté à la distribution d'applications sur l'Internet et s'intégrant au WWW.

Java ne se limite pas à créer des applets, il sert aussi au développement d'applications complexes. Il rend les mêmes services et résout les mêmes problèmes que d'autres langages de programmations tels que C ou C++. D'ailleurs il a été conçu sur le modèle de ce dernier (C++). De se fait, il est considéré comme un langage de programmation à part entière.

Dans les sections qui vont suivre nous allons tenter de présenter d'une manière générale le langage JAVA et essayer d'explorer ses multiples facettes.

2. Caractéristiques principales :

2.1. L'indépendance :

Son indépendance vis-à-vis des plate-formes matérielles est sont principal avantage. Il est donc tout indiqué pour les applications qui doivent tourner sur différentes plates-formes. Cette caractéristique est vitale pour tout logiciel destiner au WWW. L'indépendance, dans le cas de Java se situe à deux niveaux : source et binaire.

2.1.1. Au niveau source :

Au niveau source, les types de données primaires de Java ont la même taille, quelle que soit la plate-forme de développement. Les bibliothèques de classes standard de Java facilitent l'écriture du code, qui est ensuite porté de plate-forme en plate-forme, sans adaptation. Lorsqu'on écrit un programme en Java, on n'a pas à tenir compte des fonctions de notre système d'exploitation pour l'exécution des taches de base.

L'indépendance des plate-formes au niveau source signifie que vous pouvez déplacer des sources d'un système vers un autre puis les compiler et les faire marcher sans problème sur cet autre système.

2.1.2. Au niveau binaire :

Dans le cas de Java, l'indépendance des plate-formes ne se limite pas au source. Les fichiers binaires Java compilés sont également indépendants des plate-formes : ils peuvent tourner directement sur toute plate-forme pourvue d'une machine virtuelle Java (interpréteur de Pseudo-code), de sorte qu'il n'est pas nécessaire de recompiler le code source.

Processus de compilation :

Généralement, la compilation d'un programme écrit en langage C (ou tout autre langage) génère du code machine ou des instructions destinées au processeur. Ces instructions sont spécifiques au processeur de l'ordinateur. Ainsi, l'exécutable obtenu après la compilation du code sur une machine Pentium ne tourne que sur la même machine. Pour l'utiliser sur un autre système, il faut recompiler le code source avec un compilateur adapté. La conséquence de ceci est illustrée par la figure n° 1.1 : un exécutable par système.

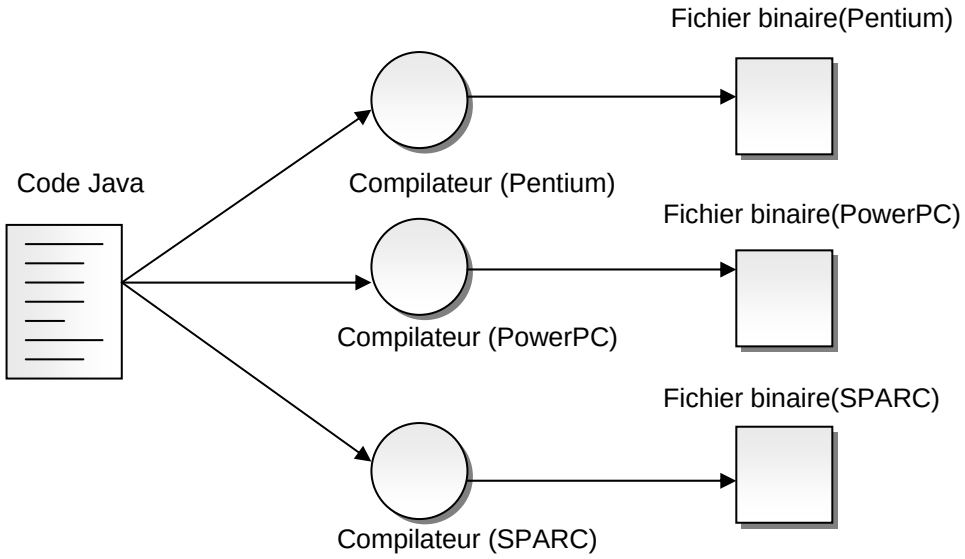


Fig. n° 1.1 : programmes classiques compilés [7].

Le mécanisme de compilation est différent pour du code Java. L'environnement de développement Java comprend un compilateur et un interpréteur. Le compilateur ne génère pas du code machine mais du Pseudo-code.

Pour exécuter un programme Java, on utilise un interpréteur qui lit le Pseudo-code et exécute le programme Java. Le Pseudo-code comporte des instructions qui ressemble à du langage machine, mais qui n'est spécifique à aucun processeur particulier. Un interpréteur spécifique à la plate-forme exécute le Pseudo-code Java. La figure n° 1.2 nous aide à mieux comprendre le mécanisme :

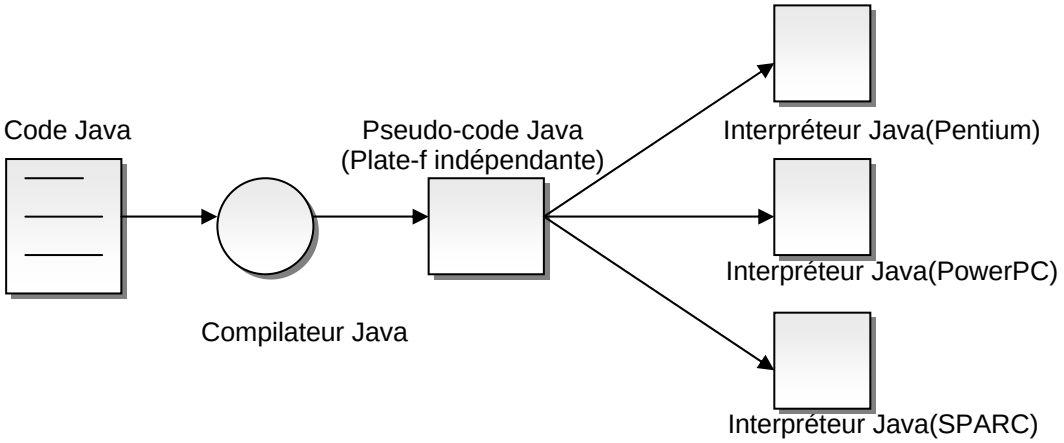


Fig. n° 1.2 : Programmes Java [7].

Interpréteur de Pseudo-code :

Dans le cas des applets l'interpréteur est inclus dans tout navigateur compatible Java, de sorte qu'on n'ait pas à se préoccuper (les applets marchent automatiquement).

Dans le cas d'applications Java, on doit disposer sur notre système d'un interpréteur pour exécuter les programmes de ces applications qui est fournit avec le **JDK** (*Java Development Kit*) [12].

2.2. Java et l'orienté objet :

Les concepts de Java sont hérités principalement de C++ dont il est issu, mais aussi d'autres langages OO (Orientés Objet). Comme la plupart de ces langages, Java possède également des bibliothèques de classes, qui fournissent les types de données de base, les possibilités d'Entées/Sorties du système et d'autres fonctions utilitaires. Ces bibliothèques de base font partie du JDK, qui comprend également des classes pour la gestion de réseau, des protocoles Internet communs et des outils d'interface utilisateur. Ces possibilités sont écrites en Java, et donc portable sur toutes les plates-formes.

Cette orientation objet n'est pas en soi la qualité principale de Java (il existe plusieurs langages OO ayant ces propriétés). Les qualités de Java résident dans l'épuration (l'assainissement) effectuée par ses concepteurs au niveau de l'implantation des concepts du langage. Ces choix d'implantation ont été guidés dans Java par des objectifs de sécurité, de sûreté de programmation et de portabilité.

2.3. La suppression des pointeurs :

Dans la plupart des langages les objets sont désignés par des pointeurs. De plus, des opérations peuvent être effectuées sur ces pointeurs (addition, comparaison, etc.). Cette pratique est considérée comme indispensable par certains programmeurs mais elle est souvent source d'erreurs.

En Java, la seule opération possible sur une référence est sa copie. Java interdit donc toutes les manipulations arithmétiques sur les pointeurs d'objets, éliminant ainsi de nombreux risques d'erreurs.

2.4. La Gestion automatique de la mémoire :

L'absence de pointeurs permet également d'établir pendant l'exécution, une liste des objets encore utilisés (ceux dont les références sont encore atteignables par les processus en cours) et une liste de ceux qui ne le sont plus. Ces derniers, désormais inutiles, peuvent donc être éliminés de la mémoire par un processus spécialisé nommé le *Garbage Collector*.

Afin d'éviter "l'émiettement" de la mémoire, on peut déplacer des objets dans la mémoire, puisque toutes les références à un objet sont connues. Java utilise à cet effet des algorithmes très puissants de nettoyage et de compactage de la mémoire (exécutés parallèlement au déroulement normal du programme).

Avantages :

- Le programme n'a plus besoin de désallouer des objets devenus inutilisables
- Le risque d'erreurs portant sur des objets encore référencés mais déjà supprimés est éliminé.

2.5. Multiprocessus et Java :

La notion de processus (*thread* : fil d'activité) est intégrée dans le langage Java. Un mécanisme de synchronisation est défini au niveau de la syntaxe, permettant de protéger aussi bien un bloc de code, une méthode ou une variable d'une exécution concurrente.

2.6. Le chargement dynamique du code:

Le code Java est chargé automatiquement s'il est utilisé. Le code peut ainsi résider sur des serveurs différents. Il peut être développé par des équipes différentes situées dans des lieux géographiques distants, et reliées par un réseau. La gestion des versions est simplifiée, le code chargé correspond toujours à la version la plus récente.

2.7. Remplacement de l'héritage multiple par la notion d'interface :

Java favorise l'héritage simple à l'héritage multiple. En effet la complexité de ce dernier n'a pas été retenue comme mécanisme d'implantation du polymorphisme en Java. Les concepteurs de Java lui ont préféré la notion d'interface. Une interface permet de spécifier des protocoles, des comportements abstraits ou génériques.

Ce mécanisme permet également d'attribuer des comportements multiples à un objet et de fédérer des objets selon leurs comportements.

2.8. La nécessité d'un typage fort :

Java supprime les conversions automatiques de types, aucune conversion n'étant effectuée implicitement. Dés lors toutes les conversions doivent être déclarées par le programmeur de manière explicite.

L'absence de pointeurs autorise une vérification accrue des types d'objet par le compilateur. En effet, aucun pointeur ne pouvant être forgé sur un objet, toutes les opérations sur celui-ci sont explicitées dans le code du programme, ce qui permet de vérifier la cohérence des types.

2.9. Java et les standards :

L'arithmétique flottante de Java est conforme à la norme IEEE 754, ce qui garantit, que les calcules effectués sur n'importe quelle machine physique supportant Java donnent toujours les mêmes résultats.

Il est ainsi possible de distribuer une tâche complexe de calcul sur un ensemble hétérogène de plates-formes. L'encodage des caractères est effectué sur 16 bits selon la norme Unicode, ce qui permet de traiter les signes diacritiques de plusieurs langues autres que l'anglais.

2.10. Familiarité de La syntaxe :

Les concepteurs de Java ont conservé les formes syntaxiques de C et de C++ pour spécifier les expressions et les structures de contrôle (boucles for, while, etc.).

Les programmeurs connaissant déjà ces langages peuvent ainsi garder certaines de leurs habitudes et ne risquent pas d'être déboussolé.

3. Ses atouts pour le World Wide Web :

Le potentiel commun d'Internet et du WWW en matière d'accès et de distribution d'informations n'est plus à démontrer. Les standards tel que HTTP ou HTML offrent un niveau élevé de compatibilité qui garantit le portabilité des informations (principalement des documents) dans de nombreux environnements. Malheureusement cette portabilité est loin d'être aussi évidente au niveau des programmes : les langages, les environnements de développement, les systèmes d'exploitation et les plate-formes matérielles définissant tous leurs propres "standards", l'incompatibilité entre systèmes demeure la seule norme.

les qualités requises pour la distribution à travers le réseau :

- Le code doit être **robuste** : c'est à dire que tout doit être mis en œuvre pour minimiser les risques de "crash".
- Le code doit être **indépendant** : le client doit pouvoir exécuter des programmes sans se soucier du type de matériel utilisé par le serveur.
- Le code doit être **sûr** : le client doit avoir la certitude de pouvoir charger des applications depuis n'importe quel point du réseau sans craindre des virus, des manipulations ou des destructions de fichiers, ni des problèmes de communication avec des serveurs malhonnêtes.

Examinant maintenant si Java remplit ses conditions (qualités) :

3.1. Robustesse :

- En éliminant l'utilisation des pointeurs, en gérant l'allocation et la désallocation de la mémoire, en ne permettant pas la surcharge des opérateurs, en utilisant les standards, en substituant les interfaces à l'héritage multiples, Java contribue grandement à améliorer la qualité et la sécurité des programmes. Les erreurs qui demeurent sont généralement dues à une mauvaise conception
- Le chargement dynamique et l'interprétation par une machine virtuelle permettent d'éviter des phases de compilation et d'édition de lien.

  Le compilateur, avec sa vérification stricte du typage, permet lui aussi de détecter de nombreuses erreurs.

3.2. Portabilité et neutralité de l'architecture :

L'utilisation de byte-code et celle des standards Unicode et IEEE754 rendent Java indépendant de la plate-forme d'exécution.

Les seules parties dépendantes des plates-formes sont encapsulées dans la machine virtuelle chargée d'interpréter les byte-codes et dans un paquetage (awt.peer) qui détermine des objets graphiques pour chaque système d'exploitation.

Une application Java saura donc s'adapter à la machine du client afin de respecter les conventions de son interface utilisateur.

L'objectif de cette portabilité et de cette neutralité est qu'une fois l'application développée, puisse s'exécuter partout, indépendamment des plates-formes choisies.

3.3. sûreté :

La sécurité, dans un environnement distribué comme Internet, est une priorité absolue. Il est impensable de tester tous les programmes avant de les chargés. La seule solution est que chaque poste de travail se protège lui-même.

L'encapsulation de l'exécution du code dans une machine virtuelle permet la portabilité par rapport à la machine physique, mais elle assure aussi un découplage par rapport aux ressources (mémoire, fichier, etc.) de cette dernière.

L'application une fois chargée, est enfermée dans la machine virtuelle et n'a donc pas directement accès aux instructions de la machine physique, ni à ses ressources.

Chacune de ses ressources sera alors vue comme un service de la machine virtuelle. L'utilisateur peut ainsi régler en tout temps les capacités de sa machine virtuelle, en fonction du degré de confiance qu'il a envers le serveur.

Enfin, ces trois qualités : **robustesse**, **portabilité** et **sûreté** étant à la portée de Java, nous pensons qu'il est **l'outil idéal** pour notre application et que cela va forcément lui assurer un succès planétaire (pour Java).

4. Java et la programmation distribuée :

Le tableau suivant représente la typologie du modèle client/serveur en considérant les apports de Java, d'Internet et des bases des données :

Type	Activité sur le serveur	Activité sur le client	Description
0	Données Traitements Présentation		Des terminaux connectés pat l'ordinateur central sur l'Internet avec Lynx, traitements par programmes sur le serveur.
1	Données Traitements Présentation	Présentation	Utilisation du Web avec un terminal X. (exp. HotJava depuis un Mac).

Table qui représente la typologie de la répartition des activités.

Type	Activité sur le serveur	Activité sur le client	Description
2	Données Traitements	Présentation	Le Web avec exécution du butineur sur le client (traitements par programmes sur le serveur), situation avant Java
3	Données Traitements	Traitements Présentation	Les traitements sont répartis entre le client et le serveur (Java, JavaScript sur le client, application Java sur le réseau.
4	Données	Traitements Présentation	Le serveur est un pur serveur de données (utilisation de Java et d'un kit de connexion aux bases de données, **JDBC** par exp**. (qui sera amplement décrit au 3ème chap**.).
5	Données	Données Traitements Présentation	Comme dans "4" une base de données locale + une connexion aux outils bureautiques locaux.
6		Données Traitements Présentation	Application Java sur le client
7	Données Traitements Présentation	Données Traitements Présentation	Au fond, la typologie n'est plus classifiante et le mode de travail sera un mélange de tout, sur le client et sur le serveur.

Table qui représente la typologie de la répartition des activités (suite) [9].

Le trio Java, base de données et Web remplit toutes les cases de la typologie. Les serveurs et les clients y acquièrent des statuts d'universalité qui les rendent indépendants. La logique du couple client/serveur est remplacée par celle d'un serveur accessible par tous les clients et d'un client connectable sur tous les serveurs.

La répartition de l'activité ne pose aucun problème. Il est possible de transférer le code Java du serveur vers le client. Le contenu des documents HTML détermine ce que le client doit exécuter

Le coût de la distribution diminue fortement. Le chargement de l'application se fait lors du chargement du document et le coût de la distribution se résume au coût de transfert, celui-ci pouvant être réduit grâce à l'utilisation d'un système de cache dans le butineur et de sites miroirs diminuant les distances et augmentant les capacités d'accès. Seul le butineur (navigateur) doit être installé sur la machine du client.

La portabilité sera maximale, autant du côté des clients qui ne connaissent pas les détails d'implantation des serveurs que du côté des serveurs qui ne connaissent pas les configurations matérielles des clients ou leur système d'exploitation.

Types de Base de Donnée exploitable par une application client/serveur Java :

Une application client/serveur Java peut exploiter l'une des trois principales architecture de BD (Base de Données) suivantes :

- Relationnelle ;
- Objet ;
- Objet-Relationnelle ;

Java profite non seulement au programmeurs souhaitant interagir avec des bases de données, mais aussi aux créateurs de ces dernières, qui disposent grâce à lui des fonctionnalités suivantes :

- Traduction d'objets en unités relationnelles facilitée ;
- Indépendance vis-à-vis des bases de données ;
- Informatique distribuée.

Remarque : Il faut noter que les bases de données jouent un rôle important dans l'univers d'Internet : en effet, il est impensable de gérer un grand nombre de documents (plusieurs milliers de pages HTML sur un même serveur) sans recourir à une BD.

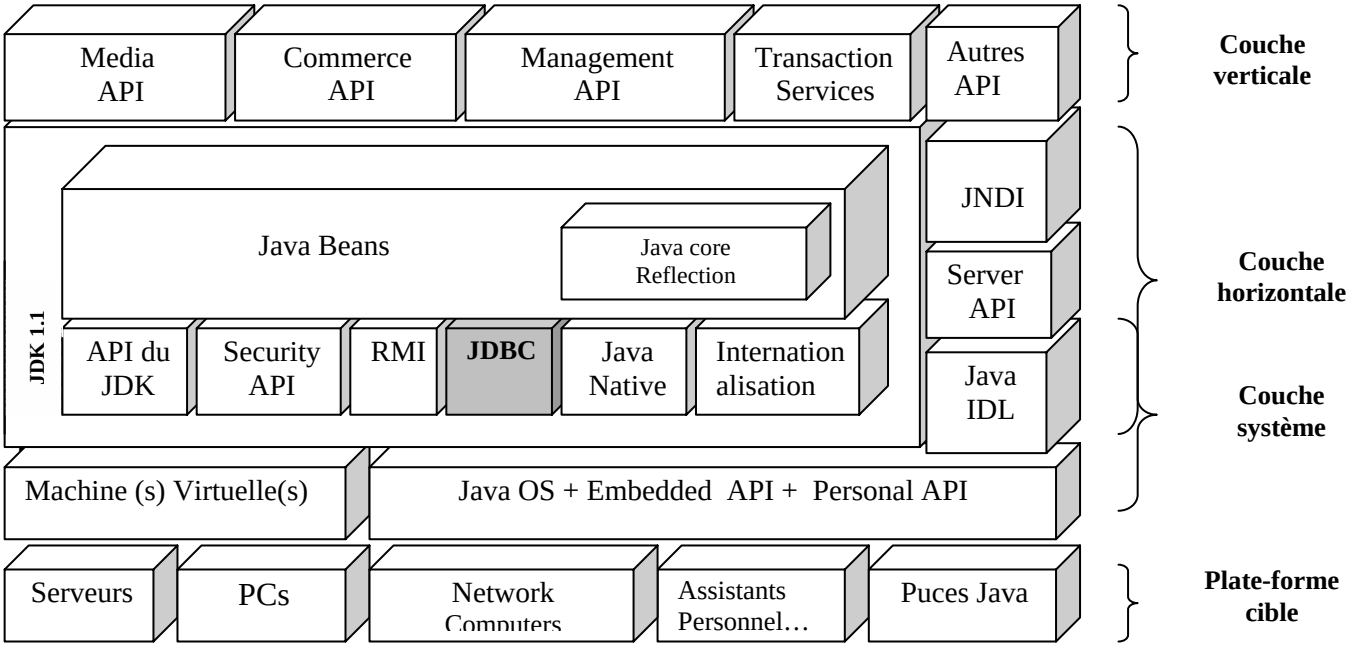
La gestion des liens, des droits d'accès et autres est simplifiée grâce à elle (BD). De plus les bases de données constituent également un moteur de recherche et de synthèse de documents.

5. Architecture de la plate forme Java :

La plate-forme Java est constituée d'un assemblage de différents composants qui sont essentiellement des librairies écrites en Java.

La plate-forme intègre un composant non écrit en Java qui est : La machine virtuelle (l'interpréteur de Pseudo-code).

Il ne s'agit pas d'un simple regroupement de différentes librairies, mais bien d'une hiérarchie de classes utilisant les mécanismes les plus fins de la conception objet (abstraction, encapsulation, héritage, polymorphisme). Le résultat est un édifice complexe où les relations entre les différents composants sont d'une richesse bien supérieure à ce qui est présenté sur le schéma qui suit :



Remarque : le module "JDBC" fera l'objet d'une étude détaillée dans le chapitre 2.

Ce schéma ne doit pas être pris comme référence de la description de la plate-forme Java mais comme une simple tentative de classification. Cette plate-forme a été conçue en ayant pour but d'atteindre les objectifs suivant :

- Faire en sorte que Java puisse tourner sur toutes les plate-formes matérielles possibles.
- Faire en sorte que les fonctionnalités mises à la disposition des développeurs soient facile d'emploi et couvrent tous leurs besoins (sécurité, réseau, multimédia, etc.).
- La plate-forme doit être suffisamment solide pour y édifier des librairies spécifiques à un segment de marché particulier : le commerce électronique, le groupware, etc.

5.1. Description de la couche "Système" :

Cette couche est, comme son nom l'indique, très proche du système d'exploitation. Elle vise l'objectif majeur de la portabilité des applications.

La stratégie de portabilité visée par les concepteurs de Java porte sur trois familles de matériels :

- Les PC : ce qui veut dire toute machine ayant une carte mère un ou plusieurs processeurs, une mémoire vive, un bus et de multiples périphériques.
- Les Network Computers : Ce sont des terminaux qui se distinguent essentiellement par l'absence de certains périphériques, comme la mémoire de masse ou le clavier pour les "set-top boxes". Les principaux marchés visés sont : les entreprises et les foyers avec la possibilité de connexion à l'Internet de manière beaucoup plus aisée (en utilisant la télévision comme périphérique d'affichage par exp.).
- Tous les autres matériels de taille encore inférieure, dont on commence à voir l'apparition sur le marché c'est à dire : assistants personnels (Organizer), téléphone portable etc. et qui peut être qu'un jour, ils pourront permettre à n'importe qui, n'importe où de se connecter à l'Internet.

Afin de pouvoir exécuter du code Java sur tous ces types de machines, les développeurs de Sun ont proposer deux solution dans la couche Système de la plate-forme Java :

- **La machine virtuelle** et son mécanisme d'interprétation du pseudo-code
- **JavaOs** : un système d'exploitation allégé capable d'exécuter du pseudo-code Java pour les systèmes ne possédant assez puissance de calcul pour supporter un mécanisme de d'interprétation requis par la machine virtuelle.

- **Personal API** : c'est un sous-ensemble du JDK (java developpement kit) destiné aux terminaux personnels (Network Computers, agendas électroniques, GSM, etc.)
  Personal API s'exécute dans moins de 2 Mo.

- **Embedded API** : un sous-ensemble encore plus petit destiné à des terminaux avec une interface graphique minimale ou, même pas du tout d'interface. Cette API s'exécute dans moins de 0,5 Mo.

5.2. Description de la couche "Horizontale" :

Cette couche a pour objectif d'offrir aux développeurs la palette la plus large de classes. Le qualificatif "horizontale" s'applique aux librairies susceptibles d'être utilisées dans toutes les applications, à l'inverse des librairies "verticales" destinée à des applications précises.

Le JDK 1.1 est composé des librairies suivantes :

- **API du JDK 1.0** : les paquetages de base de la version initiale du JDK c'est à dire : java.lang, java.io, java.net, java.util, java.awt, java.applet.

- **RMI et Java IDL** : offre d'objets distribués.

- **JDBC** : interface avec les bases de données.

- **Security API** : sécurisation du code Java, permettant la levée de certaines restrictions dans les navigateurs, par le biais de l'utilisation d'applets signées.

- **Server API** : exécution de code sur les serveurs.

- **Reflection API** : API de bas niveau utilisée essentiellement par les Java Beans.

- **Java Beans** : composants réutilisables. Cette technologie est comparable à Active-x de Microsoft.

5.3. Description de la couche "Verticale" :

Cette couche est destinée à un type d'applications particulières et donc forcément utilisée par un nombre plus restreint de développeurs.

Elle comporte les API suivantes :

- **Media API** : ensemble vaste de librairies multimédia allant du dessin 2D jusqu'à la téléphonie en Java.

- **Commerce API** : tout ce qui permettra à Java de faire du commerce électronique.

- **Management API** : ensemble d'objets standard en Java permettant de faire de la gestion distribuée de systèmes.

- **Java Transaction Services** : interface avec les moniteurs transactionnels.

6. Acquisition d'un environnement de développement Java :

Pour écrire des programmes Java, on a évidemment besoin d'un environnement de développement Java. Les navigateurs tel que Netscape ou Internet Explorer nous permettent d'exécuter des applets Java, mais ils ne nous permettent pas d'en écrire. Pour cela, il faut un outil séparé.

Le JDK (Java Development Kit) peut être utilisé dans ce but [12].

Cependant malgré l'enthousiasme qu'il suscite, ce n'est pas l'outil de développement le plus facile à utiliser, surtout si on a l'habitude d'un environnement de développement comportant une interface utilisateur graphique avec un éditeur et débogueur intégrés.

En effet l'interface du JDK, à base de ligne de commandes paraît primitive face aux logiciels avec lesquels on a l'habitude de travailler mais heureusement le JDK n'est pas le seul outil disponible.

Les environnements de développement intégrés disponible comprennent : Café de Symantec, Visual J++ pour Windows (95, 98, NT), Java Workshop pour Solaris.

Configuration du JDK :

Après l'installation du JDK (qu'on suppose faite et sans échec), la plupart des problèmes rencontrés avec Java étant dus à des erreurs de configurations.

Le JDK exige deux modifications importantes dans votre fichier **autoexec.bat** pour tourner correctement :

- le répertoire JDK1.1\bin doit se trouver sur le chemin d'exécution et la variable CLASSPATH doit avoir été initialisée

  Modifier votre autoexec.bat au moyen de votre éditeur texte. Recherchez une ligne du genre : PATH C:\WINDOWS; C:WINDOWS\COMMAND; C:\DOS;….

  Sur cette ligne il faut avoir une entrée pour le JDK ; en installant le JDK depuis un CD-ROM, la ligne devrait avoir l'aspect suivant :

  PATH C:\WINDOWS; ….C:\JDK1.1\BIN;…

  S'il n'existe pas de référence à JDK\BIN ou JAVA\BIN dans le PATH, Il faut ajouter à la fin de cette ligne le nom complet du chemin conduisant à votre installation JDK, commençant par C et finissant par BIN ; par exemple :

  C:\JDK1.1\BIN.

- La seconde addition indispensable au fichier autoexec.bat concerne la variable CLASSPATH, si elle n'y figure pas déjà. Cherchez une ligne du genre :

  SET CLASSPATH = C\JDK1.1.\lib\classes.zip ; . ;

La variable CLASSPATH peut contenir d'autres indications pour Netscape ou Internet Explorer mais ce qui vous intéresse le plus, c'est référence au fichier *classes.zip* du JDK, et au répertoire courant **(.)**.

7. Notion d'Applet et d'Application :

Un programme Java peut prendre deux formes distinctes, chacune étant adaptée à un contexte d'invocation et d'exécution différent.

La première " **Applet** " est destinée à des programmes **invoqués depuis des documents HTML**, et exécutés à l'intérieur d'un butineur (navigateur) ou d'un "appletviewer" (livrée avec le JDK) équipé d'un interpréteur Java.

La seconde " **Application** " permet de créer des applications au sens classique du terme, c'est à dire s'exécutant de manière autonome à l'aide de l'interpréteur.

Le cycle de développement est identique pour les deux formes, seul le contexte d'invocation et d'exécution varie :

- Une applet réside en général sur un serveur. Un document HTML fait référence à l'applet à l'aide d'un URL.
  Au moment du chargement du document HTML. Le butineur détecte un délimiteur d'invocation de l'applet, charge son code dans l'interpréteur intégré qui lance son exécution. L'applet s'affiche alors dans la fenêtre du butineur.

- Une application quant à elle réside habituellement sur la machine où elle est exécutée. Elle s'exécute à l'aide d'un interpréteur.

La figure n° 1.3 illustre le cycle de développement en Java et les différences entre une applet (**p**) et une application (**a**). A noter la référence du document HTML vers l'applet, qui indiquera au butineur de charger son code.

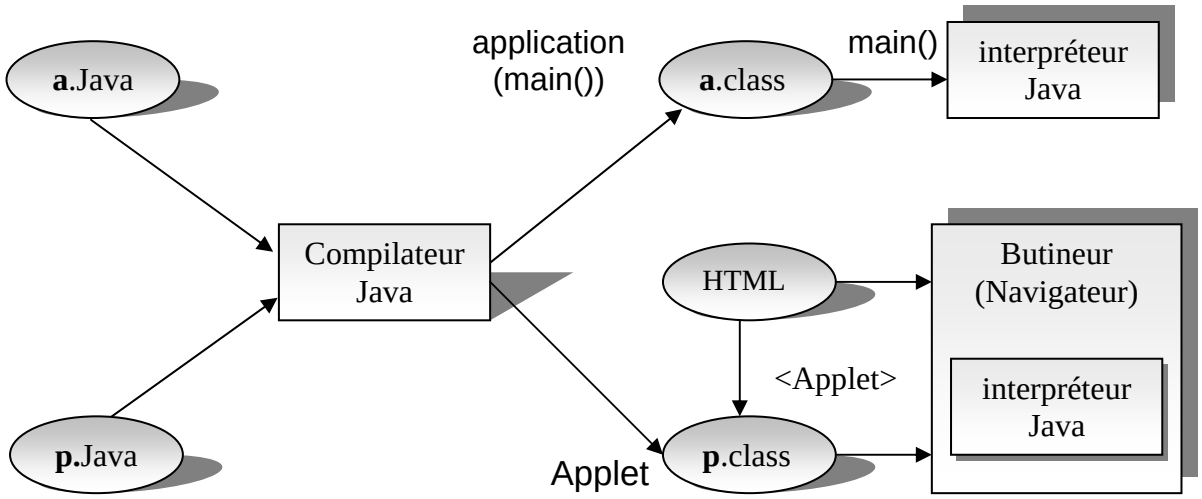


Fig. n° 1.3 : Développement et exécution d'un programme Java (applet ou application) [7].

8. Développer avec Java :

Le cycle de développement d'un programme Java est très semblable à celui d'autres langages compilés puis interprétés par une machine virtuelle. Il se déroule de la manière suivante :

8.1. Dans le cas d'une Application :

8.1.1. Création du fichier source :

Dans tous les langages de programmation, les fichiers source sont crées à l'aide d'un éditeur de texte. Java n'échappe pas à cette règle ; les fichiers sont sauvegardés en mode ASCII simple, sans caractères de contrôle. Les éditeurs exploitables sous Windows sont le bloc note ou l'Editeur DOS). Si on utilise un environnement de développement comme Café (de Symantec) on pourra travailler avec l'éditeur de texte qui lui est associé.

Note : il est important pour ceux qui travaillent sous Windows de s'assurer que le système comprend l'extension " **. Java** ".

Après avoir sélectionné l'éditeur, il ne nous reste plus qu'à saisir nos programmes Java.

N.B :Les fichiers source Java doivent avoir le même nom que la classe qu'ils définissent (y compris les mêmes lettres majuscules ou minuscules). Ils doivent avoir l'extension ".Java".

8.1.2. Compilation du fichier source :

Si on utilise un environnement de développement graphique, on dispose souvent d'un bouton ou d'une option pour lancer la compilation.

Le compilateur du JDK est piloté par des lignes de commandes. Pour utiliser ce compilateur, on doit d'abord démarrer un shell DOS. Une fois dans le DOS, on change le répertoire pour aller à l'endroit où notre fichier est stocké.

Il ne nous reste maintenant qu'à utiliser la commande : **javac** (qui veut dire Java Compiler) suivie par le nom de notre fichier tout en respectant parfaitement la casse et ca aura la syntaxe suivante :

javac MonFichier**.java**

Si tout se passe bien et qu'aucune erreur n'est signalée, on obtient un fichier appelé "Monfichier**.class**": c'est notre fichier de Pseudo-code Java.

8.1.3. Interprétation :

Lorsqu'on dispose d'un fichier de classe, on peut s'en servir au moyen de l'interpréteur de Pseudo-code Java.

L'interpréteur Java s'appelle simplement **java** et est lancé depuis le Shell DOS de même manière que javac. Donc l'exécution de notre application se fait par ligne suivante :

java MonFichier

Remarque :

Bien que le compilateur Java et l'interpréteur Java, sont deux choses différentes. Le **Compilateur** est utilisé pour créer des fichiers "**.class**" à partir des fichiers source Java et l'**Interpréteur** Java pour **exécuter** effectivement les fichiers "**.class**"

8.2. Dans le cas d'une Applet Java :

La création d'applet Java diffère de celle d'une simple application. En effet les applets Java tournent et sont affichées dans une page Web, parmi d'autres éléments. Elles suivent par conséquent des règles de comportement particulières, rendant leur création plus complexe que celle d'une application.

Comme dans le cas d'une application on commence par la création du fichier source au sein d'un éditeur de texte mais la déclaration de la classe aura la forme suivante :

Class MonExempleApplet **extends** java.applet.Applet {}

Au lieu de : **Class** MonExemple {} pour une simple application

Note : **extends** veut dire que notre classe hérite de la classe ***Applet***.

De plus, une Applet a différentes activités liées aux événements majeurs de son cycle de vie, comme l'initialisation, la peinture et les événements souris. A chacune de ces activités correspond une méthode, de sorte que lorsqu'un événement se produit, le navigateur ou un autre outil compatible Java, appelle la méthode appropriée. Les implémentations par défaut de ces méthodes ne font rien : pour fournir le comportement nécessité par un événement, on doit redéfinir la méthode adéquate dans la sous-class de notre Applet. Les lignes qui suivent, donnent une vue d'ensemble des méthodes les plus importantes pour l'exécution d'une Applet :

a- La méthode *init()* :

Elle sert à initialiser l'Applet lorsqu'elle est chargée ou rechargée. L'initialisation d'une Applet peut inclure sa lecture et l'analyse de ses paramètres, la création d'objets auxiliaires dont elle peut avoir besoin comme c'est le cas dans notre Applet ; l'établissement de la connexion, le chargement du pilote et la préparation des objets nécessaires au bon fonctionnement de l'exécution se fait dans la méthode ***init()***. Pour cela il suffit de la redéfinir:

```
public void init() {

        //……
        //Préparation des parametres necéssaire pour le bon déroulement de l'applet.
        //…….
```

```
}
```

b- La méthode *start()* :

Le démarrage se distingue de l'initialisation, car li peut se produire à des moments très différents dans la vie d'une Applet, alors que l'initialisation ne se produit qu'une fois. Le démarrage peut ainsi se produire après que l'Applet à été arrêtée parce que l'utilisateur active un lien à une page différente et son redémarrage s'effectue lorsque l'utilisateur revient à la page de l'Applet. Pour établir le comportement de démarrage d'une Applet on se doit de redéfinir la méthode *start()* :

```
public void start(){

        //La création et le démarrage du thread pour piloter l'Applet
          if (runner==null) {  //runner étant un objet Thread
              runner=new Thread(this);
              runner.start();
        //……
           }

}
```

c- La méthode *stop()* :

L'arrêt et le démarrage sont complémentaires. L'arrêt se produit lorsque l'utilisateur quitte la page contenant l'Applet qui est en train de s'exécuter. Par défaut lorsque l'utilisateur quitte une page, tout thread démarré par l'Applet continue de s'exécuter ; il est donc nécessaire de le stoppé comme le montre ces lignes de code:

```
public void stop() {

          if(runner!=null) {
              runner.stop();
              runner=null;
           }
      }
```

d- La méthode *destroy()* :

La méthode *destroy()* permet à une Applet de faire une remise en ordre juste avant qu'elle ne cesse d'être utilisée ou avant la sortie du navigateur. Elle permet par exemple de stopper et d'éliminer des threads en exécution, de fermer des connexions réseau ouvertes ou supprimer les liaisons à d'autres objets actifs.

e- La méthode *paint(Graphics g)* :

La peinture recouvre tout ce qu'une Applet dessine sur l'écran, qu'il s'agisse du texte d'une ligne, d'un arrière plan coloré ou d'une image. Contrairement aux autres méthodes décrites auparavant, *paint(Graphics g)* prend comme argument une instance de la classe *Graphics* qui est essentielle si on veut jouer sur les couleurs, la police et autres fonctionnalités graphiques.

8.2.1. Compilation du fichier source :

L'étape suivante consiste à compiler le fichier de notre applet Java. Bien qu'il s'agisse d'une applet on la compile de la même façon qu'une application c'est à dire par la ligne de commande suivante :

javac MonExempleApplet**.java**

N.B : Il ne faut pas oublier de bien respecter la casse (distinction entre les majuscules et les minuscules)en tapant le nom du fichier.

8.2.2. Inclusion de l'applet dans une page Web :

Si le fichier a été correctement tapé et que la compilation s'est bien passée, on obtient un fichier appelé *MonExempleApplet.class.*

Pour faire tourner l'applet dans une page Web, on doit, à l'intérieur du code HTML de celle-ci, faire une référence à ce fichier class en utilisant la balise **<APPLET>**. Le listing 1.1 montre un simple fichier HTML susceptible d'être utiliser à cette fin :

Listing 1.1 :code HTML avec l'applet

```
<HTML>
<HEAD>
<TITLE> Mon exemple d'applet </TITLE>
</HEAD>
<BODY>
<P> Mon applet Java dit :
<APPLET CODE="MonExempleApplet.class"   WIDTH=150   HEIGHT=25> </APPLET>
</BODY>
</HTML>
```

Afin que le listing ci-dessus soit plus compréhensible, nous allons tenter d'expliquer quelques balises :

- **<APPLET>** référence une applet dans le fichier HTML
- L'attribut **CODE** est utilisé pour indiquer le nom de la classe contenant l'applet.
- Les attributs WIDTH et HEIGHT indiquent la taille de l'applet sur la page.

Sauvegardez le fichier HTML dans le même dossier que le fichier classe (MonExempleApplet.class)

8.2.3. Visualisation de l'Applet :

Pour visualiser l'applet il nous faut l'un des éléments suivants :

- Un navigateur supportant les applets Java, comme Netscape (pour le JDK 1.1 il est préférable que la version soit supérieure à 3.0) ou Explorer (qui n'est pas vraiment compatible avec Java même pour les dernière versions).

- L'application Appletviewer, incluse dans le JDK (Java Development Kit). Ce n'est pas un programme de navigation Web, il ne permet pas la visualisation de la page Web entière. C'est une solution acceptable, faute de mieux, pour tester et voir à quoi ressemble une applet.

- Un visualisateur d'applet ou un outil d'exécution fournit par l'environnement de développement.

Avec Netscape par exemple, on utilise "Fichier/Ouvrir fichier" pour naviguer vers le fichier HTML contenant l'applet et la visualiser (il faut ouvrir le fichier HTML et non le fichier class).

N.B : Une **application** a besoin d'une méthode **main()** pour s'exécuter tandis qu'une ***applet*** à besoin d'une méthode ***init()*** pour tourner.

9. Outils pour la distribution des applets :

Il existe des sites sur lesquels on peut déposer des documents pour en faciliter la distribution (Multimania par exemple), sans avoir à héberger un serveur sur sa propre machine. Cependant, si l'on souhaite réaliser un serveur HTTP, les outils nécessaires sont :

- Une connexion réseau (Intranet/Internet) supportant un débit en rapport avec le nombre d'utilisateurs.

- Un serveur HTTP.

- Une base de données pour gérer la multitude de pages.

- Un kit de développement Java

- Un éditeur HTML

Si l'on héberge un serveur HTTP connecté à l'Internet, des outils supplémentaires sont à prévoir : gestionnaire de sécurité, gestionnaire des statistiques d'accès, etc.

Note : il n'est pas nécessaire de disposer d'une connexion Internet pour utiliser Java. De plus, le code développé étant portable, toutes vos applications peuvent être diffusées à d'autres personnes sans recompilation.

10. Conclusion :

Dans ce chapitre nous avons essayé de présenter le langage Java avec ses caractéristiques et ses objectifs.

Java est un langage de programmation Orienté Objet complètement fonctionnel. Les nouvelles fonctionnalités qu'il propose lui ont permis de se faire une place sur l'Internet. L'une de ses utilisations la plus courante actuellement est le développement d'Applets pour les navigateurs compatibles Java.

Les forces de Java sont sa portabilité (Source et Binaire), sa conception OO et sa simplicité. En plus de la création d'Applets, ces caractéristiques font de Java un langage excellent pour l'écriture de programmes plus généraux s'exécutant sans navigateurs. Pour les distinguer des Applets, ces programmes sont appelés Applications.

1. Introduction :

Les accès aux bases de données sont au cœur des applications de l'entreprise. Cette importance n'a pas échappé aux concepteurs de Java, ce qui, au contraire, les a poussés à se pencher très tôt sur ce problème.

Vu la nature distribuée de Java, sa portabilité et ses possibilités d'accès à des machines distantes le positionnent comme un langage de choix pour le développement d'applications interagissant avec des systèmes de gestion de bases de données (SGBD). De plus, les mécanismes de sécurité propres à Java limitent les possibilités d'accès à des données locales (fichiers).

Et les développeurs de Java ont très vite compris que la meilleure solution serait d'écrire un code indépendant de la base de données ou du moyen de connectivité utilisé et qu'une interface indépendante de tout SGBD est non seulement le moyen le plus rapide de donner accès à une large gamme de SGBD, mais aussi que ça serait plus profitable pour la communauté Java de définir le cadre générique d'un accès aux bases SQL en proposant une interface uniforme d'accès aux différents modules de connectivité. Cela permet aux développeurs l'écriture d'une seule interface aux bases de données, autorise le développement d'applications et d'outils indépendants d'un SGBD donné.

La réponse des concepteurs de Java à toutes ces restrictions a été la publication d'une proposition d'API (*Application Programing Interface*) définissant l'interaction entre un programme Java et une base de donnée (relationnelle dans un premier temps).

Cette API est nommée **JDBC** (*Java Data Base Connectivity*) [8,17].

2. Présentation de JDBC :

JDBC est un ensemble de classes et d'interfaces permettant de réaliser des connexions vers des bases de données, d'effectuer des requêtes (tous cela au sein du code Java), plus quelques fonctions utilitaires (conversion de types Java vers SQL et inversement).

Il fait ainsi fonction de **passerelle** entre l'univers de la **base de donnée** et celui de **l'application**.

Ces classes sont destinées à des accès dits de bas niveau : en aucun cas la complexité de la syntaxe SQL et des requêtes n'est cachée.

2.1. Principales Fonctionnalités :

En général un programme utilisant JDBC fonctionne de la façon suivante :

1- Il établit la connexion avec une source de données.
2- Il effectue des requêtes.
3- Il utilise les données récupérées afin de les traiter ou les afficher.
4- Il achève la connexion.

2.2. JDBC et les standards :

Les concepteurs de JDBC ont pris le soin de faire accepter cette API par l'ensemble des acteurs concernés : développeurs de SGBD, d'environnements de programmation, etc.

Ils ont donc tout fait pour utiliser des éléments déjà connus, à savoir **ODBC** et **SQL**.

2.2.1. JDBC et ODBC :

La première question qui vient à l'esprit et qui mérite d'être éclaircie avant même d'être posée est *: pourquoi ne pas utiliser carrément ODBC ?*

Tout simplement parce que ODBC est trop lié au langage C, c'est à dire :

1- ODBC se base sur des éléments propres au C tel que les types, les pointeurs (qui n'existe pas en Java) ce qui rend l'intégration d'ODBC dans Java un peu délicate bien que techniquement réalisable.

2- ODBC est réputé pour être compliqué et difficile à apprendre et à mettre en œuvre.

3- Enfin, le détail le plus important est que l'intégration Java-ODBC s'écarte de la devise de Java en terme d'uniformité, de robustesse et de simplicité.

Les concepteurs de Java ont choisi de réaliser dans un premier temps l'API JDBC que l'on peut situer «au-dessus» de la couche ODBC, celle-ci jouant le rôle de sous-protocole et pouvant être utilisée à l'aide de passerelles JDBC-ODBC.

2.2.2. JDBC et SQL :

JDBC supporte les fonctionnalités de bases de SQL, telles qu'elles sont définies dans la norme ***SQL2 Entry Level.*** Ce choix se justifie par le fait qu'il représente le «dénominateur commun » en matière de SQL entre les principaux SGBD relationnels.

Ici encore, l'idée des concepteurs est d'offrir rapidement un noyau de bas niveau commun et standardisé, des fonctionnalités de plus haut niveau pouvant être définies par la suite sous forme de nouvelles API ou d'outils Java invoquant JDBC.

JDBC permet à un programme Java de transmettre n'importe quelle requête à un SGBD, permettant ainsi l'utilisation de fonctionnalités propres à celui-ci. Cette possibilité risque néanmoins de compromettre sérieusement la portabilité de ce programme sur un autre SGBD.

Un label de conformité "JDBC Compliant ^{Tm} " a été défini par Sun. Un driver JDBC désirant obtenir ce label devra se soumettre à des tests montrant qu'il supporte au minimum *SQL2 Entry Level.*

A long terme, le but est de supporter la totalité de la norme SQL2, lorsque celle-ci aura vraiment été adoptée et surtout lorsque les principaux SGBDs la supporteront de manière complète.

Correspondance entre types Java et SQL :

Le tableau suivant résume les conversions possibles :

TypeSQL → TypeJava ↓	TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DECIMAL, NUMERIC	CHAR, VARCHAR, LONGVARCHAR	BINARY, VARBINARY, LONGVARBINARY	DATE, TIME-STAMP	TIME
String	Possible	Possible	Possible	Possible	Possible
BigDecimal Boolean Integer, Long Float, Double	Possible	Possible	—	—	—
byte[ ]	—	—	Possible	—	—
java.sql.Date	—	Possible	—	Possible	—
java.sql.Time	—	Possible	—	—	Possible
java.sql. Timestamp	—	Possible	—	Possible	—

Tableau résumant les conversions possibles(Java-SQL et SQL-Java) [2].

3. Architecture de JDBC :

Il y a deux principaux ensembles d'interfaces. Le premier concerne l'API pour l'écriture d'applications. Le second est une API de bas niveau pour la conception de pilote JDBC.

3.1. L'API JDBC :

L'API JDBC se matérialise sous forme d'une série d'interfaces abstraite au sens de Java, autorisant un programmeur à se connecter à une base de données particulière, à exécuter des ordres SQL et à traiter les résultats.

Les principales interfaces définies dans l'API JDBC sont décrites dans le tableau qui suit ainsi que leurs hiérarchisations dans la figure n° 2.1 :

Interface	Description
Java.sql.Connection	Gère les connexions existantes, gère les transactions
Java.sql.DriverManager	Gère le chargement des drivers et la création des connexions à des BDs.
Java.sql.Statement	Gère les requêtes à exécuter et reçoit les résultats.
Java.sql.ResultSet	Gère l'accès aux tuples d'un résultat d'un objet *Statement* donné.
Java.sql.PreparedStatement	Représente une requête paramétrée (précompilée)
Java.sql.CallabelStatement	Permet de faire appel à des procédures stockées.

Tableau représentant les principales Interfaces de l'API JDBC [9].

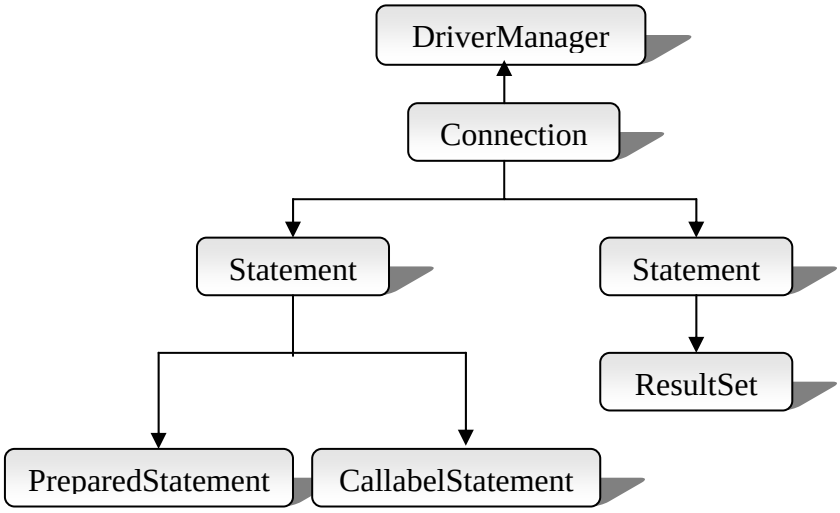


Fig. n° 2.1 : Relations de composition entre les objets JDBC [2].

3.2. JDBC Driver API :

L'API JDBC est, dans son ensemble, un modèle à couches. La plus externe est celle visible (utile) pour développer des applications. Quand on se réfère à **l'API JDBC**, c'est en fait à ce sous-ensemble que l'on pense. Il est représenté par le paquetage ***java.sql.***

Les couches inférieures sont destinées à faciliter l'implémentation de pilotes pour des bases de données. Elles représentent une interface entre les accès de bas niveau au moteur du SGBD et la partie applicative de Java.

L'interface *java.sql.Driver* est destinée aux développeurs de drivers (pilotes) désirant interfacer un SGBD à Java en utilisant JDBC. La programmation d'un driver JDBC consiste à implanter les éléments définis dans les interfaces abstraites de l'API JDBC (décrites précédemment). Le digramme suivant (*Fig*. n°2.2) donne vision globale de ces différentes couches :

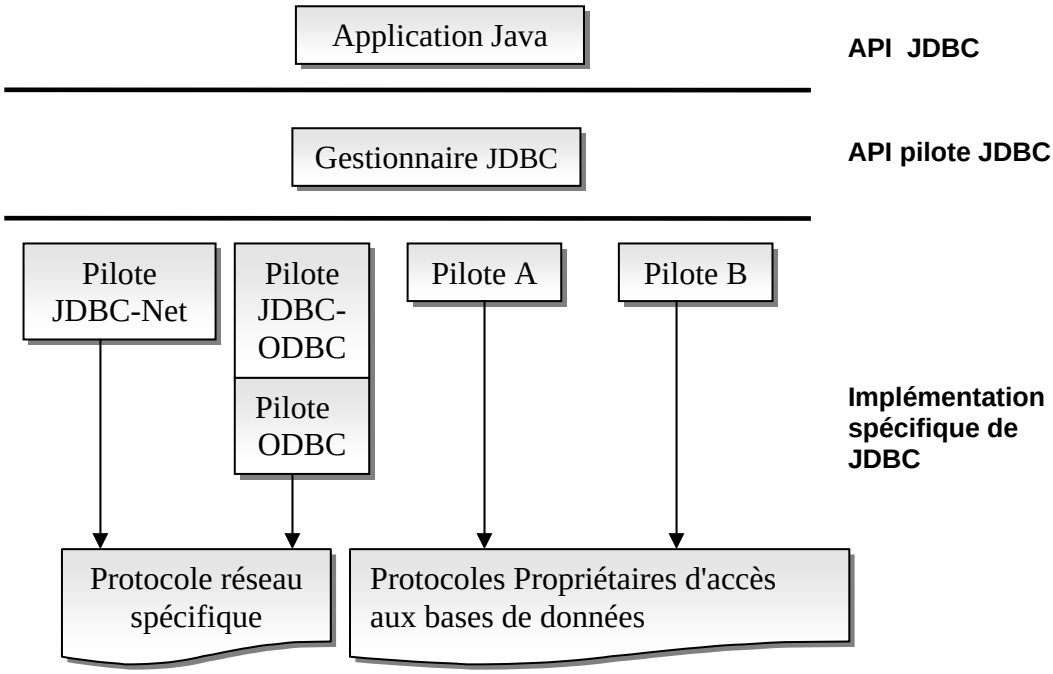


Fig. n° 2.2 : Le modèle en couche de JDBC [9].

De plus chaque pilote a besoin de fournir une classe qui implémente l'interface java.sql.Driver utilisée par la classe générique *java.sql.DriverManager* lorsqu'elle a besoin de connaître le pilote associé à l'URL d'une base de données.

3.2.1. Choix du pilote JDBC :

Le marché des pilotes est en pleine expansion et de nos jours il est souvent très facile de se procurer un pilote qui convient le mieux à notre application grâce à *l'Internet,* en plus, vu la médiatisation qu'a eu Java ces derniers temps, les éditeurs de logiciels propose chaque jour de plus en plus de pilotes supportés par les SGBD les plus utilisés.

Pour rendre les choses plus claires, JavaSoft a classifié les pilotes en quatre types :

1- Pilotes accédant aux bases de données grâce à une technologie de ponts.

   Exp. : le pont ODBC inclus dans le *JDK1.1* et qui fournit une passerelle à l'API ODBC, dont les implantations assurent l'accès aux bases de données.

   Les ponts requièrent en général d'installer du code natif sur les systèmes clients, ce qui est dans certain cas n'est pas toléré.

2- Accès en Java aux librairies natives du moteur de la base de données. Nécessite l'emploi d'une librairie native.

3- Ce type fournit au client une API réseau générique qui une fois traduit au niveau du serveur permet l'accès aux bases de données.

   Le pilote sur la machine du client communique au moyen de sockets avec une application intermédiaire sur le serveur, qui convertit les requêtes du client en appels à l'API spécifique du pilote souhaité.

   Avantages :
   - l'installation d'un code dans la machine cliente n'est plus nécessaire.
   - un seul pilote permet l'interaction avec de nombreuses bases de données

4- En utilisant des sockets Java, ces pilotes interagissant directement avec les gestionnaires de bases de données et ca semble être la solution la plus adéquate et plus directe.

3.2.2. Type de bases de données exploitables par L'API JDBC :

Une application utilisant JDBC comme moyen d'interaction avec une BD peut exploiter l'une des trois principales architectures de base de données suivantes :

- Relationnelle ;
- Objet ;
- Objet-Relationnelle ;

Remarque :

L'immense majorité des applications lourdes actuelles mettent en œuvre des bases de données relationnelles. L'API JDBC reflète cette tendance puisqu'elle favorise ce type de base et leur langage de requête standard, SQL. Les bases de données relationnelles sont à l'opposé de la philosophie orienté objet de java.

Traduire la réalité au moyen d'objet Java suppose par conséquent de créer une couche logicielle, absente de JDBC, permettant de stocker les objets dans une base de données relationnelles puis de les relire.

4. Principaux modules de JDBC :

Les points suivants décrivent les principaux éléments (interfaces, classes) composant JDBC. En plus du *DriverManager* (gestionnaire des pilotes), les notions de *Connection*, de *Statement* et de *ResultSet* sont indispensables pour interagir avec la base de donnée.

Remarque :

Il va sans dire que les classes et les interfaces qui vont être décrites, se trouve dans le paquetage ***java.sql***.

4.1. DriverManager :

Le *DriverManager* gère la liste des drivers JDBC chargés dans la machine virtuelle Java.

Plus précisément : il décortique les URL (Uniforme Ressource Location) afin d'en extraire le sous-protocole et le service, puis recherche dans sa liste de drivers celui (ou ceux) capable d'établir la connexion.

La figure n° 2.3 : montre comment une application, utilise JDBC pour communiquer avec **une** ou **plusieurs bases de données** sans connaître les détails de l'implantation du pilote concerné.

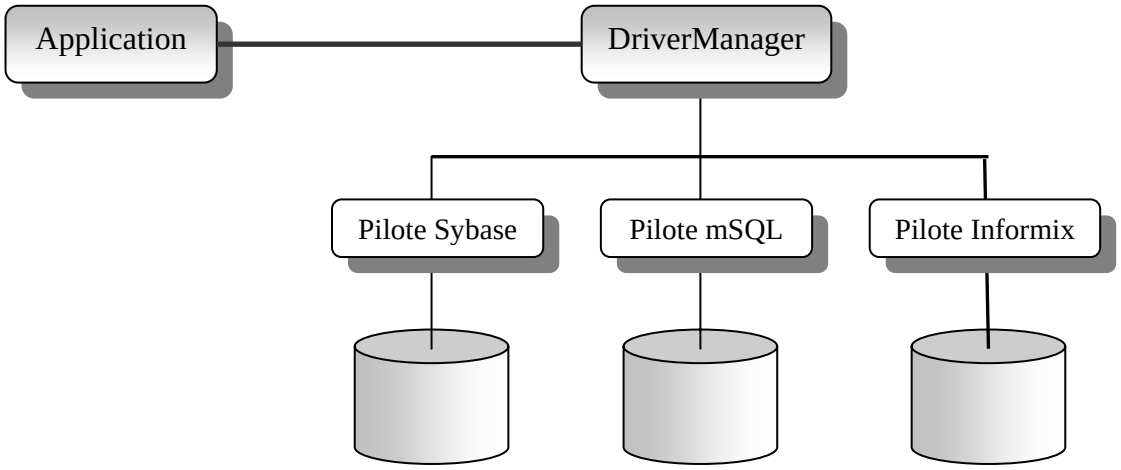


Fig. n°2.3 : JDBC cache les spécificités des implantations des bases de données [8].

Note : Dans le cas ou plusieurs pilotes sont disponibles le *DriverManager* consulte une propriété Java (*sql.Driver*) que l'utilisateur a la possibilité de configurer ; cette propriété regroupe une liste de noms de drivers.

Le *DriverManager* parcourt la liste dans l'ordre, choisissant le premier pilote utilisable.
Ou cas où la propriété *java.sql* n'est pas définie c'est l'ordre dans lequel les drivers ont été chargés qui sera considéré.

4.2. Connection :

Une *Connection* représente un canal de communication (une connexion) entre un programme Java et une base de données. La création de protocole et le service à utiliser. Lors de la création d'une *Connection*, des arguments tels que le nom de l'utilisateur et son mot de passe peuvent être fournis sous la forme d'objets de la classe *java.util.Properties.*

Une fois créée, une *Connection* va servir à envoyer des requêtes au SGBD et récupérer les résultats, ces deux tâches étant effectuées à l'aide des interfaces *Statement* (requête) et *ResultSet* (ensemble de résultat).

Transactions :

L'interface *Connection* est capable de gérer des transactions SQL. Par défaut, une *Connection* est placée en mode *commit* (confirmation) automatique : c'est à dire que chaque requête est exécuter dans une *transaction* SQL séparer puis confirmée automatiquement. Ce mode peut être désactivé (méthode *setAutoCommit(Booléen*)) afin de regrouper l'exécution de plusieurs requêtes en une seule *transaction*. En mode non automatiquement, la *Connection* maintient ouverte une transaction "courante", qui peut être confirmée (*Connection.commit()*) ou annulée (*Connection.rollback()*) au moment voulu. La *Connection* libère alors cette *transaction* et en crée immédiatement une nouvelle.

4.3. Requêtes SQL dans JDBC avec l'interface *Statement* :

Des trois classes JDBC représentant des instructions SQL, celle-ci (***Statement***) passe pour la plus simple et la plus facile à manipulée et effectue toutes les instructions SQL élémentaires.
Il y a aussi deux autres interfaces : ***PreparedStatement*** pour exécuter les requêtes paramètrées et ***CallabelStatement*** pour les procédures stockées paramètrées.

Note : Une requête doit être obtenue auprès d'une *Connection* existante avant de pouvoir être préparée et exécutée. Le résultat d'une requête est obtenu auprès de la requête elle-même.

4.3.1. *Statement* :

Statement est destinée aux requêtes SQL simples et dont l'expression SQL pouvant être transmise telle quelle au SGBD.

Une fois obtenue auprès d'une *Connection*, une requête peut être exécutée, son résultat prenant la forme d'un *ResultSet* dans le cas d'un SELECT.Ce résultat peut alors être vérifié et parcouru ligne par ligne.

Les ordres DELETE, INSERT, UPDATE retournent un entier représentant le nombre de tuples touchés.

4.3.2. *PreparedStatement (extension de Statement) :*

PrepareStatement permet la création de requêtes contenant des paramètres. Son utilisation prend tout son importance lors d'utilisation d'une même requêtes plusieurs fois et où il n'y a que la valeur (paramètre) d'un (ou plusieurs) champ qui change et non pas le champ lui-même.

Son code SQL contient des caractères ' ? ' indiquant l'emplacement des paramètres, qui sont ensuite remplacés par des valeurs claires (réels) au moment de l'envoi de la requête à la *Connection* associée.
La fixation des paramètres se fait à l'aide des méthodes *PreparedStatement.setXXX()* où XXX représente le type du paramètre fixé (entier, chaîne de caractères, float. etc.).

4.3.3. CallabelStatement (extension de Statement) :

La classe *java.sql.CallabelStatement* permet d'exploiter les procédures stockées. Cette fonctionnalité des bases de données offre plusieurs avantages par rapport à la simple saisie explicite du code SQL dans le code source de l'application.

Avantages :

- Précompilée dans la base de données elle-même à l'intention de tous les gestionnaires, une procédure stockée s'exécute plus rapidement qu'un code SQL qui doit être réinterpréter à chaque fois qu'il lui parvient.

- Pour utiliser une procédure stockée, il suffit d'en connaître le nom ainsi que la nature de ses entrées et sorties.

- Les erreurs de syntaxe peuvent être décelées lors de la compilation plutôt que dans l'exécution.

Les procédures de ce type acceptent généralement des paramètres en entrée (IN) et fournissent d'autres paramètres en sortie (OUT).

L'assignation des paramètres (IN) s'effectue à l'aide des méthodes *PreparedStatement.setXXX().* Les paramètres OUT doivent être déclarés *CallabelStatement.registerOutParameter()* avant l'exécution de la procédure : on invoque une fois cette méthode pour chaque paramètre OUT, en indiquant son type SQL.

Après l'exécution, les paramètres OUT peuvent être récupérés à l'aide des méthodes CallabelStatement.getXXX() adaptées à leurs types.

4.4. ResultSet :

Une requête SQL produit différents types de résultats. Dans le cas d'une insertion, d'une mise à jour ou d'une suppression, le résultat retourné par le SGBD est généralement un entier indiquant uniquement le nombre de tuples ayant été touchés par la modification.

Par contre dans le cas d'une sélection (ordre SELECT), le résultat, fourni sous la forme d'un ensemble de tuples géré par un *ResultSet*, est beaucoup plus intéressant :

4.4.1. Résultat d'un SELECT :

La méthode *executeQuery()* exécute une requête et retourne le résultat sous forme d'un *ResultSet*. Un *ResultSet* est un ensemble de lignes (chaque ligne représentant un tuple de la relation résultat) ; chaque ligne comporte le même nombre de colonnes (chaque colonne représentant un attribut de la relation résultat).

De nombreuses méthodes sont définies dans la classe *ResultSet* afin de pouvoir parcourir le résultat ligne par ligne et d'accéder de différentes manières aux colonnes de la ligne courante. Au niveau du programme Java, une simple boucle *while* combinée à la méthode *ResultSet.next()* permet de récupérer chaque ligne du résultat (voir le paragraphe suivant).

4.4.1.1. Utilisation de la méthode next() :

Supposons une variable rs, qui est une instance de *ResultSet*. Pour accéder aux différents champs nous allons parcourir chaque ligne et retrouver les valeurs en fonction de leurs types. La méthode next déplace ce qui est appelé un curseur, jusqu'à la ligne suivante et fait de cette ligne (appelée la ligne courante) celle sur laquelle nous pouvons agir. Comme le curseur est positionné juste au-dessus de la première ligne d'un objet *ResultSet*, le premier appel de la méthode *next* déplace le curseur sur la première ligne et en fait la ligne courante. Les appels successifs de la méthode *next* déplacent vers le bas le curseur, ligne par ligne, du haut vers le bas. à notez qu'avec l'API JDBC 2.0, on peut déplacer le curseur en arrière, à des positions spécifiques, et à des positions relatives à la position courante, tout cela en plus de pouvoir déplacer le curseur en avant.

4.4.1.2. Utilisation des méthodes getXXX() :

Nous utiliserons les méthodes getXXX() du type approprié pour retrouver les valeurs dans chaque colonne. Par exemple, la première colonne dans chaque ligne de rs, contient des valeurs de type VARCHAR. La méthode pour retrouver une valeur de type SQL VARCHAR est getString(). La seconde colonne dans chaque ligne contient une valeur de type SQL FLOAT, et la méthode pour retrouver les valeurs de ce type est getFloat().Chaque fois que la méthode next est invoquée, la prochaine ligne devient la ligne courante et la boucle continue tant qu'il y a des lignes à lire dans rs.

4.4.1.3. Utilisation de la méthode ResultSetMetaData :

Elle appartient au *java.sql.* Cette classe apporte des précisions sur les objets *ResultSet* générés en réponse à une requête, que l'on peut utiliser pour rendre le résultat plus compréhensible et agréable à consulter. Pour être plus précis elle nous éclaire sur :

- Les noms des colonnes.
- De quelle table la colonne choisie provient-elle?
- Quel type de données la colonne choisie renferme-t-elle?

L'utilisation de la méthode *getMetaData()* est indispensable pour obtenir des précisions sur un objet *ResultSet.*

4.4.2. Résultat d'un ordre UPDATE, INSERT ou DELETE :

Les ordres de mise à jour ne retourne généralement que des indications concernant le nombre de tuples touchés par l'exécution ou concernant le statut de la requête (exécutée, refusée, etc.).

4.4.3. Cas de valeurs nulles :

Il est souvent utile de distinguer une valeur NULL (au sens SQL, il s'agit d'une colonne ne contenant aucune valeur. NULL n'est pas typé) d'une valeur "par défaut" pour un type donné (par exemple la chaîne vide pour des chaînes de caractères, ce qui est différent de NULL).

Par exemple, les méthodes *getXXX( )* de *ResultSet* retournent une valeur correspondant au type XXX, mais si la valeur dans la base SQL est un NULL, il y a conversion vers une valeur acceptable par le type d'objet demandé. Les lignes suivantes résument ces conversions :

- Un "NULL" Java est renvoyé par les méthodes qui retournent un objet (*getString*, *getObject, getDate*, etc.).
- Un "0" est renvoyé par les méthodes numériques *getByte, getShort, getInt*, etc.
- Une valeur "false" est renvoyée par *getBoolean*.

il est important de pouvoir distinguer, notamment pour les valeurs numériques, un NULL d'une valeur réellement stockée dans la base.

Les moteurs de base de données sont en effet généralement conçus pour tirer parti de cette valeur spéciale, par exemple pour l'optimisation des requêtes ou du stockage des données.

JDBC propose la démarche suivante : le programmeur lit la donnée puis teste si ce qu'il vient de lire est en réalité un NULL SQL. Cette méthode est à posteriori la plus efficace pour les pilotes JDBC, mais oblige la lecture de toutes les données.

La méthode *wasNull()* de *ResultSet* permet d'effectuer ce test, elle renvoie "true" si l'on vient de lire un NULL et "false" dans le cas contraire.

La méthode *isNullable()* de la classe *ResultSetMetaData* permet de mettre une valeur NULL dans la colonne d'une table.

Lors de la mise à jour d'une colonne, il est nécessaire de savoir si elle accepte les valeurs NUL. Lors de la lecture d'une colonne, il faut savoir qu'elle peut contenir des valeurs NULL.

L'interface graphique d'une application manipule très souvent des chaînes de caractères, même pour représenter des valeurs numériques ou autres.

On peut alors écrire une classe très simple qui représente une chaîne de caractères avec des méthodes de conversion prenant en compte un éventuel NULL, selon que l'on souhaite une version " affichable " dans l'interface ou "stockable" dans la base avec la meilleure efficacité.

Code source d'une méthode qui traite les valeurs NULL [2]:

```
import java.sql.* ;

public class SQLString {
  //la valeur courante
  String value;
//On conserve le type de colonne
  boolean isnullable;
//Un constructeur qui associe une chaîne avec les caractéristiques d'une colonne
public SQLString(ResultSet rs, int col) {
      value=rs.getString(col);
      isnullable=(rs.getMetaData().isNullable()= =ResultSetMetData.columnNullable); }
//Une méthode pour convertir cette chaîne en valeur SQL toujours correcte
public String toSQL() {
  if(value= =null) {
     if(isnullable) return "NULL";
     else return "";
}
return value; }
//On surcharge toString pour avoir un comportement correct avec les méthodes
//print, ou la concaténation de la chaîne. Il n'y a pas de conversion cela permet
//de voir la valeur réelle de la chaîne pour contrôle.
public String toString(){
 return value; }
}
```

Remarque :

Il très important de noter qu'il faut fermer systématiquement les instances des classes : *Connection, Statement, ResultSet* dés qu'elles deviennent inutiles, à l'aide de la méthode close(). Si on n'est pas en mode de confirmation automatique on risque de perdre les informations si la méthode est appelée avant confirmation.

5. Mise en œuvre de JDBC :

Après avoir décrit les principales fonctionnalités de JDBC ainsi que ses principaux modules en détails (voir la section précédente), nous pensons que la meilleure façon de les assimilés est de les appliqués dans un exemple concret (qui est en réalité, une partie de notre application réelle, c'est à dire gestion des comptes particuliers).

La décomposition de la mise en œuvre en étapes et le suivi de l'évolution par insertion de code Java au fur et à mesure, vous faciliteras sûrement la compréhension de notre étude et vous aidera à y voir claire sur l'**API JDBC**.

Note : Nous ne considérons pas la préparation de la base de données comme une "étape" et nous la supposons qu'elle est bien préparée et correctement configurée (surtout en ce qui concerne le mot de passe).

Dans notre cas il s'agît d'une base de données relationnelle crée avec ACCESS (de Microsoft).

Configuration du pilote ODBC :

La configuration du pilote étant, elle aussi une démarche cruciale pour le bon fonctionnement de l'application, nous avons décidé de lui attribuer ce paragraphe afin de bien expliquer le processus à suivre. (cette étape est externes à celles propres à JDBC).

Une fois la base de donnée bien conçue, nous devons la rendre accessible à notre application Java. Cela se fait en passant par les étapes suivantes:

- Cliquer sur l'icône "ODBC 32bit" dans le panneau de configuration.
- Choisissez l'anglet "DSN utilisateur".
- Sélectionnez un pilote pour lequel vous souhaitez définir une source de donnée (dans notre cas "driver pour Access").
- Une fenêtre de titre "Installation ODBC pour Microsoft Access" apparaît.
- Remplir le champ "Nom de la base" (dans notre cas PORTEFEUILLE).
- Remplir le champ "Description" (optionnelle).
- Cliquez sur le bouton "Sélectionner" pour indiquer le chemin exact ou se trouve votre base de donnée.
- Validez en cliquant sur le bouton "OK".

Une fois ces étapes complétées l'application Java sera apte à interagir avec la source de données.

Maintenant que la base de données est prête et le pilote bien installé nous pouvons entamer l'élaboration de notre application qui implémente l'API JDBC.

5.1. 1ére étape : Etablir la connexion avec la base de données.

D'habitude pour expliquer un mécanisme, on commence par le tout début puis en remonte mais dans notre cas on va procéder inversement.

Une connexion s'obtient (à titre démonstratif) par les lignes suivantes :

```
try {
        Class.forName (driver) ;          (1)
        Connection  connexion=DriverManager.getConnection(url,"Admin", "Password") ;      (2)
catch(Exception e)  {}
```

a- Explication de la première ligne : (Class.forName(driver) ;

Pour enregistrer un pilote, il existe deux moyens :

1- On usant de la propriété système ***sql.drivers*** dont la valeur est une liste de nom de classes qui implémentent l'interface *java.sql.Driver.* L'ensemble des classes est ensuite chargé automatiquement par le *DriverManager* dans l'application.

inconvénient :

- Elle empêche la diffusion par le réseau du fichier de propriété.
- Elle convient plus lorsque l'interface est de type ligne de commande mais pas aux applets et applications dotées d'interfaces utilisateur graphiques

2- Le deuxième moyen et sans doute le plus intéressant, est l'utilisation de la méthode *Class.forName (String)* qui est une méthode de chargement <à la demande>. Autrement dit, elle charge uniquement la classe dans la quelle se trouve le pilote approprier à notre connexion.

Grâce à cette méthode on bénéficie du mécanisme de chargement des classes par le réseau et on utilise que celles nécessaires à notre tâche.

Dans notre cas, le pilote **jdbc-odbc** se trouve dans la classe **JdbcOdbcDriver** du paquetage **sun.jdbc.odbc**. Donc pour charger notre pilote on devra passer comme argument à la méthode *Class.forName(String)* le nom de la classe complet (c'est à dire le chemin jusqu'à elle) et aura la syntaxe suivante:

Class.forName (sun.jdbc.odbc.JdbcOdbcDriver) ;

Remarque :

Vu l'aspect attrayant et les avantages qu'apporte la méthode Class.forName(), notre choix pour le chargement du pilote c'est porté pour elle.

b- Explication de la deuxième ligne :

DriverManager.getConnection(**url**,"**Admin**","**Password**") ;

Remarque : Après la création d'un objet *Connection* on lui affecte la ligne ci-dessus.

Afin d'obtenir une connexion vers une source de donnée on est obliger de faire appel à la méthode *getConnection()* de la classe *DriverManager* qui prend comme arguments l'**url**, **Admin** et **Password** et qu'on va tenter de les expliqués un à un :

L'url :

L'url est celui de notre source de données. La convention de nommage retenu par JDBC est dérivée de celle utilisée pour les URL de l'Internet.

Ainsi on retrouve dans "url" la syntaxe (ou la forme générale) suivante:

jdbc:<sous-protocole>:<complément>

- Le protocole <jdbc> serre à distinguer les URL JDBC des autres.

- Le <sous-protocole> pour distinguer le type de pilote utilisé. Le pilote jdbc-odbc livré par JDBC utilise <odbc>comme nom de sous-protocole.

- Reste la partie <complément> qui est variable quant à son contenu. Elle serre à identifier la source de données. Pour une source de données locale l'**url** aura la **forme** suivante :

  jdbc:odbc:nom_de_base_de_données

  Note : Pour les sources de données distantes il faudra utiliser la notation traditionnelle des URL pour nommer la machine à contacter et elle aura la forme suivante (jdbc-net étant un nom de protocole d'un pilote donné) :

  jdbc:jdbc-net://nom_du_serveur:port/nom_de_la_bd

Admin :

Le deuxième argument de la méthode *getConnection* est : Admin. qui représente le nom de l'administrateur de la base de donnés (qui peut être optionnelle).

Password :

Le dernier argument est celui du mot de passe de la source de données lui aussi peut être optionnelle.

Une fois tous les éléments nécessaires à une connexion ont étés définis, il nous reste plus qu'à présenter les "Exceptions" qui peuvent être levées et la liste est la suivante :

- ClassNotFoundException : levée par la méthode *Class.forName() ;*

- SQLException : levée par *getConnection() ;*

- MalformedURLException : levée par le constructeur de la classe URL (ou cas où on déclarerait l'url comme une instance de la classe URL et non pas comme String).

Remarque *:*

Il ne faut pas oublier d'importer les paquetages :

- java.sql.* ; (le symbole " * " représente toutes les classes contenues dans le paquetage)

- java.net.URL ;

Enfin, l'écriture du code serra la dernière formalité pour complété notre explication :

Code source pour l'établissement d'une connexion :

```
import java.sql.* ;
import java.net.URL ;

public class PincipaleClasse extends java.applet.Applet {

//Déclaration (définition) de l'url qui de préférence doit être déclarer comme une variable
//d'instance de la classe

      String url="jdbc:odbc:PORTEFEUILLE" ;

// Remarque : s'il s'agit d'une applet, les lignes ci-dessous se doivent d'être dans la méthode
// init() de l'applet.(si c'est une application ca serra dans la méthode main()).
// Dans notre cas c'est une applet.

init ( ) {

  try {
        //recherche et chargement de la classe qui contient le pilote

              Class.forName (sun.jdbc.odbc.JdbcOdbcDriver) ;

       //Création d'un objet Connection et ouverture de la connexion

              Connection connexion=DriverManager.getConnection(url, "Admin", " motP");
       }
  //dans les closes "catch" on essaye d'anticiper les exceptions qui pourront  être levées par
  //les différentes méthode participantes à l'obtention  de la connexion

   catch (ClassNotFoundException e) {

         //classe du pilote introuvable
     }

  catch (MalformedURLException e) {

        //dans le cas ou on utilise un constructeur de URL et si l'url donnée ne correspond pas
        //au format  URL. Si url est déclaré comme String il inutile de "catcher" cette exception.
     }

  catch (SQLException e) {

         //cas où l'accès à la base n'a pas pu se faire (url faux)
  }

  catch (Exception e) {
        //Divers autres Exceptions.
  }
 }
}
```

Une fois cet objet connexion (de la classe *Connection*) obtenue, on peut passer à l'étape suivante.

5.2. 2éme étape : Exécution des requêtes :

Une fois la connexion établie, l'application peut interagir avec la source de données et lui transmettre des requêtes et effectuer des mises à jour. Pour cela il y a ce qu'on appelle des requêtes statiques et autres dynamiques (précompilées, procédures stockées) et on va tenter de les expliquées tous les deux, cependant on commencera par la plus simple.

5.2.1. Requête simple :

a- L'objet *Statement* :

Afin de pouvoir effectuer une requête on doit avoir un objet ***Statement*** qu'on peut obtenir au pré d'un objet *Connection* grâce à la méthode ***createStatement()*** de la classe *Connection*. Ainsi une seule ligne suffirait pour le faire et elle aura la syntaxe suivante :

```
// On suppose la connexion établie

    Statement statement = connexion.createStatement() ;

//connexion étant un objet Connection
```

Créer un objet *Statement* ne suffit pas, il faut lui associer des méthodes qui pourront réaliser les requêtes sur la source de données associée. Parmi ses méthodes, la méthode ***executeQuery()***, qui prend comme argument une chaîne de caractères qui représente l'ordre SQL à exécuter et le résultat (en sortie) de la méthode serra un objet *ResultSet* qu'on définira plus tard. Quelques lignes de codes s'imposent :

```
//statement est un objet Statement

    ResultSet rs = statement.executeQuery (" SELECT * FROM ……") ;
```

Remarques :

- Le résultat de la requête n'est accessible qu'à travers l'objet *ResultSet*
- La syntaxe exacte de la requête dépend du SGBD et non pas de Java
- On cas d'échec une *SQLException* est levée.

b- L'objet *ResultSet* :

L'objet *ResultSet* permet l'accès aux résultats de la requête obtenus par la méthode *executeQuery()*.

Après l'obtention de celui-ci (*ResultSet*), il est nécessaire de lire manuellement les données et de les stockées dans des variables afin de les utilisées ou pour les traitées et c'est là, l'un des inconvénients de la version 1.0 de JDBC (qui ne transfère en mémoire que les données demandées) et qui a été corrigé dans la version 2.0, c'est à dire qu'on peut naviguer comme on veut dans les résultats.

L'accès aux tuples se fait à l'aide de la méthode ***next()*** (décrite dans la section précédente) qui renvois une valeur "false" si on arrive au dernier tuple.

Remarques :

- La méthode *next()* adopte la technique des curseurs, c'est à dire qu'il faut d'abord pointer le curseur sur le premier enregistrement pour pouvoir lire son contenu, il faut donc l'exécuter au moins une fois.
- On accède au champ voulu dans le tuple grâce aux méthodes ***getXXX()*** qui prennent comme arguments selon notre choix :
  - Une chaîne de caractère représentant le nom du champ (dans la BD).
  - Un entier représentant le numéro de la colonne dans la table correspondante.

On doit pouvoir vérifier le bon fonctionnement de ce qui a été décrit et expliquer dans cette étape en exécutant le code suivant :

Code source pour effectuer une requête :

```
import java.sql.* ;

import java.net.URL ;

public class PincipaleClasse extends java.applet.Applet {


      String url="jdbc:odbc:PORTEFEUILLE" ;

init ( ) {

  try {

        //recherche et chargement de la classe qui contient le pilote

              Class.forName (sun.jdbc.odbc.JdbcOdbcDriver) ;

       //Création d'un objet Connection et ouverture de la connexion

              Connection connexion=DriverManager.getConnection(url, "Admin", " motP") ;

       //Création d'un objet Statement

              Statement  statement = connexion.createStatement() ;

      //Exécution d'une requête avec la méthode executeQury()
```

```
        ResultSet rs = statement.executeQuery (" SELECT NumCompte, DateS, BalE FROMCompte ") ;

    // Le résultat de la requête est dans l'objet ResultSet (dans la variable rs)
    //pour parcourir les tuples on utilise la méthode next() associer à une boucle while

        while (rs.next()) {

            int numéroC = rs.getInt (1) ;
            //on peut désigner le champ par son nom : rs.getInt("NumCompte")

             Date dateSign = rs.getDate (2) ;
            float  balance = rs.getFloat (3) ;

            System.out.println (numéroC+dateSign+balance) ;   }    //pour afficher les résultats.
    }

  catch (ClassNotFoundException e) {

        //classe du pilote introuvable
    }

  catch (MalformedURLException e) {}

  catch (SQLException e) {

        //cas où l'accès à la base n'a pas pu se faire (url faux)
  }

  catch (Exception e) {
        //Diverses autres Exceptions.
  }
 }
}
```

5.2.2. Requêtes de mise à jour :

UPDATE, INSERT et DELETE (dans une expression SQL) sont les commandes nécessaires à une mise à jour. Mais avec JDBC l'objet retourné, dans ce cas, est un entier représentant le nombre de tuples affectés par la mise à jour et non pas un enregistrement.

La méthode destinée à effectuer les mise à jour est : ***executeUpdate( )*** de la classe *Statement.* Elle aura comme argument l'ordre SQL (une chaîne de caractères).

Le code source relatif à une requête de mise à jour :

```
//On suppose que la connexion et que l'objet Statement sont bien préparés.

try {
        int nbrTuples = statement.executeUpdate ("DELETE FROM Compte WHERE NumCompte =  10") ;

catch (Exception e) { }
```

5.2.3. Utilisation d'une requête précompilées : l'objet *PreparedStatement*

L'utilisation de requêtes précompilées est très intéressante, surtout si une opération se doit d'être répétée plusieurs fois.

Par exemple, dans notre cas, l'insertion d'un nouveau client : on n'est pas obligé d'écrire une requête pour chaque client, il suffit d'utiliser la même requête et cela en remplaçant les valeurs des attributs par des variables, aux quelles on leurs affecterais à chaque fois les valeurs des champs et donc on aura une seule requête qu'on utilisera autant de fois qu'il est nécessaire.

La classe qui offre cette possibilité est nommée ***PreparedStatement*** qui est une extension de *Statement*

On obtient un objet de cette classe grâce à la méthode ***prepareStatement( )*** de la classe *Connection* et qui renvoie un objet *PreparedStatement*.

La classe PreparedStatement a les mêmes méthodes que la classe Statement (c'est à dire *executeQuery ()* et *executeUpdate ()*) sauf n'ont pas d'arguments (c'est à dire une chaîne de caractères qui représente l'ordre SQL).

Le mieux serait de donner un exemple :

```
PreparedStatement preparer = connexion.prepareStatement("DELETE FROM" +
                    "  Compte WHERE NumCompte =  ? ") ;
```

Signification du symbole " ? ":

La requête décrite ci-dessous efface, à chaque fois qu'elle est appelée, l'enregistrement dont le numéro de compte correspond à la valeur qui est attribuer à la variable " ? ".

La fixation de la valeur du symbole " ? " se fait par les méthodes de la forme :

setXXX (int, XXX).

- Le premier argument de cette méthode est un entier représentant l'ordre du paramètre dans la commande SQL à modifier (qui commence par 1).
- Le deuxième une instance d'un objet du type de retour attendu.

Voici maintenant l'emploi d'une requête précompilée dans un exemple concret (que nous avons utiliser pour notre application afin d'optimiser l'accès au SGBD) .

Code source démontrant l'utilisation d'une requête précompilée :

```
import java.sql.* ;
import java.net.URL ;

public class PincipaleClasse extends java.applet.Applet {

                String url="jdbc:odbc:PORTEFEUILLE" ;
init ( ) {

  try {
            Class.forName (sun.jdbc.odbc.JdbcOdbcDriver) ;

            Connection connexion=DriverManager.getConnection(url, "Admin", " motP");

        //préparation de l'objet PreparedStatement par la méthode prepareStatement()

          PreparedStatement psclient=bm.dbConnection.prepareStatement("INSERT INTO Client"
                        +  " (NomCli,PrénCli,AdrCli,DateNCli,ProfCli,TélCli) VALUES(?,?,?,?,?,?)");

     //Remarque : dans notre cas : l'opération d'insertion (càd la fixation des paramètres) se
     //fait dans une procédure (qui à comme arguments les informations prises dans les
     //champs de texte que l'opérateur aura rempli soigneusement) et ses lignes ne sont qu'à
     //titre démonstratif et ne correspondent pas à la logique que nous avons suivie, c'est
     //juste pour ne pas compliquer les choses.

     //Fixation des paramètres (càd attribution de valeurs aux  " ? "

          psclient.setString(1, textfieldNom.getText());
          psclient.setString(2, textfieldPrenom.getText());
          psclient.setString(3, textfieldAdr.getText());
          psclient.setString(4, textfieldDateN.getText());
          psclient.setString(5, textfieldProf.getText());
          psclient.setString(6, textfieldTel.getText());

          int row1=psclient.executeUpdate();

catch (ClassNotFoundException e) {  //classe du pilote introuvable}

catch (MalformedURLException e) {

   //dans le cas ou on utilise un constructeur de URL et si l'url donnée ne correspond pas
   //au format  URL. Si url est déclaré comme String il est inutile de "catcher" cette exception.
     }
  catch (SQLException e) { //cas où l'accès à la base n'a pas pu se faire (url faux) }

  catch (Exception e) {

        //Diverses autres Exceptions.
  }
 }
}
```

Remarque :

Cette méthode permet d'améliorer les performances : le driver JDBC n'est pas obliger d'effectuer un accès supplémentaire pour découvrir les types des colonnes impliquées dans la requête.

5.2.4. Utilisation de procédures stockées : l'objet *CallableStatement*

L'utilisation de procédures stockées est similaire à celle des procédures précompilées.

JDBC vous permet d'appeler une procédure stockée d'une base de données depuis une application écrite en langage Java.

La première étape consiste à créer un objet ***CallableStatement.*** Comme avec les objets *Statement* et *PreparedStatement*, cela est fait avec l'objet *Connection*. Un objet *CallableStatement* contient un appel vers une procédure stockée, il ne contient pas la procédure stockée en elle-même.

```
CallableStatement cs = connexion.prepareCall("{call FICHE_DE_COMPTE}");

ResultSet rs = cs.executeQuery() ;
```

La première ligne de code ci-dessous crée un appel à la procédure stockée FICHE_DE_COMPTE en utilisant la connexion "connexion".

La partie qui est entre accolades est la syntaxe d'échappement pour les procédures stockées. Lorsque le driver rencontre "{call FICHE_DE_COMPTE }" il va traduire cette syntaxe d'échappement par l'expression SQL native utilisée dans la base de données pour appeler la procédure stockée appelée FICHE_DE_COMPTE.

La classe *CallableStatement* est une sous-classe de *PreparedStatement*, donc un objet *CallableStatement* peut prendre des paramètres d'entrée tout comme un objet et peut aussi recevoir les paramètres de retour éventuels de la procédure stockée. Dans ce cas la syntaxe prendra la forme suivante :

```
CallabelStatement cs=connexion.prepareCall ("{?= call Extrait_de_Compte (?,?)}") ;
```

On utilise la méthode ***registerOutParameter (int, Type)*** pour indiquer que le paramètre de sortie (deuxième argument de la méthode), dont la position est déterminer par le premier argument de la méthode, est du type *java.sql.Type*

Remarque :

Il y a une seconde forme de la méthode *registerOutParameter (int, Type* , int), ou le troisième argument représente le nombre de chiffres après la virgule si jamais le type de retour était "DECIMAL" ou "NUMERIC".

Une fois ces paramètres de retour enregistrés, on appelle la méthode ***executeQuery()*** (requête simple) ou ***executeUpdate()*** (pour une mise à jour), et on lit les paramètres avec les méthodes *getXXX(int)* appropriées.

Cependant, parfois on trouve le cas ou une procédure stockée contient plus d'une expression SQL, dans ce cas, l'expression va produire plus d'un ensemble de résultats, plus d'un compteur de mise à jour, ou une combinaison d'ensembles de résultats et de compteurs de mise à jour.

Dans ce cas, où il y a des résultats multiples, la méthode ***execute ()*** devrait être utilisée pour exécuter l'objet *CallableStatement.*

L'exemple suivant utilise une procédure stockée (nommée Extrait_de_compte) dans la source de données qui en lui passant un numéro de fiche et une date nous renvoie le solde (à titre d'exemple) :

```
//On suppose la connexion établie

try {

//préparation de l'objet CallabelStatement avec la méthode prepareCall()

        CallabelStatement cs=connexion.prepareCall("{? = call Extrait_de_compte (?,?)}") ;

//On enregistre le premier paramètre comme valeur de retour

        cs.registreOutParameter (1, java.sql.FLOAT) ;

//On fixe la valeur du premier paramètre en entrée c'est à dire le numéro de compte

        cs.setInt (2, "1500") ;    //le chiffre "2" représente l'emplacement du paramètre

         cs.setDate(3, 12/03/2000 ) ;

//On a plus qu'à exécuter la procédure stockée

        ResultSet  rs = cs.executeQuery () ;  //il nous reste plus qu'a extraire le résultat
```

5.3. 3éme étape : Libération des ressources

L'établissement d'une *Connection* et par la suite d'un objet *Statement* et *ResultSet* sont les seuls moyens afin d'interagir avec la source de données, mais il ne faut pas oublier qu'elles sont consommatrices de ressources système et mémoire. Il est donc souhaitable pour ne pas dire impératif de libérer les ressources si on vient à quitter l'application ou quant on a plus à s'en servir. Cela se fait par la méthode ***close()***.

Chacun des objets *Connection, Statement* et *ResultSet* arrivé à sa fin d'utilisation, doit appeler la méthode ***close()***.

Ces objets sont en principe automatiquement supprimés de la mémoire par le "ramasse-miettes" (ou le *Garbage-Collector*). Mais il est préférable de le faire soi-même.

Remarque : Certains pilotes n'acceptent pas plus d'une connexion à la fois.

L'exemple qui va suivre est le parfait exemple d'une application utilisant JDBC. C'est à dire qu'elle :

- Etablit une connexion
- Effectue une requête
- Récupère les résultats
- **Achève la connexion.**

*Code source (surtout, pour montrer comment utiliser la méthode **close()**) :*

```
import java.sql.* ;

import java.net.URL ;

public class PincipaleClasse extends java.applet.Applet {

                String url="jdbc:odbc:PORTEFEUILLE" ;
init ( ) {

      try {

          Class.forName (sun.jdbc.odbc.JdbcOdbcDriver) ;

          Connection connexion=DriverManager.getConnection(url, "Admin", " motP") ;

          Statement  statement = connexion.createStatement() ;

          ResultSet rs = statement.executeQuery (" SELECT NumCompte, DateS, BalE FROMCompte ") ;

          while (rs.next()) {

           int numéroC = rs.getInt (1) ;  //le "1" dans la méthode getInt() représente le numéro de
           Date dateSign = rs.getDate (2) ;   // la colonne dans la base de données
           float  balance = rs.getFloat (3) ;

           System.out.println (numéroC+dateSign+balance) ;     }    //pour afficher les résultats.

              //Avant de sortir :

              //On ferme l'objet  connexion (de la classe Connection)

                  connexion.close() ;

             //On ferme l'objet statement (de la classe Statement )

                  statement.close() ;

             //On ferme l'objet rs (de la classe ResultSet )

                  rs.close() ;
      }
```

```
 catch (ClassNotFoundException e) {}

  catch (MalformedURLException e) {}

  catch (SQLException e) {}

  catch (Exception e) {}

 }
}
```

L'explication des différentes étapes, a peut-être, été longue mais nous pensons que nous avons aborder tous les aspects et tous les mécanismes qu'une application, utilisant JDBC, pourrait avoir à utiliser (dans la plupart des situations).

Il faut noter que l'apprentissage de quelques techniques supplémentaires est requis dans le cas de certaines applications avancées.

Bien sûr, notre application (applet) ne se limite pas à ce qui a été décrit ci-dessus mais elle a à peut près la même structure (squelette).

6. Sécurité :

Dans le monde de l'Internet, le mot "sécurité" occupe une place importante, que se sois pour les développeurs de logiciels ou pour les utilisateurs mais cela n'a pas échappé aux concepteurs de JDBC qui se sont pencher sur le problème et l'on abordé à travers les cas suivants :

6.1. Sécurité des Applets :

Les règles de sécurités qui ont été imposer au applets n'utilisant pas JDBC sont les mêmes pour ceux qui l'utilisent c'est à dire qu'elles ne peuvent ni lire ni écrire sur la machine du client (ni accéder aux données locales), elles nous pourront établir de connexion qu'avec des bases de données situées sur les machines d'où elles proviennent.

6.2. Sécurité des Drivers (pilotes) :

Le driver est le maillon sensible de la chaîne de connexion entre le client et le serveur et les concepteurs de JDBC recommande fortement aux développeurs de drivers de protéger leur code contre les utilisations abusives.

Dans le cas d'une connexion simple (non partagée), les contrôles effectués par le *SecurityManager* suffise à garantir la sécurité d'une connexion mais dans le cas où un driver désir partagé une connexion TCP (*Transfer Control Protocol*) entre plusieurs *Connection JDBC* (cas d'un accès concurrent). Dans ce cas, c'est au driver de vérifier que les différents clients ont bien le droit d'accéder à la source de données définie dans la connexion TCP partagée.

Remarque : Les méthodes d'un driver doivent supporter des accès concurrents.

7. Conclusion :

Tout au long de ce chapitre nous avons essayé d'expliquer les différents mécanismes manipulés par l'API JDBC et sa capacité d'interagir avec des sources de données (locales ou distantes).

Il est claire que cette API ouvre la porte à beaucoup de perspectives aux développeurs d'applications de nature distribuées (Intranet, Internet). Néanmoins, rappelons que l'API JDBC n'ont n'est qu'à ses débuts (première version), c'est pourquoi, il ne faut pas s'attendre à ce qu'elle puisse ménager vos efforts de programmations ; en effet l'API JDBC reste relativement complexe à manipulée et ses méthodes assez archaïques.

Tout cela devra être compenser par une programmation rigoureuse de la part des développeurs en attendant les améliorations futures qui pourront être apportées par des versions à venir.

Au fil de ce chapitre nous vous avons éclairée sur la puissance de l'api JDBC qui ne sort pas du concept JAVA c'est à dire la facilité et la portabilité de toutes ses classes, en gardant une efficacité exorbitante en ce qui concerne l'accès aux bases de données locales ou distantes cette efficacité se résume dans différentes classes conçues d'une manière intelligente tout on respectant les différents standards et protocoles de transmission distante de données (TCP/IP[1], HTTP[2],…).

Et comme tout autre langage il a aussi des faiblesses paralysantes tel que la difficulté dans la conversion de types java/sql et l'inverse qui demande une bonne conception de la part du programmeur.

(1) TCP/IP : *Transfer Control Protocol / Internet Protocol*.
(2) HTTP : *Hyper Text Transfer Protocol.*

1. Introduction :

Après avoir bien étudié l'API JDBC et démontré sa capacité d'interaction avec les Bases de Données (BD) distantes, nous arrivons au stade de l'application ; celle-ci ne sera pas réalisable sans une BD concrète.

Plutôt que de prendre une (BD) déjà existante, nous avons jugé préférable de la concevoir nous même afin de bien s'assurer de l'intégrité et de la cohérence des données et par la même occasion enrichir nos connaissances avec l'expérience que pourra nous apporter la conception d'un Système d'Information.

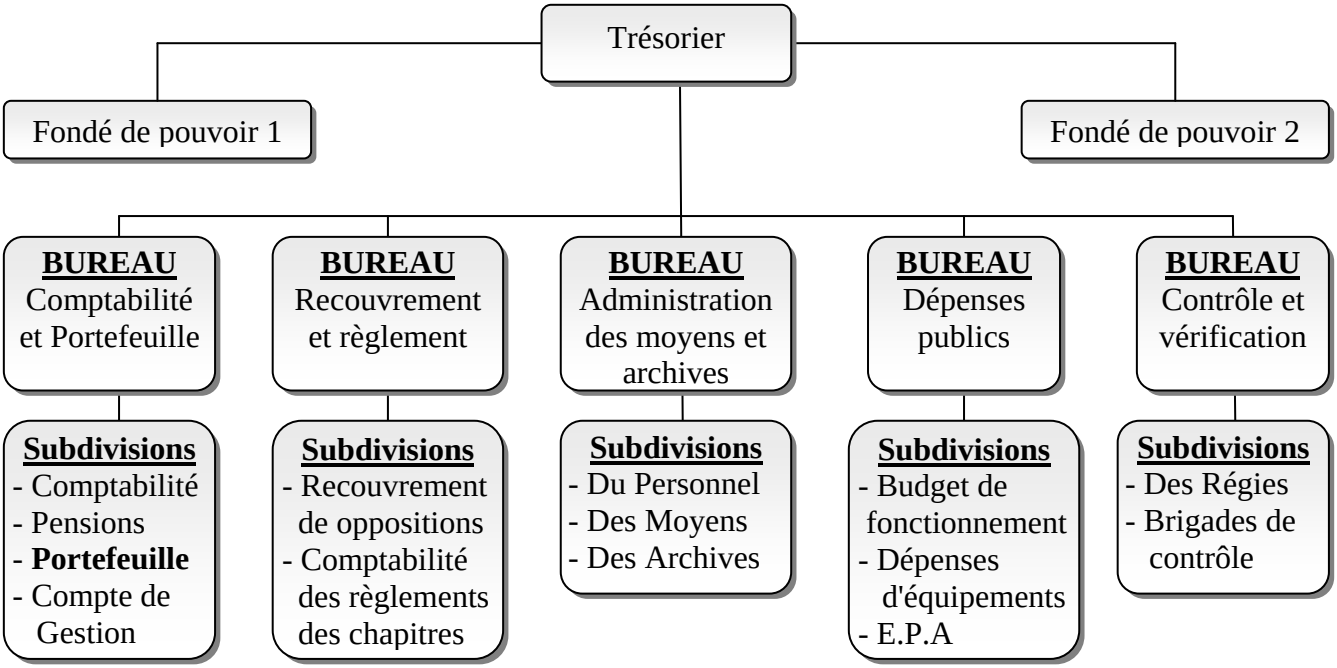
Vu l'engouement et l'importance que prennent les transactions financières dans le monde du WWW, la Gestion du Portefeuille (compte des fonds particuliers) du Trésor Public de Constantine nous semblait être le sujet idéal et le mieux adapté à notre application tout en pensant aux avantages qu'elle pourra apportée aux utilisateurs (consultation du solde, extrait de compte…etc).

Toutefois, vu les besoins et les restrictions imposées par le manque de temps, nous avons choisi de ne pas trop approfondir les concepts sans pour autant passer à coté d'une bonne conception et une étude bien fondée.

2. Présentation de l'organisme d'accueil

Le Trésor Public est un service de l'état, ayant pour mission d'effectuer des opérations financières conformément aux dispositions des lois de finance pour le compte de l'état et des organismes publics. Il représente aussi l'état en tant que banquier et à ce titre il bénéficie d'une indépendance dans les actions à entreprendre.A ce titre il collecte les capitaux, reçoit des dépôts, fait des emprunts, prend des participations, donne des garanties et veille à la régularisation de la circulation monétaire.

2.1. Organigramme structurel :



Notre étude ce limite à la gestion du service **PORTEFEUILLE(Gestion des Comptes Particuliers)**.

2.2. Présentation de la Subdivision du Portefeuille :

Il revêt une importance considérable au sein de la trésorerie. Sa mission consiste à gérer les compte aussi bien des particuliers que des Etablissements Publics Administratifs (E.P.A). Il enregistre toutes les opérations effectuées sur le fond (débit ou crédit).

La subdivision du Portefeuille gère les comptes suivants :

a- Les comptes de fonds Particuliers :

Ils sont ouverts à la demande des particuliers jouissants de leurs droits civiques.
ex : Fonctionnaires, pour le virement de leurs traitements.

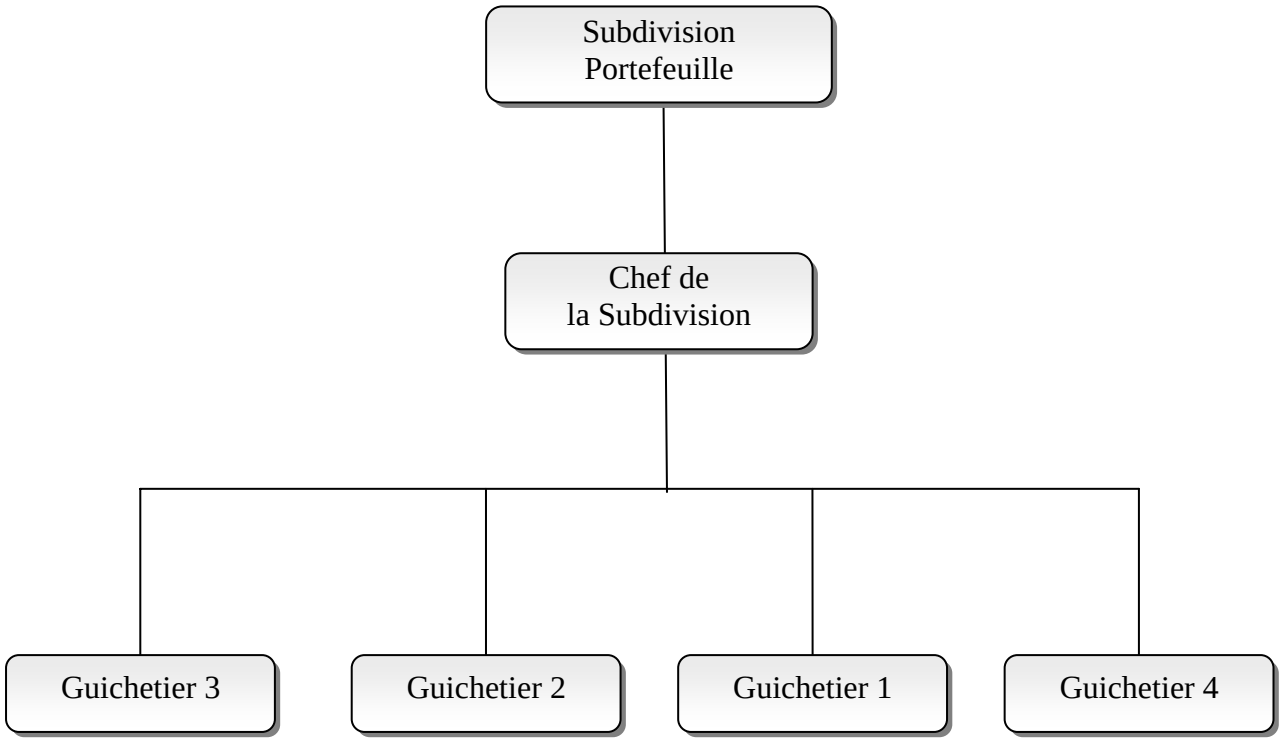
b- Les comptes de dépôts de fonds :

Ils sont ouverts obligatoirement aux organismes publics ou subventionnés par l'état:

Compte avec intérêts : C.N.E.P.

Comptes sans intérêts : caisse de sécurité sociale - lycée - banques - associations diverses

2.3. Structure de la subdivision du Portefeuille :



3. Exposition du problème :

Etant donné que le Portefeuille relève d'une grande importance et que les fonds ne cessent de croître en nombre, révélant ainsi une grande difficulté dans la gestion, l'informatisation du service Portefeuille s'impose.

En se basant sur les enquêtes et les interviews effectuées auprès des responsables du service, qui n'ont pas hésiter à nous exposer les états des faits et les problèmes auxquels ils se heurtent couramment, il est apparu un certain nombre de points :

- Une grande charge dans le travail du service du Portefeuille due au volume important des opérations.
- Une lenteur dans les calculs et le contrôle des opérations ce qui cause un retard considérable pour l'envoi des situation comptables (Extrait de Compte, Balance…etc.)

3.1. Résultats attendus :

Les résultats attendus par la mise en place du nouveau Système d'Information se traduit comme suit :

- Permettre une disponibilité de l'information à tout moment.
- Supprimer la redondance des informations.
- Permettre d'établir la Feuille de Journée, Balance, Extrait de Compte…etc.
- Automatiser toutes les procédures de calcul et de contrôle afin d'avoir une bonne fiabilité et facilité
  la tache des agents et éliminer les retards.

3.2. Choix de la Méthode :

Pour la réalisation de notre Système d'Information, notre choix s'est porté sur la Méthode de conception **MERISE** [3,5,6] qui utilise le formalisme Entité/Association et qui, par sa capacité, permet de traduire une vue globale de l'entreprise de façon à lier la mise en place d'un système informatisé de gestion à une refonte de l'organisation.

4. Règles de gestion :

Les règles de gestion se présentent comme suit :

- Un client ne peut avoir qu'un seul compte.
- Un compte est désigné par un Numéro de Fiche (unique), une Date de Signature et un Etat (considéré comme "Actif "à son ouverture) et les informations concernant le Client : Nom, Prénom, Date de Naissance, Lieu de Naissance, Adresse, Profession.
- Une fois le compte soldé, une mention "Inactif" apparaît sur la fiche.
- Un Débit s'effectue par un Chèque qui possède un Numéro (unique), un Montant, le Nom et Prénom du Débiteur et une Date.
- Un chèque ne désigne qu'un seul numéro de fiche.
- A chaque opération de crédit est attribuée un Numéro (unique et séquentiel), un Montant, le Nom et le Prénom du verseur, la Nature de l'opération et une Date.
- Une opération de Crédit ne concerne qu'un seul numéro de fiche.

5. Le Modèle Conceptuel des Données :

5.1. Dictionnaire de Données :

N.B: Nous voulons dire par la désignation "Fiche" : un Compte.

CODE	SIGNIFICATION	TYPE	LONG	Nature		REGLES
				GM SIGM	CA COE	
NumCompte	Numéro de fiche	N	4	Sig	E	/
NomCli	Nom Client	A	25	Sig	E	/
PrénCli	Prénom Client	A	25	Sig	E	/
DateNCli	Date de naissance client	AN	10	Sig	E	JJ/MM/AAAA
LieuNCli	Lieu de naissance client	A	50	Sig	E	/
AdrCli	Adresse Client	AN	70	Sig	E	/
ProfCli	Profession client	A	50	Sig	E	/
DateSig	Date de signature (d'ouverture du compte)	AN	10	Sig	E	JJ/MM/AAAA
EtatC	Etat du Compte (actif ou inactif)	N	1	Situ	E	(0=inactif, 1=actif)
NumChèq	Numéro chèque	N	6	Sig	E	/
MontantD	Montant Débit	N	12	Sig	E	/
DateChèq	Date Chèque	AN	10	Sig	E	JJ/MM/AAAA
NumOp	Numéro d'Opération	N	6	Sig	E	999999/AA
NatOp	Nature de l'Opération	N	1	Situ	E	/
MontantC	Montant Crédit	N	12	Sig	E	/
DateOp	Date de l'Opération	AN	10	Sig	E	JJ/MM/AAAA
SoldeC	Solde Créditeur	N	12	Sig	CA	Etant donné
BalE	Balance d'Entrée	N	12	M	CA	
N-S	Nouveau Solde	N	12	Sig	CA	le volume
TJD	Total des Journées Débit pour chaque fiche	N	12	Sig	CA	
TJC	Total des Journées Crédit pour chaque fiche	N	12	Sig	CA	important des
T-D-J	Total Débit de la Journée	N	12	Sig	CA	
T-C-J	Total Crédit de la journée	N	12	Sig	CA	règles de
TJD_m	Total des Journées Débit Mensuel par fiche	N	12	Sig	CA	
TJC_m	Total des Journées Crédit Mensuel par fiche	N	12	Sig	CA	calcul, ils ont
T-G-D-M	Total Général Débit Mensuel (pour toutes les fiches)	N	12	Sig	CA	été décrites
T-G-C-M	Total Général Crédit Mensuel(pour toutes les fiches)	N	12	Sig	CA	à part dans
T-G-S-M	Total Général Solde Mensuel(pour toutes les fiches)	N	12	Sig	CA	les pages qui
T-D-A-F	Total Débit Annuel par Fiche	N	12	Sig	CA	
T-C-A-F	Total Crédit Annuel par Fiche	N	12	Sig	CA	vont suivre
INV-F	Solde annuel par Fiche	N	12	Sig	CA	

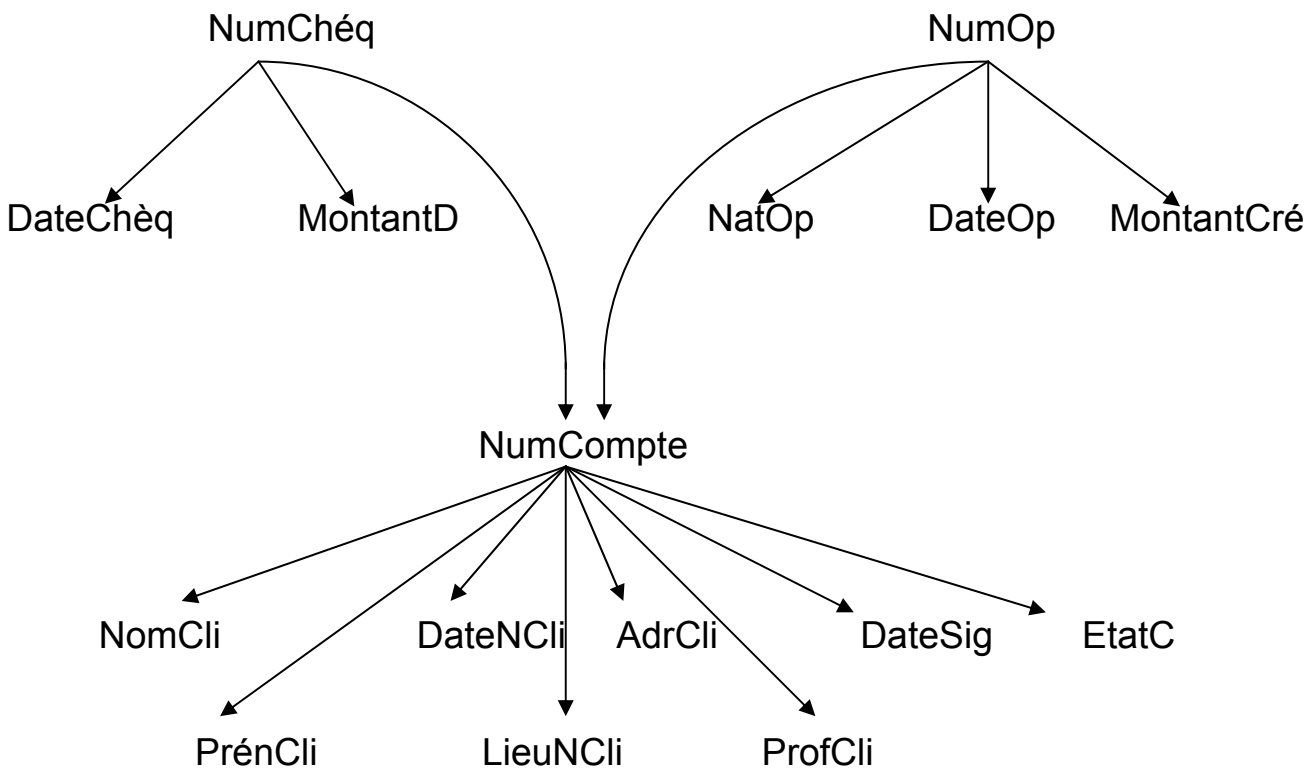
5.2. Dépendances Fonctionnelles :

NumCompte → NomCli, PrénCli, DateNCli, AdrCli, ProfCli, LieuNCli, DateSig, BalE, EtatC.

NumChèq → DateChèq, MontantD, NumCompte.

NumOp → NatOp, MontantCré, DateOp, NumCompte.

5.3. Le Graphe des Dépendances Fonctionnelles :



5.4. Règles de calcul et d'intégrité :

- BalE (Balance d'Entrée) : c'est le Solde avec lequel le client entame la nouvelle année.
- TJD : Total des Débits effectués à partir du début du mois jusqu'à la date courante par fiche :

$$TJD = \sum_{I=1}^{\text{La date courante}} MontantD$$

- TJC:Total des crédits effectués à partir du début du mois jusqu'à la date courante par fiche

$$TJC = \sum_{I=1}^{\text{La date courante}} MontantCré$$

- N-S : Le nouveau solde après une opération de débit ou de crédit :

$$N\text{-}S \begin{cases} = SoldeC + MontantCré \\ \text{ou} \\ = SoldeC - MontantD \end{cases}$$

- SoldeC : Le solde a une date particulière (solde créditeur) : SoldeC = BalE + (TJC-TJD)
- TJD_m : C'est le TJD à la fin du Mois courant (pour chaque fiche).
- TJC_m: C'est le TJC à la fin du Mois courant (pour chaque fiche).
- T-G-D-M : Total Général des Débits Mensuel pour toutes les fiches :

$$T\text{-}G\text{-}D\text{-}M = \sum_{NumCompte=1}^{n} TJD_m.$$

- T-G-C-M : Total Général des Crédits Mensuel pour toutes les fiches :

$$T\text{-}G\text{-}C\text{-}M = \sum_{NumCompte=1}^{n} TJC_m$$

- T-G-S-M : Total Général des Soldes Mensuel pour toutes les fiches :

$$T\text{-}G\text{-}S\text{-}M = T\text{-}G\text{-}C\text{-}M - T\text{-}G\text{-}D\text{-}M$$

- T-D-J : Total des montants Débit pour toutes les fiches à une date donnée :

$$T\text{-}D\text{-}J = \sum_{NbrChéq=1}^{n} MontantD \text{ (a une date D)}$$

- T-C-J : Total des montants Crédit pour toutes les fiches à une date donnée :

$$T\text{-}C\text{-}J = \sum_{NumOp=1}^{n} MontantCré \text{ (a une date D)}$$

- TDAF = Total des Débits Annuel pour une seule Fiche :

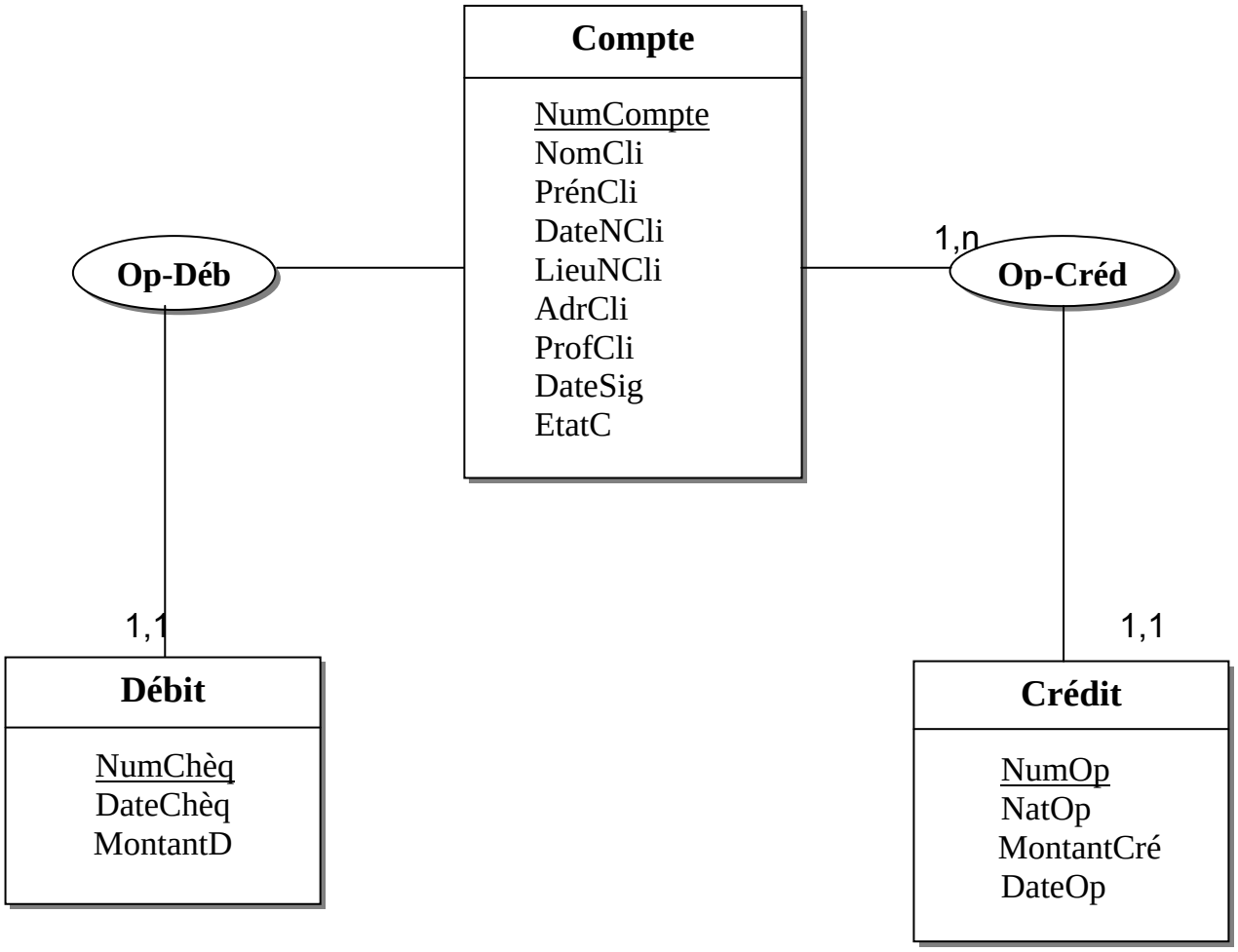
$$T\text{-}D\text{-}A\text{-}F = \sum_{M=1}^{12} TJD_m$$

- T-C-A-F = Total des Crédits Annuel pour une seule Fiche :

$$T\text{-}D\text{-}A\text{-}F = \sum_{M=1}^{12} TJC_m$$

- INV-F : Solde annuel pour une seule Fiche : INV-F = T-C-AF - T-D-AF

5.5. Construction du MCD Relatif à la Gestion du Portefeuille (Comptes Particuliers) :



6. Modèle Conceptuel des Traitements :

6.1. Description des taches et Diagramme de flux:

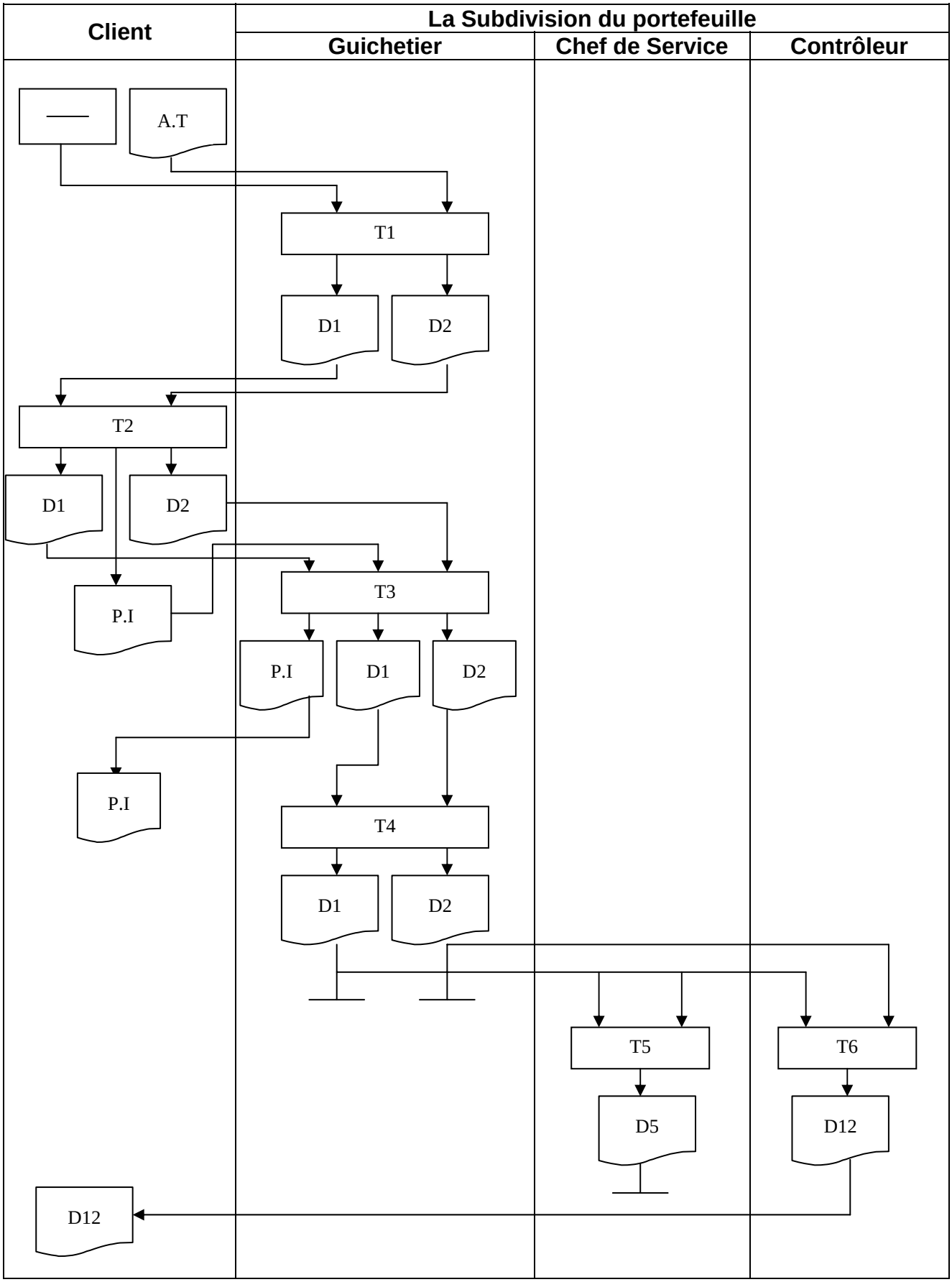
Processus 1 : Ouverture d'un nouveau compte

N° Tache	Libellé	Poste de Travail	Documents Entrants	Documents Sortants
T1	Délivrance des documents demandés	Guichetier du portefeuille	Demande écrite + A.T	D1.D2
T2	Remplissage des documents	Client	D1.D2	D1 Remplie D2 Remplie
T3	Contrôle des documents	Guichetier	D1 Remplie D2 Remplie + P.I	D1 Contrôlée D2 Contrôlée +P.I
T4	Classement des documents	Guichetier	D1 Contrôlée D2 Contrôlée	D1 Classée D2 Classée
T5	Etablissement de la feuille du compte	Chef de service	D1.D2	D5 Etabli et classé
T6	Confection d'un Carnet de Chèques	Contrôleur	D1.D2	D12 Etabli

Légendes :

A.T: Attestation de Travail.
D1 : Fiche d'ouverture d'un compte.
D2 : Demande de Carnet de Chèques.
P.I : Pièce d'Identité.
D5 : Feuillet de Compte.
D12 : Chèque.

Digramme1 : **L'ouverture d'un nouveau compte**



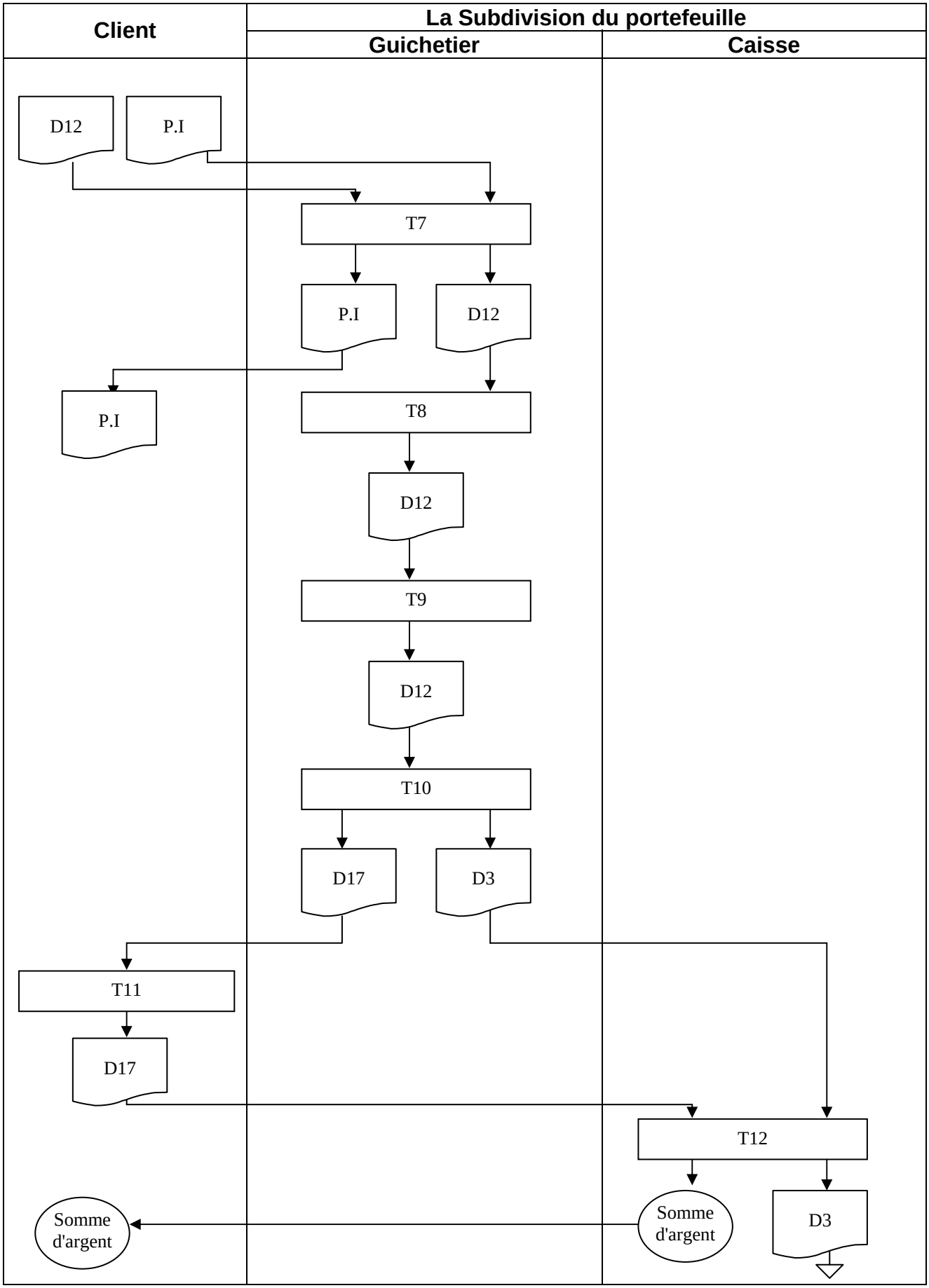
Procédure 2 : **Paiement du client**

N° Tache	Libellé	Poste de travail	Documents Entrants	Documents Sortants
T7	Vérification de l'identité	Guichetier	D12 PI	D12 PI Vérifiée
T8	Vérification de la conformité du chèque	Guichetier	D12	D12 Conforme
T9	Vérification du solde du client	Guichetier	D12	D12 Vérifié
T10	Etablissement de la Fiche de Paiement qui est remise à la caisse et remise du jeton au client	Guichetier	D12	D3 D17
T11	Remise du jeton a la caisse	Client	D17	D17
T12	Paiement du client	Caissier	D3 D17	Somme d'argent D3 Archivée

Légendes :

D12 : Chèque
PI : Pièce d'identité
D3 : Fiche de paiement
D 17 : Jeton

Diagramme 2 : **Paiement du client**


Client
La Subdivision du portefeuille
Guichetier
Caisse
D12
P.I
T7
P.I
D12
P.I
T8
D12
T9
D12
T10
D17
D3
T11
D17
T12
Somme d'argent
Somme d'argent
D3

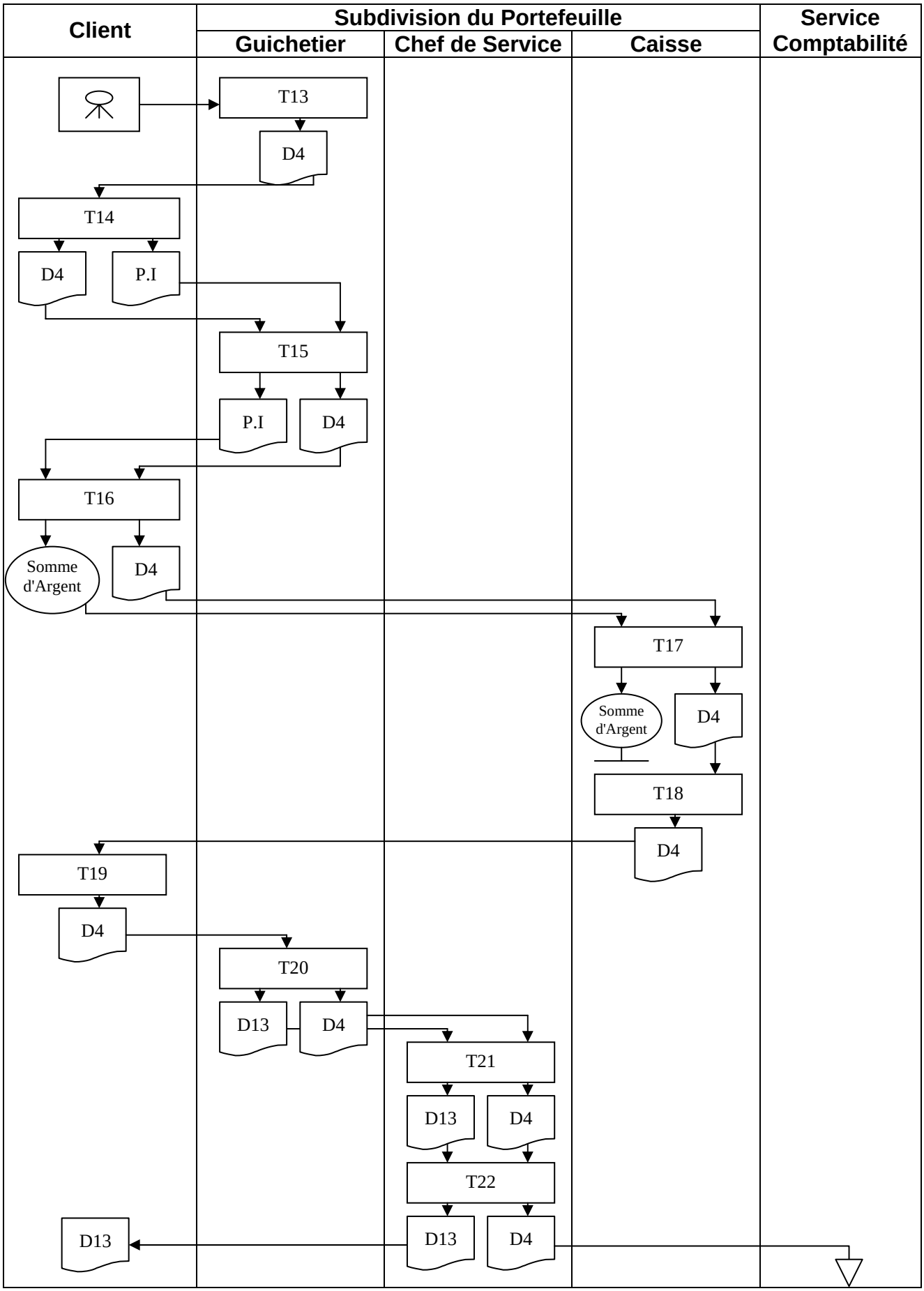
Procédure 3 : Versement

N° Tache	Libellé	Poste de Travail	Documents Entrants	Documents Sortants
T13	Remise de la Fiche de Versement	Guichetier	Demande de D4	D4
T14	Remplissage de la Fiche de versement	Client	D4	D4 Remplie
T15	Contrôle de la Fiche de Versement	Guichetier	D4 + P.I	D4 Contrôlée P.I
T16	Envoi de la Fiche de Versement à la caisse	Client	D4	D4
T17	Contrôle de la Somme d'Argent	Caissier	D4 S.A	D4 S.A Contrôlée
T18	Visa de la Fiche de Versement	Caissier	D4 S.A	D4 Signée
T19	Remise de la fiche de versement au guichetier	Client	D4 Signée	D4 Signée
T20	Etablissement du Récépissé Versement	guichetier	D4 Signée	D4 Signée D13
T21	Signature du Récépissé de Versement	Chef de Service	D4 Signée D13	D4 Signée D13 Signé
T22	Remise du Récépissé de Versement au client et la Fiche de Versement au service Comptabilité	Guichetier	D4 Signée D13 Signée	D4 Signée D13 Signé

Légendes :

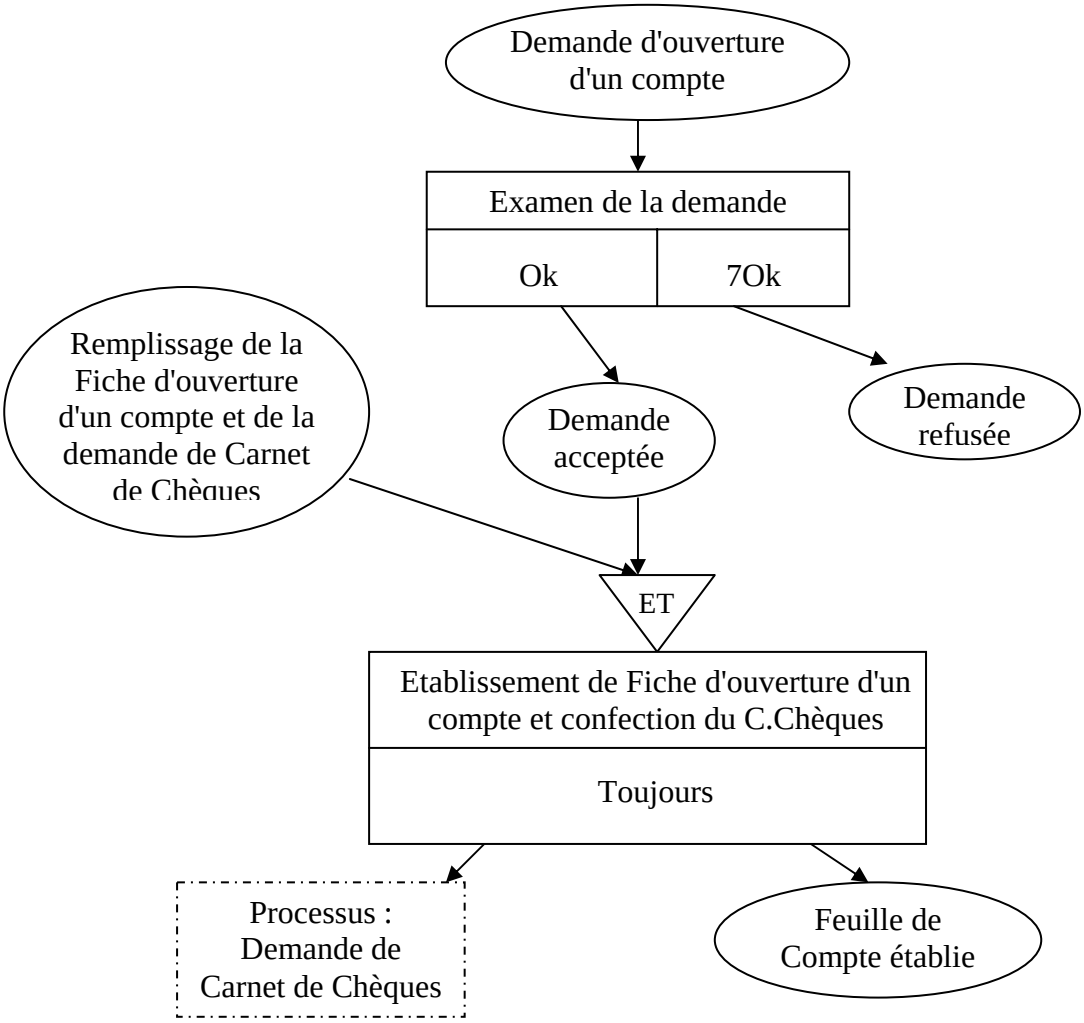
D4 : Fiche de Versement.
P.I : Pièce d'identité.
D13 : Récépissé de Versement.

Diagramme 3 : Versement

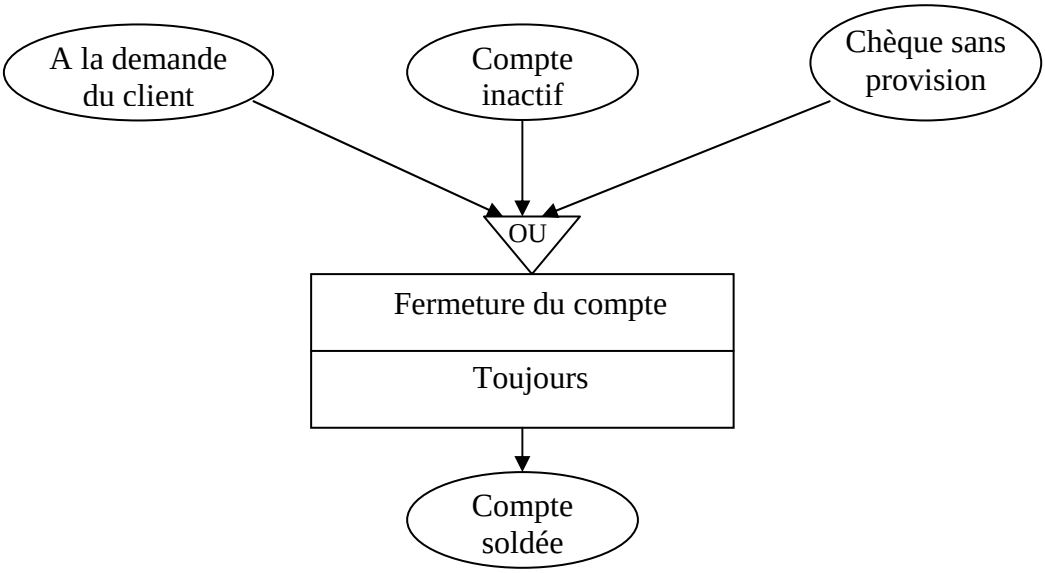

Client
Subdivision du Portefeuille
Service Comptabilité
Guichetier
Chef de Service
Caisse
T13
D4
T14
D4
P.I
T15
P.I
D4
T16
Somme d'Argent
D4
T17
Somme d'Argent
D4
T18
D4
T19
D4
T20
D13
D4
T21
D13
D4
T22
D13
D4
D13

6.2. Construction du Modèle Conceptuel des Traitements :

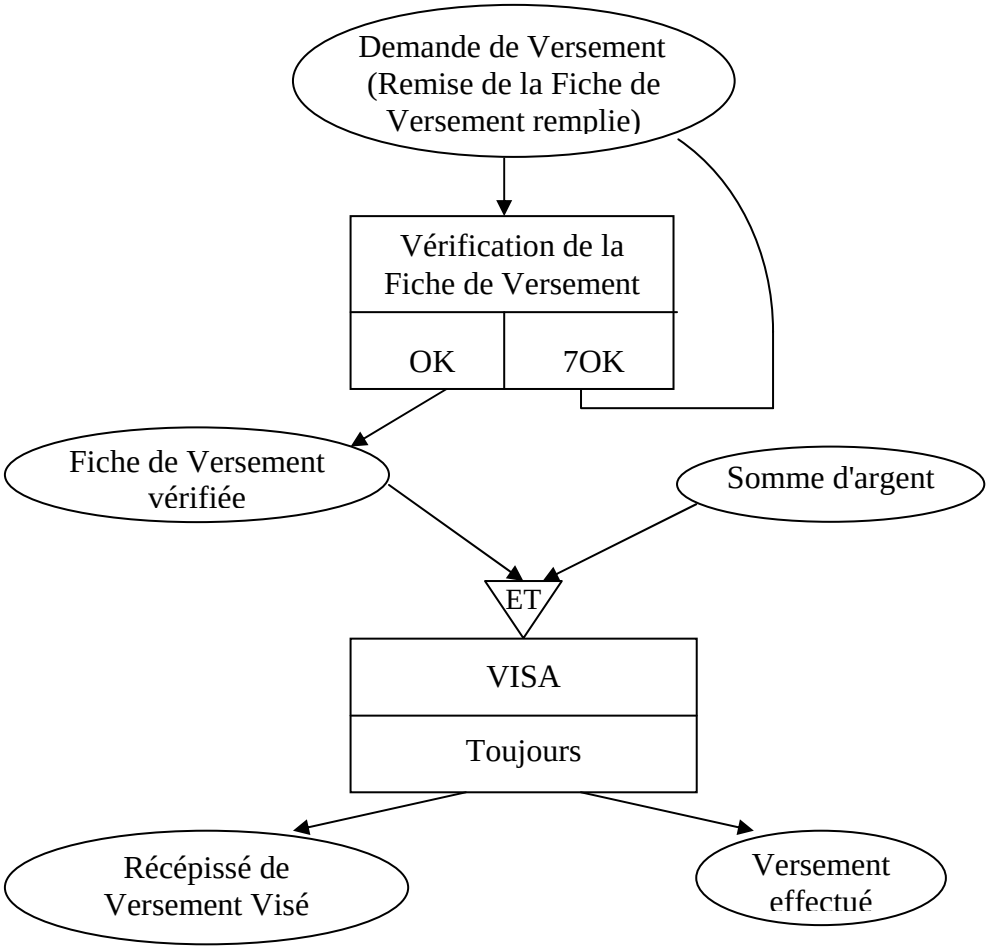
<u>Processus :</u> Ouverture d'un nouveau compte



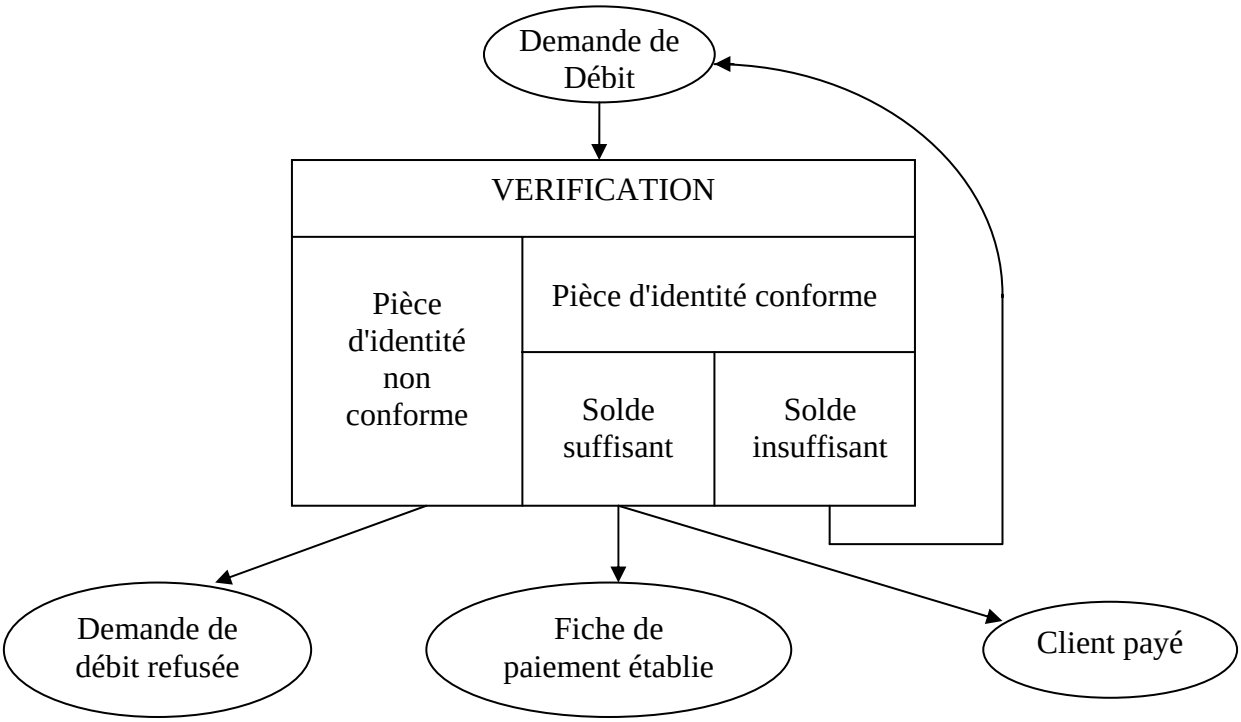
<u>Processus :</u> Fermeture d'un compte (compte soldé)



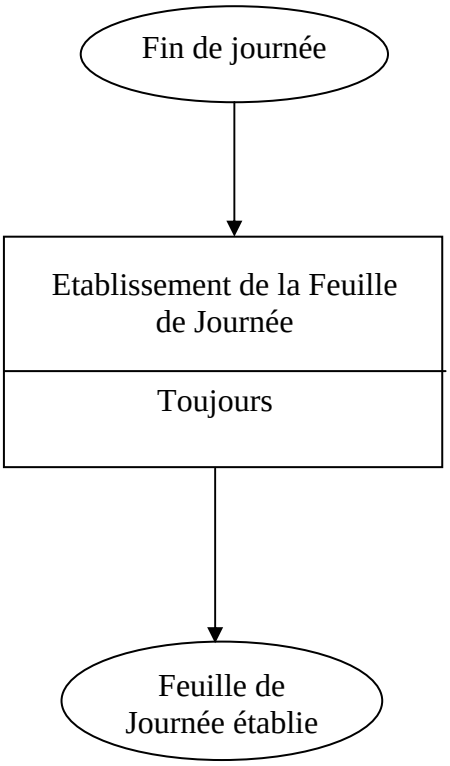
Processus : Versement



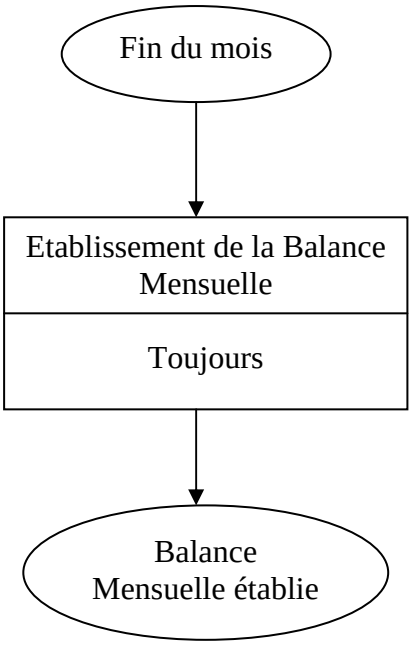
Processus : Débit (Paiement du client)



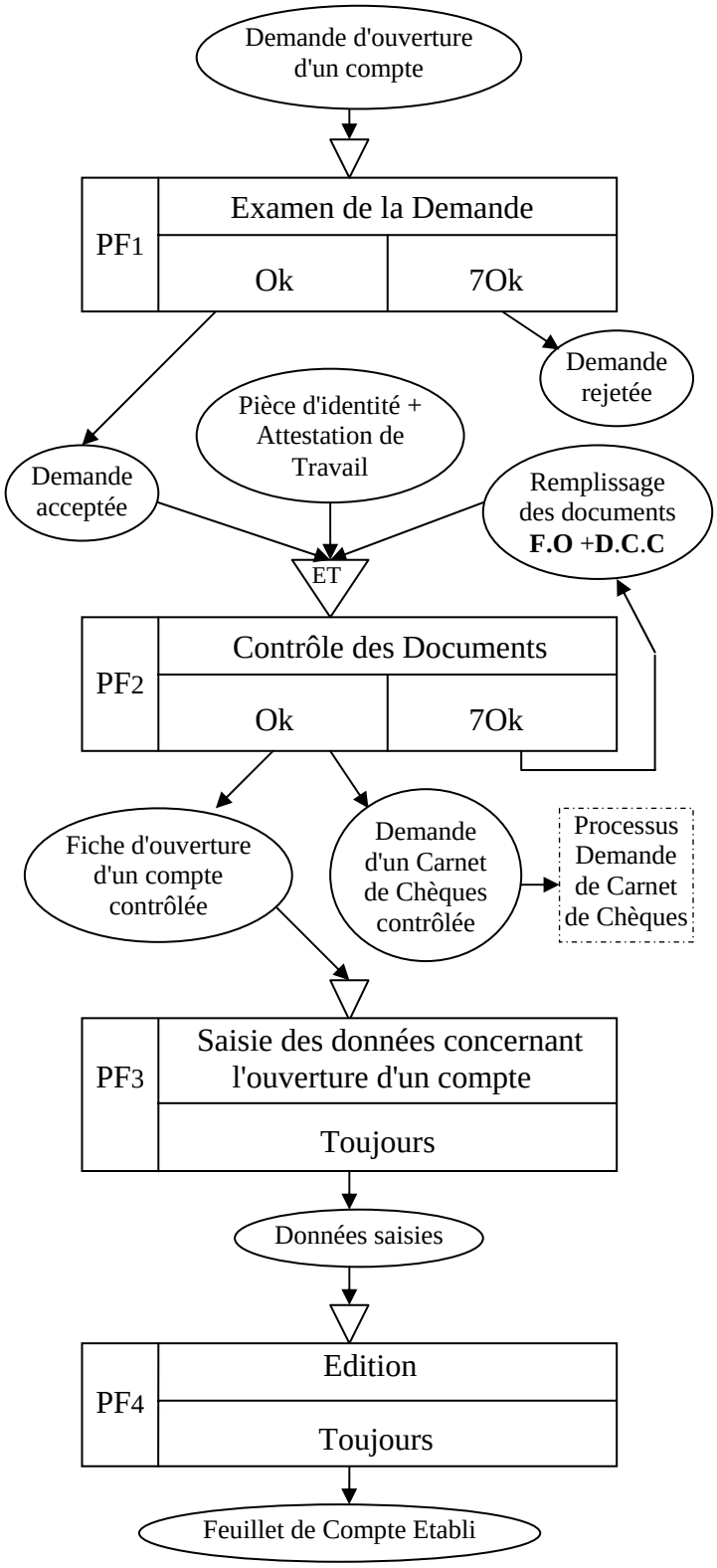
<u>Processus :</u> Traitement à effectuer chaque fin de journée



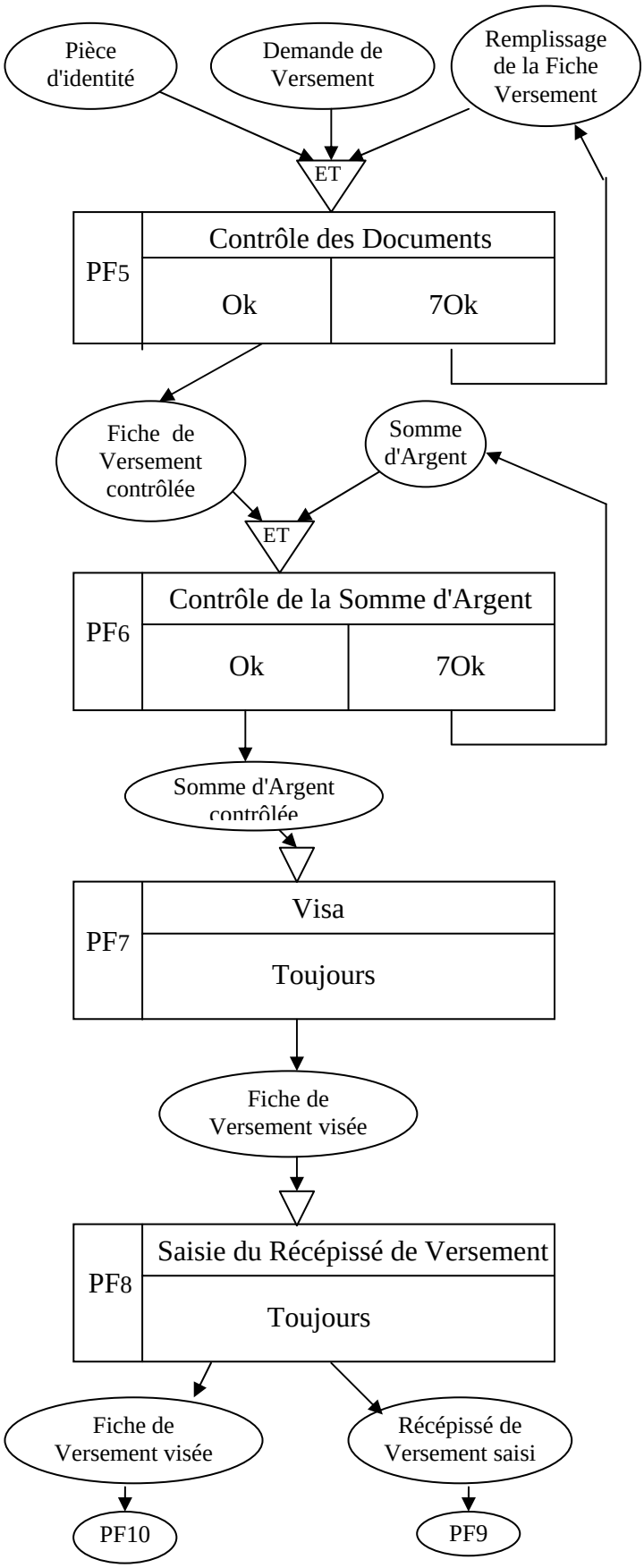
<u>Processus :</u> Traitement à effectuer chaque fin de mois

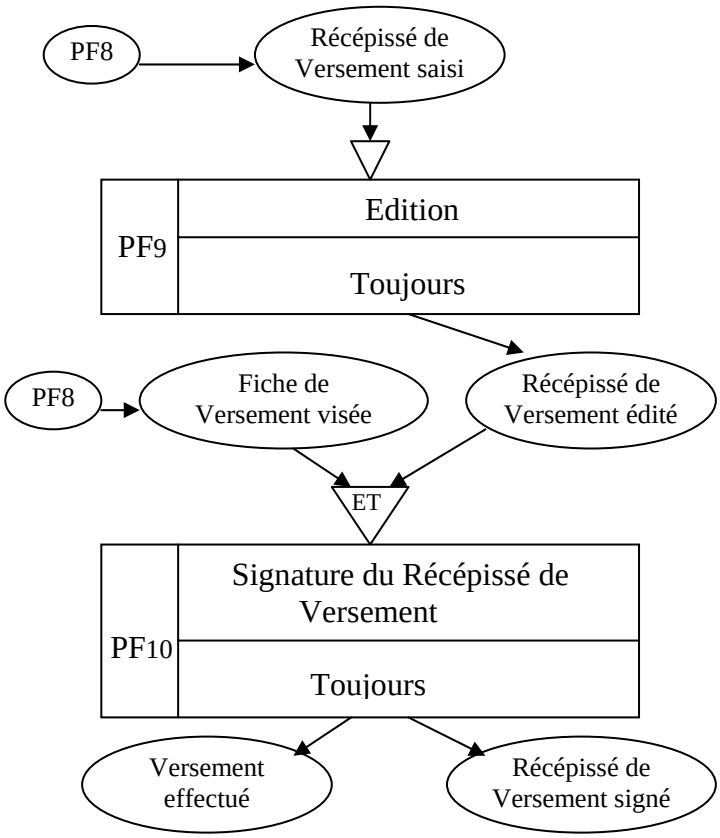


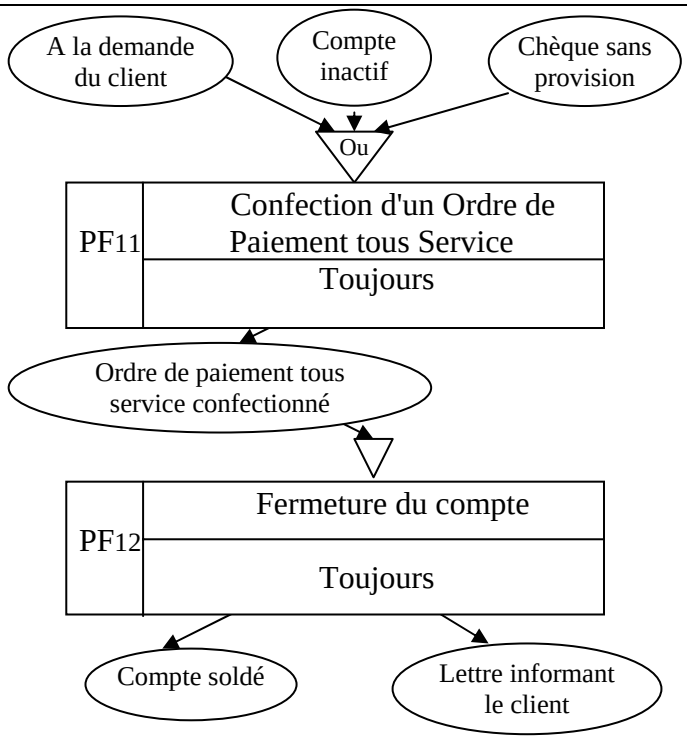
7. Modèle Organisationnel des Traitements :

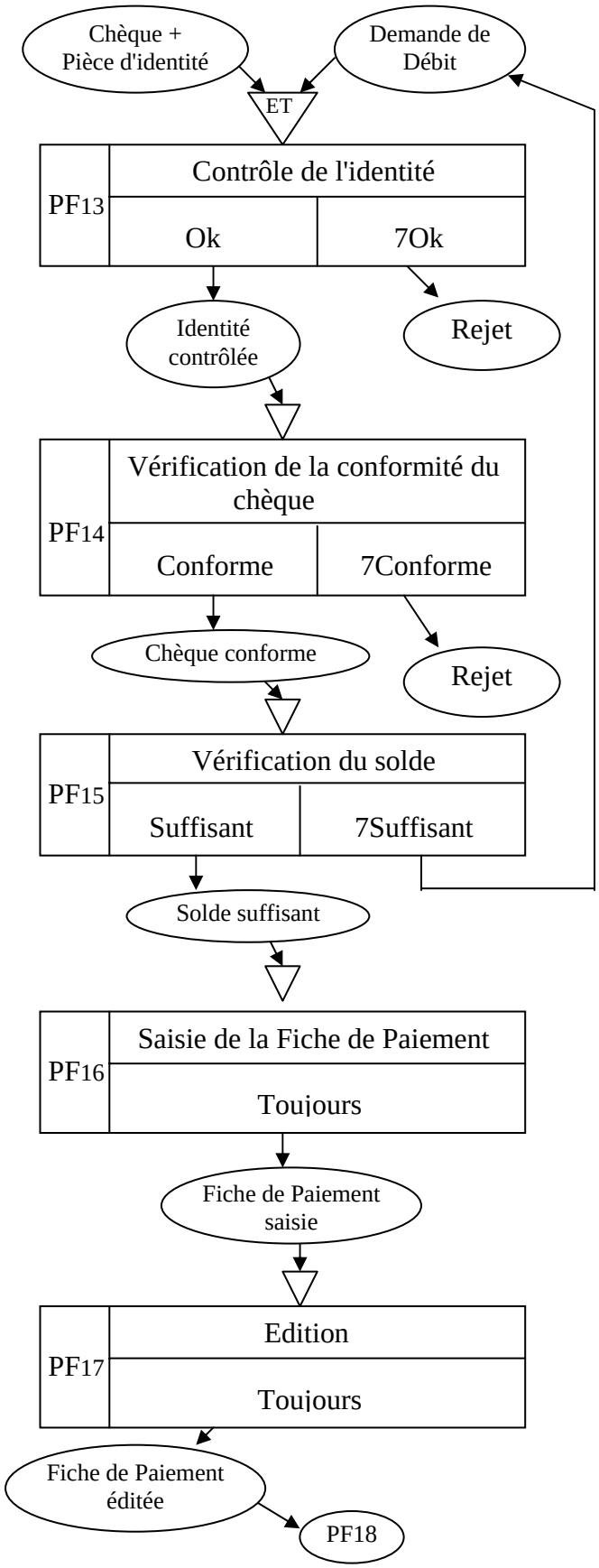
Période	Enchaînement des PFs Processus : Ouverture d'un nouveau compte	Nature	Poste de Travail : - Lieu - Responsable
		M M AC AB	Guichets du Portefeuille Guichetier Guichets du Portefeuille Guichetier Guichets du Portefeuille Guichetier Guichets du Portefeuille Ordinateur

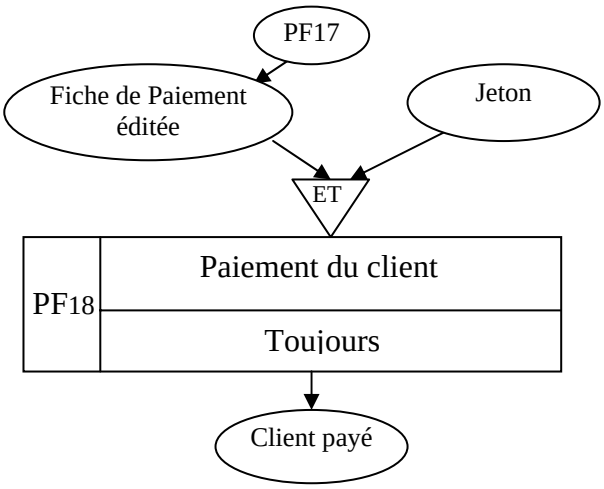
Note : **M** : **M**anuel. **AC** : **A**utomatisée **C**onversationnelle. **AB** : **A**utomatisée **B**atch.
F.O :**F**iche d'**O**uverture d'un compte. **D.C.C** : **D**emande de **C**arnet de **C**hèques.

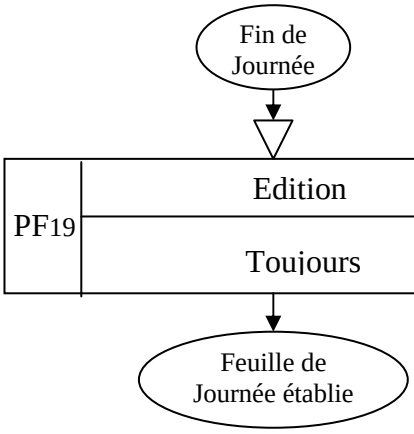
Période	Enchaînement des PFs Processus : Versement	Nature	Poste de Travail : - Lieu - Responsable
		M M M AC	Guichets Guichetier Caisse Caissier Caisse Caissier Guichets Guichetier

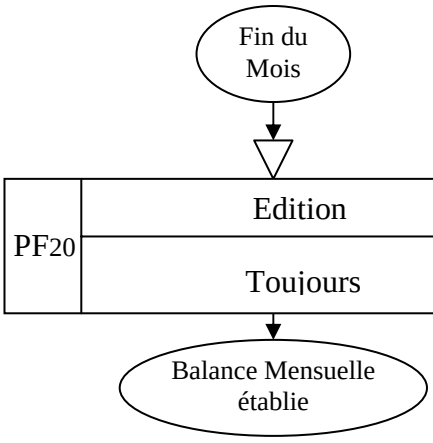
Période	Enchaînement des PFs Processus : Versement (suite)	Nature	Poste de Travail : - Lieu - Responsable
		AB M	Guichets Ordinateur Bureau du Chef de Service Chef de Service

Période	Enchaînement des PFs Processus Fermeture d'un compte	Nature	Poste de Travail : - Lieu - Responsable
		M M	Guichets Guichetier Guichets Guichetier

Période	Enchaînement des PFs Processus : Paiement	Nature	Poste de Travail : - Lieu - Responsable
		M AC AC AC AB	Guichets Guichetier Guichets Guichetier Guichets Guichetier Guichets Guichetier Guichets Ordinateur

Période	**Enchaînement des PFs** **Processus : Paiement (suite)**	**Nature**	**Poste de Travail :** **- Lieu** **- Responsable**
		M	Caisse Caissier

Période	**Enchaînement des PFs** **Processus Traitement de chaque Fin de Journée**	**Nature**	**Poste de Travail :** **- Lieu** **- Responsable**
Une heure avant la fermeture (15h30)		AB	Service Informatique Ordinateur

Période	**Enchaînement des PFs** **Processus Traitement de chaque Fin de Mois**	**Nature**	**Poste de Travail :** **- Lieu** **- Responsable**
A la Fin de chaque de Mois		AB	Service Informatique Ordinateur

8. Modèles Externes :

Modèle Externe de la procédure PF3 :

En mise à jour :

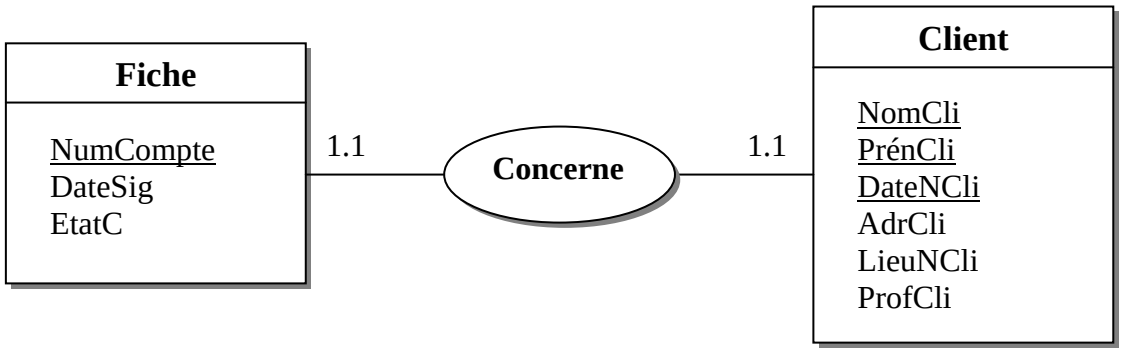
Le diagramme de répartition des taches H/M [*Fiche d'ouverture d'un compte*] :

HOMME	MACHINE	ECRAN DE SAISIE
	1. La machine affiche le message ---- et affecte à cette fiche un numéro séquentiel	Numéro de Fiche :
	2. Affiche le message---------- et donne la date courante	Date :
4. Entre le nom du client	3. Affiche le message ----------	Nom :
6. Donne le prénom du client	5. La machine affiche le message-----	Prénom :
8. Donne l'adresse du client	7. Affiche le message----------	Adresse :
10. Donne la date de naissance du client	9. Affiche le message----------	Date de Naissance:
12. Donne le lieu de naissance du client	11. Affiche le message----------	Lieu de Naissance :
14.entre la profession du client	13. Affiche le message----------	Profession :

On voit que ce diagramme est reparti en deux parties, la première concerne les données propres au client et la deuxième, les données (NumCompte, DateSig, EtatC) et qui concerne le compte.

Note : On peut retrouver le numéro du compte à partir du Nom du Client, son Prénom et sa Date de naissance (au cas ou le client l'aurait oublié).

Donc on peut éclater la fiche en deux parties :



NB : L'attribut DateSig est la Date d'ouverture (Date de Signature) du compte.

Modèle externe de la procédure PF8 :

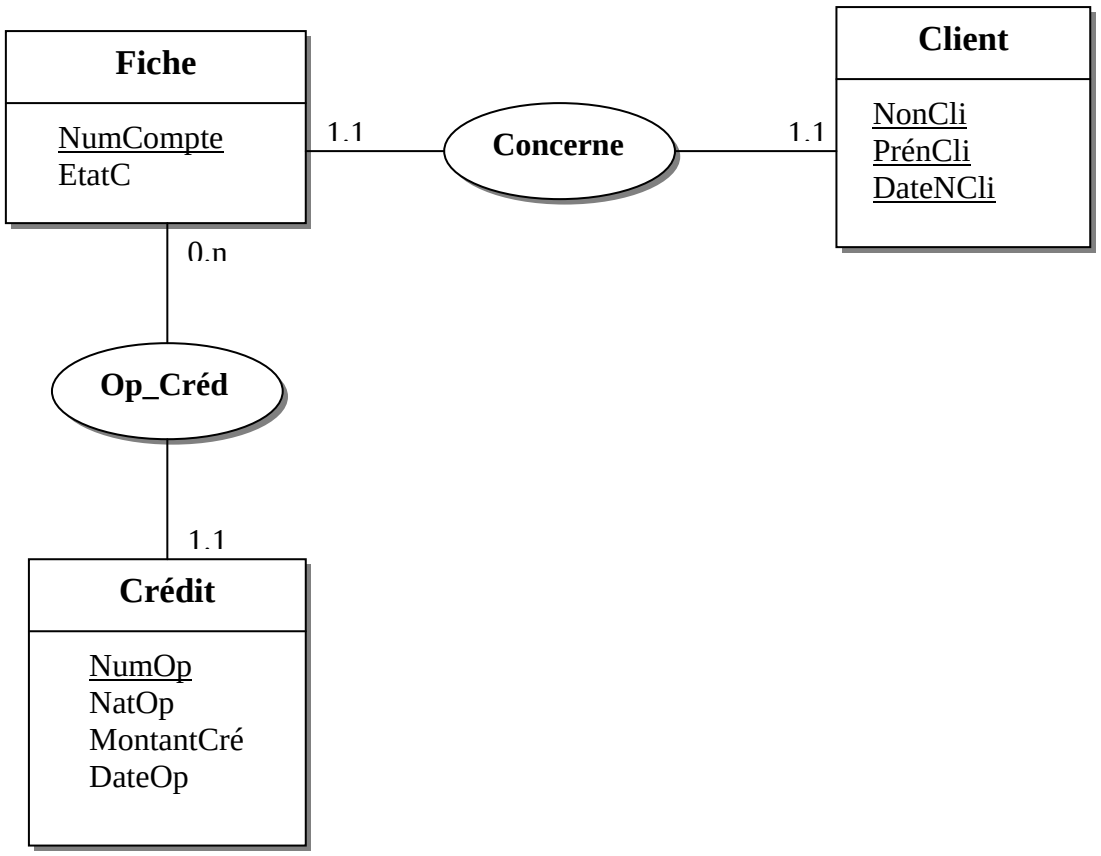
En mise à jour :

Diagramme de répartition des taches H/M [*Fiche de Versement*] :

HOMME	MACHINE	ECRAN DE SAISIE
	1. Affiche le message---------- et donne la date courante	Date :
3.Donne le numéro de compte a crédité	2. Affiche message ----------	Compte à Créditer N° :
5.Donne le libellé du numéro de compte	4. Affiche message ----------	Au Nom de :
7.Donne le montant versé	6. Affiche message ----------	Montant Versé :
9.Donne le montant en lettres	8. Affiche message ----------	Montant en Lettre

Remarque : On ne prend pas en considération le "montant en lettre", c'est une donnée organisationnelle.

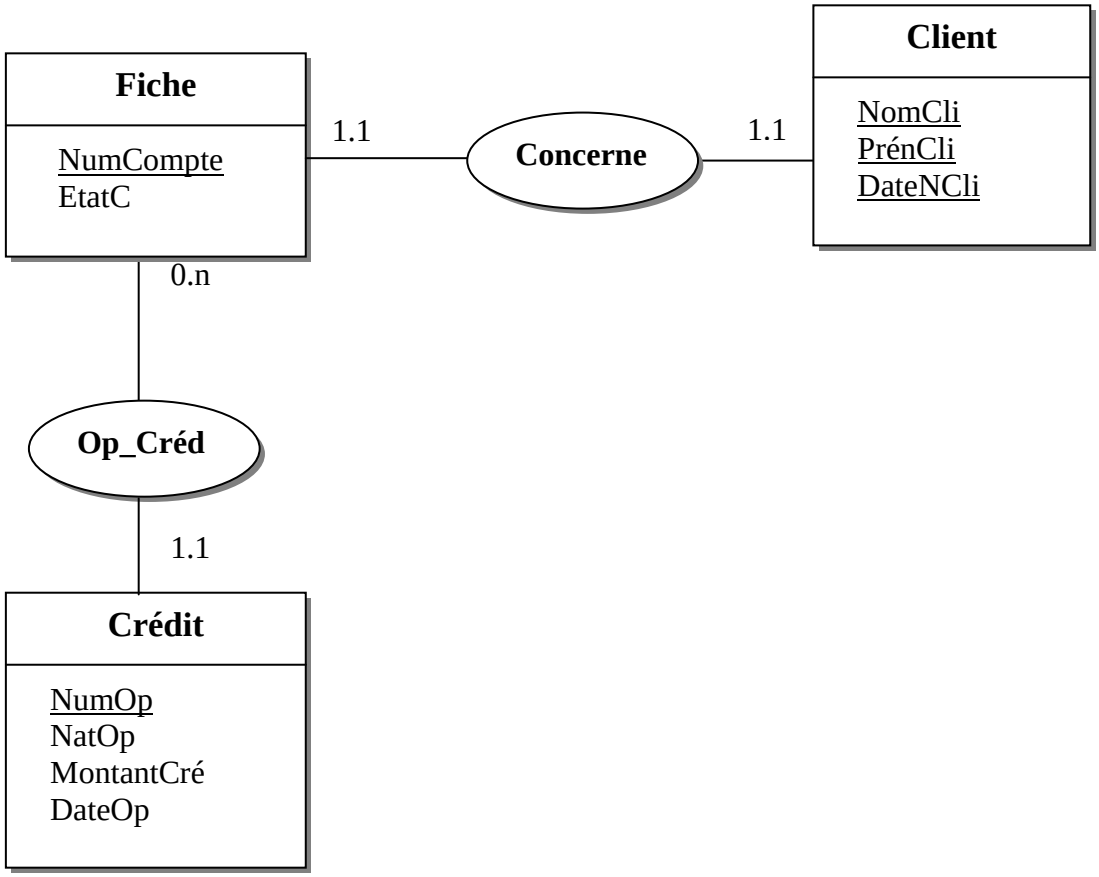
Dans cette vue externe, on remarque que l'entité "Fiche" n'opère qu'avec le numéro de compte et la balance d'entrée c'est pourquoi la vue externe de PF8 aura la forme suivante :



En consultation :

HOMME	MACHINE	ECRAN DE SAISIE
	1. Affiche le message ---------- et donne la date courante	Date :
3. Donne le N° du compte	2. Affiche le message : ----------	Compte à créditer N° :
	4. Affiche le message---------- et affecte à l'opération un numéro séquentiel.	Numéro d'Opération:
6. Détermine la somme à créditer	5. Affiche le message : ----------	Montant Versé :
8. Donne la somme en lettres	7. Affiche le message : ----------	Somme versée en lettres :

Chaque opération ne concerne que le numéro de compte et la balance d'entrée, donc on éclate l'entité "Fiche" en deux et on obtiendra :

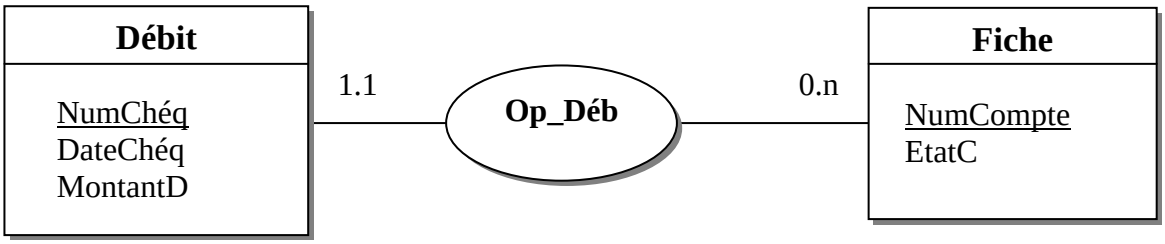


Modèle Externe de la Procédure PF 14 :

En consultation :

On détermine grâce au chèque le numéro du compte correspondant, puis on teste si le chèque à été déjà utilisé (en cas de fraude).

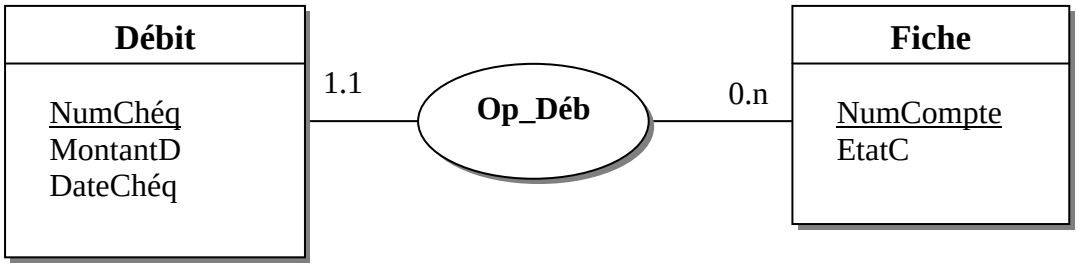
La vue externe se résume à çà :



Modèle Externe de la procédure PF15 (Vérification du Solde) :

En Consultation :

Le guichetier vérifie le montant du chèque avec le solde du compte concerné. On aura :



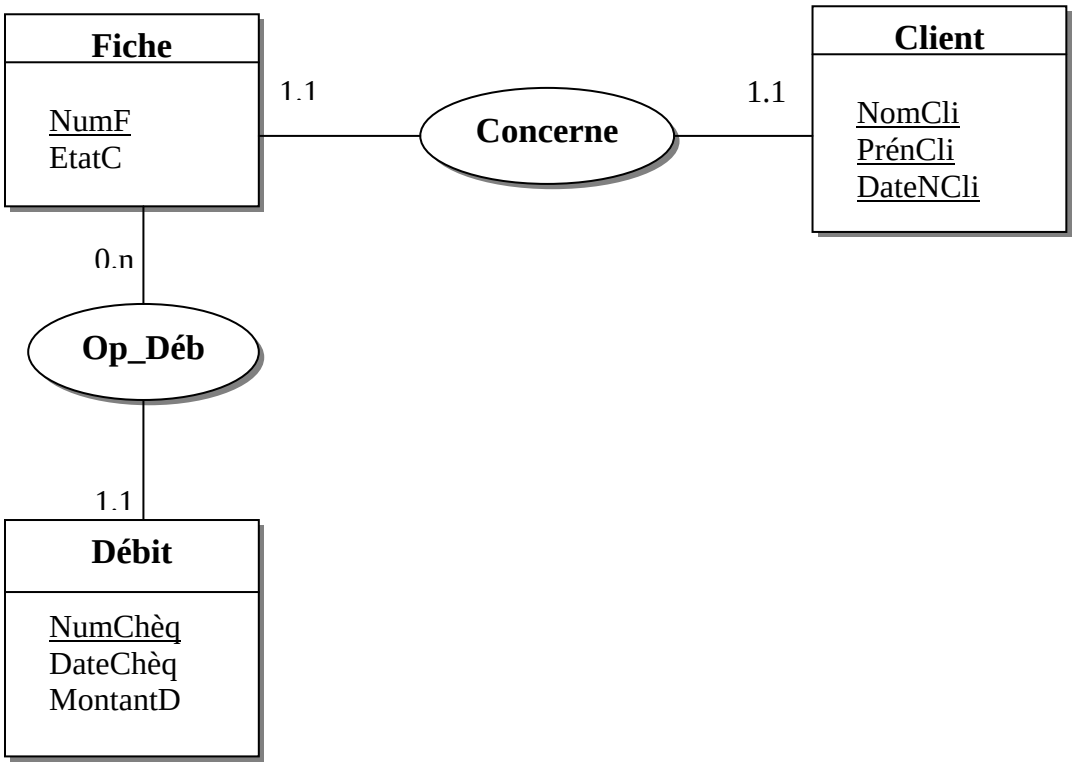
Modèle Externe de la procédure PF 16 :

En consultation :

Diagramme de répartition des taches H/M [*Fiche de paiement*] :

HOMME	MACHINE	ECRAN DE SAISIE
	1. Affiche le message ---------- et donne la date courante	DATE :
3. Donne le numéro de la fiche concernée	2. Affiche le message ----------	CPTE N°
5. Le numéro de chèque	4. Affiche le message ----------	CHEQUE N°
7. Donne la somme du débit	6. Affiche le message ----------	Somme à payer :
9. Donne le montant en lettre	8. Affiche le message ----------	Somme en lettres :

Remarque : On ne prend pas en considération "montant en lettre", c'est une donnée organisationnelle.

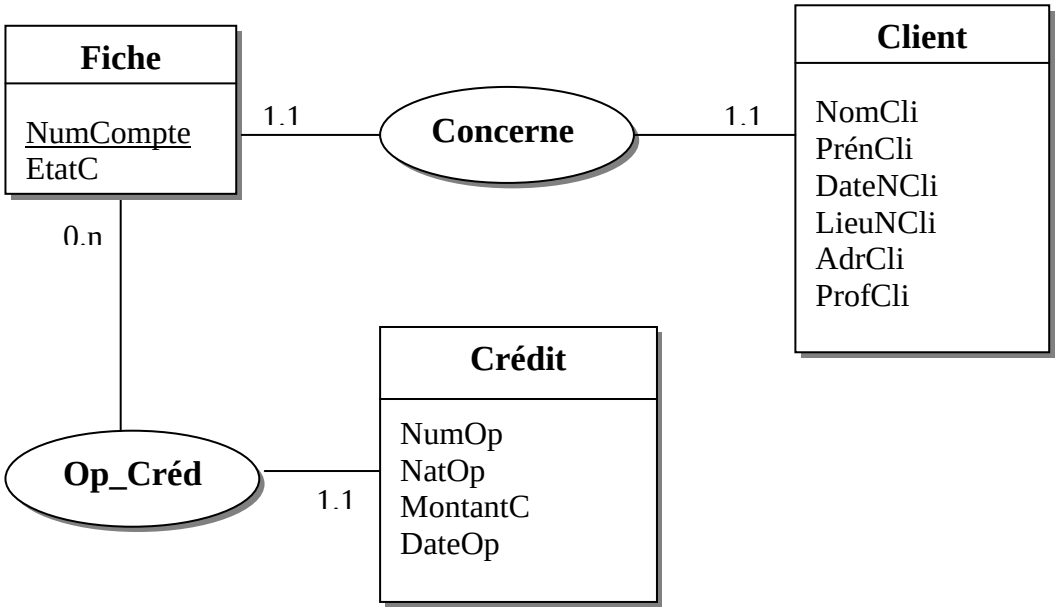


9. Validation entre le MCD et les Modèles Externes :

Dans le modèle externe de la procédure **PF8**

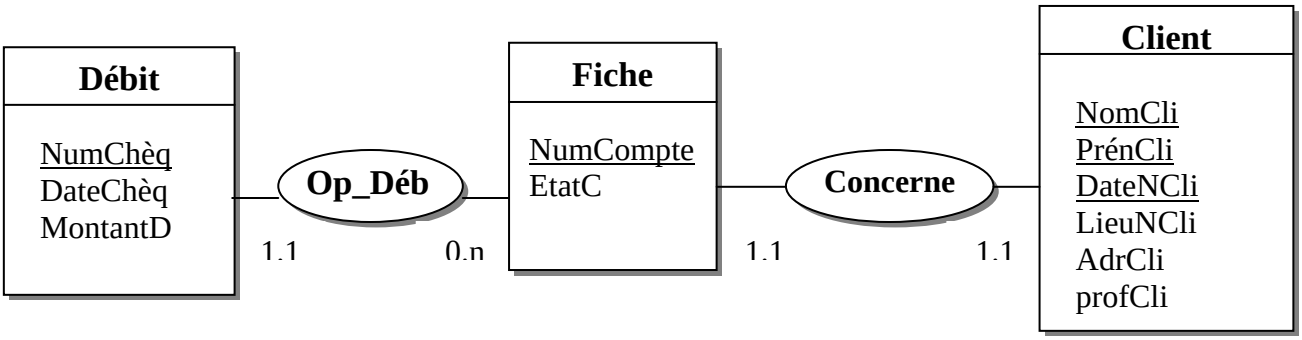
- Tous les montants en lettres ne sont pas considérés.
- On a aussi changé la cardinalité de la relation "**Fiche-Crédit**" de "1.n " a " 0.n " parce qu'une fiche peut ne pas avoir d'opération.
- On a aussi éclaté l'entité "Fiche" et "Client" la première, contient le numéro de fiche concerné et sa balance d'entrée, ces données son utilisées dans le traitement de cette vue externe (PF8), et l'autre contient les informations propres au client (propriétaire du compte) et qui ne sont pas considérées par les traitements.

On obtient le MCD validé suivant :



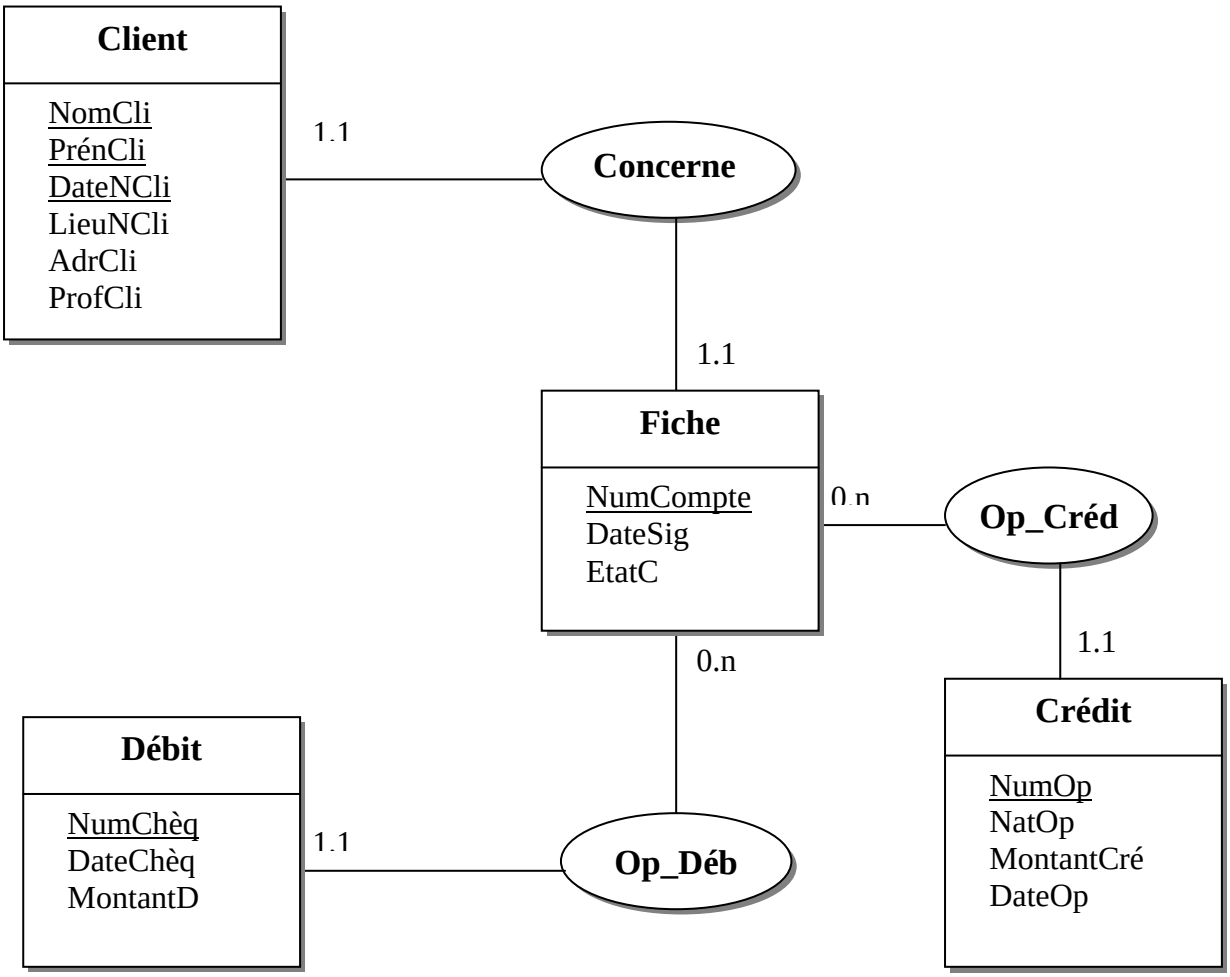
La même chose en ce qui concerne le traitement de la vue externe PF16 :

- Changer la cardinalité de la relation "**Fiche-Débit**" de "1.n" à "0.n".
- Eclater la l'entité "Fiche" en deux



On a établit les cardinalités (1.1)-(1.1) entre "**Fiche**" et "**Client**" parce qu'une fiche (compte) ne concerne qu'un seul client et qu'un client ne peut avoir plusieurs comptes dans la même banque.

MCD VALIDE



10. Modèle Logique des Données :

En utilisant les règles de passege du MCD Entités/Relations au MLD relationnel on obtient le MLD suivant :

Fiche (NumCompte, DateSig, EtatC, NomCli, PrénCli, DateNCli).
Client (NomCli, PrénCli, DateNCli, LieuNCli, AdrCli, ProfCli).
Débit (NumChèq, DateChèq, MontantD, NumCompte).
Crédit (NumOp, NatOp, DateOp, MontantCré, NumCompte).

11. Conclusion :

Nous venons par le biais de ce travail de concevoir un Système d'Information relatif à la Gestion du Portefeuille au niveau du trésor public de Constantine("Fonds Particuliers"). Notre objectif étant d'obtenir un schéma de Base de Données normalisé (en mettant en pratique et sans s'approfondir dans les concepts, une méthode que nous connaissons déjà : méthode MERISE) afin d'implémenter notre application et s'assurer de l'intégrité des données utilisées.

1. Introduction :

Après avoir présenté le langage JAVA et défini ses principaux caractéristiques, expliqué les concepts et les mécanismes relatifs à l'interrogation de bases de données via l'API JDBC et la conception d'un Système d'Information proportionnel à notre application ; nous arrivons à la réalisation proprement dite.

Nous allons tenter au cours de ce chapitre d'expliquer les différentes étapes qui ont conduit à la réalisation d'une application de type Intranet relative à la gestion du Portefeuille et qui à notre sens pourrait être l'exemple type d'une application Intranet pouvant tirer profit d'une architecture JAVA en générale et de l'API JDBC en particulier.

2. Objectifs :

Notre objectif est de concrétiser tout ce que nous avons appris jusqu à présent en une application réelle. Pour cela nous avons choisi de la réaliser en 2 versions :

La première : est une application de type Intranet interagissant simultanément avec deux bases de données distantes à l'aide de requêtes SQL à travers une **Applet** et qui pourra s'exécutée sur n'importe quelles plate-formes pourvues d'une machine virtuelle Java (en l'occurrence un navigateur compatible Java). Cela se fait par l'inclusion d'une Applet dans une page HTML (*Hyper Text Markup Langage*) [11].

La seconde : est une **Application** qui se déroulera sur une machine monoposte (l'interface utilisateur est la même que celle de l'Applet) et qui s'exécutera par le lancement de l'interpréteur en ligne de commande de Java.

3. Réalisation :

3.1. 1ère Version : Une Applet

Toute démarche de réalisation d'application d'une certaine envergure doit nécessairement englober les aspects suivants :

- Architecture logicielle.
- Outils de conception utilisés.
- Etapes de réalisation.

3.1.1. Architecture logicielle :

a- Principes de développement :

Les schémas d'architecture donnés ci-après s'appuient sur le **principe de base** suivant : **le seul logiciel installé sur le poste de l'utilisateur final est un navigateur standardisé** (par exemple Netscape Communicator -que nous présenterons dans la suite) qui constitue la fenêtre d'accès unique vers le système d'information de l'entreprise. Les ressources nécessaires à l'utilisation du système d'information sont stockées sur des serveurs, et sont téléchargées sur le poste final. Ces ressources peuvent être : des données, des composants ou des applications.

A partir de ce principe fondateur d'une application de type Intranet, différentes déclinaisons sont possibles selon les besoins. Les architectures Intranet ne sont donc pas fondamentalement différentes des architectures Client/Serveur classiques, les distinctions majeures amenées par Intranet/Internet étant cependant :

- L'utilisation de protocoles standardisés.
- La généralisation de la capacité de téléchargement.

Les similitudes se retrouvent notamment dans la distinction entre les architectures à deux niveaux et les architectures à trois niveaux.

Pour bien comprendre les architectures Java, il convient de faire un petit rappel sur les architectures Intranet en général.

b- Typologie des architectures Intranet :

Les architectures Intranet peuvent se séparer en deux grandes familles :

- Les architectures à deux niveaux, où un client discute directement avec un serveur de traitement et de données.

- Les architectures à trois ou multiniveaux où au moins un serveur intermédiaire est intercalé entre les postes clients et les serveurs de données.

Notre intérêt portera sur l'architecture à deux niveaux pour sa simplicité à mettre en œuvre.

Une architecture à deux niveaux n'effectue pas de distinction précise dans le codage et les outils mis en œuvre entre interface utilisateur, traitements et données. Les architectures à deux niveaux sont généralement organisées comme suit (Fig. n°4.1 : A et B) :

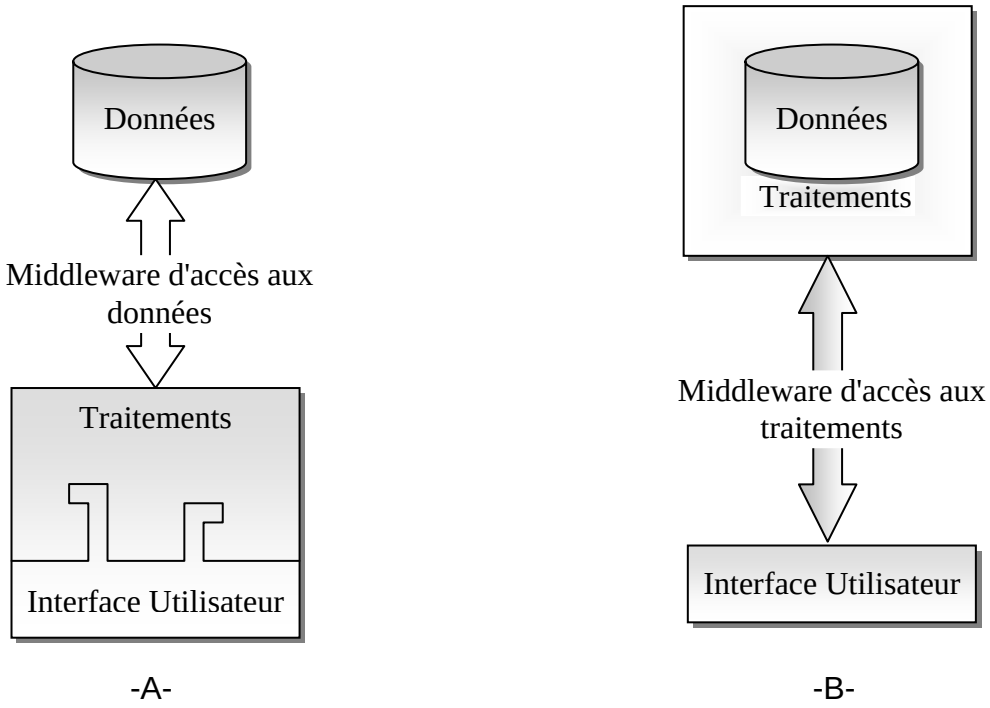


Fig. n°4.1: Schéma représentant les possibilités d'architecture deux niveaux [2].

Le premier type d'architecture décrit ci-dessus(Fig. n°4.1: -A-) ne différencie pas réellement l'interface utilisateur des traitements : généralement les objets de l'interface utilisateur lancent des requêtes SQL vers le moteur de base de données et récupèrent les résultats directement. Un middleware transporte les requêtes entre les deux parties.

Le second type d'architecture(Fig. n°4.1: -B-) ne différencie pas les traitements des données. C'est le cas des procédures stockées pour un SGBD relationnel ou bien d'appel de services depuis l'interface utilisateur. Un middleware transporte les messages entre les deux parties.

Dans notre cas, notre application utilise **un mélange** des possibilités d'architectures à deux niveaux décrite précédemment(Fig. n°4.1). La première se fait par les objets de l'interface utilisateur qui lancent des requêtes SQL vers le moteur de la BD grâce à l'objet ***Statement*** (décrit en détail au chapitre précédent) et récupère les résultats avec l'objet ***ResultSet*** comme le démontre ces quelques lignes de code :

```
//En réaction à un événement déclenché par un composant de l'interface utilisateur

Statement  statement = connexion.createStatement() ;

//Exécution d'une requête avec la méthode executeQury()

ResultSet rs = statement.executeQuery (" SELECT NumCompte, DateS, EtaC FROM Compte ") ;
```

La seconde possibilité d'architecture apparaît avec l'utilisation de procédure stockée(gérée par l'objet ***CallableStatement***) et de requêtes pré compilées(voir le chapitre précédent) et qui font que la limite entre données et traitements est plus mince. Pour ce faire il suffit de préparer la procédure dans la base de données et de l'invoquer en citant son nom dans le code Java et cela se traduit par les lignes suivantes :

```
CallableStatement cs = connexion.prepareCall("{call FICHE_DE_COMPTE}");

ResultSet rs = cs.executeQuery() ;
```

FICHE_DE_COMPTE étant une procédure stockée dans la base de données crée à l'aide d'Access.

c- Architecture deux niveaux avec Java :

Cette architecture utilise des avancées majeures apportées par les technologies Intranet, introduite avec Java : le téléchargement de code à la demande, qui permet un enrichissement de l'interface HTML nettement supérieur, ainsi qu'un contournement des limites du protocole HTTP.

Dans une architecture à deux niveaux, les objets chargés comportent des traitements fonctionnels(par exemple ils émettent des requêtes SQL), et peuvent aussi contenir du code technique d'encapsulation d'un protocole spécifique. Le schéma (Fig. n°4.2) résume parfaitement les propos cités ci-dessus :

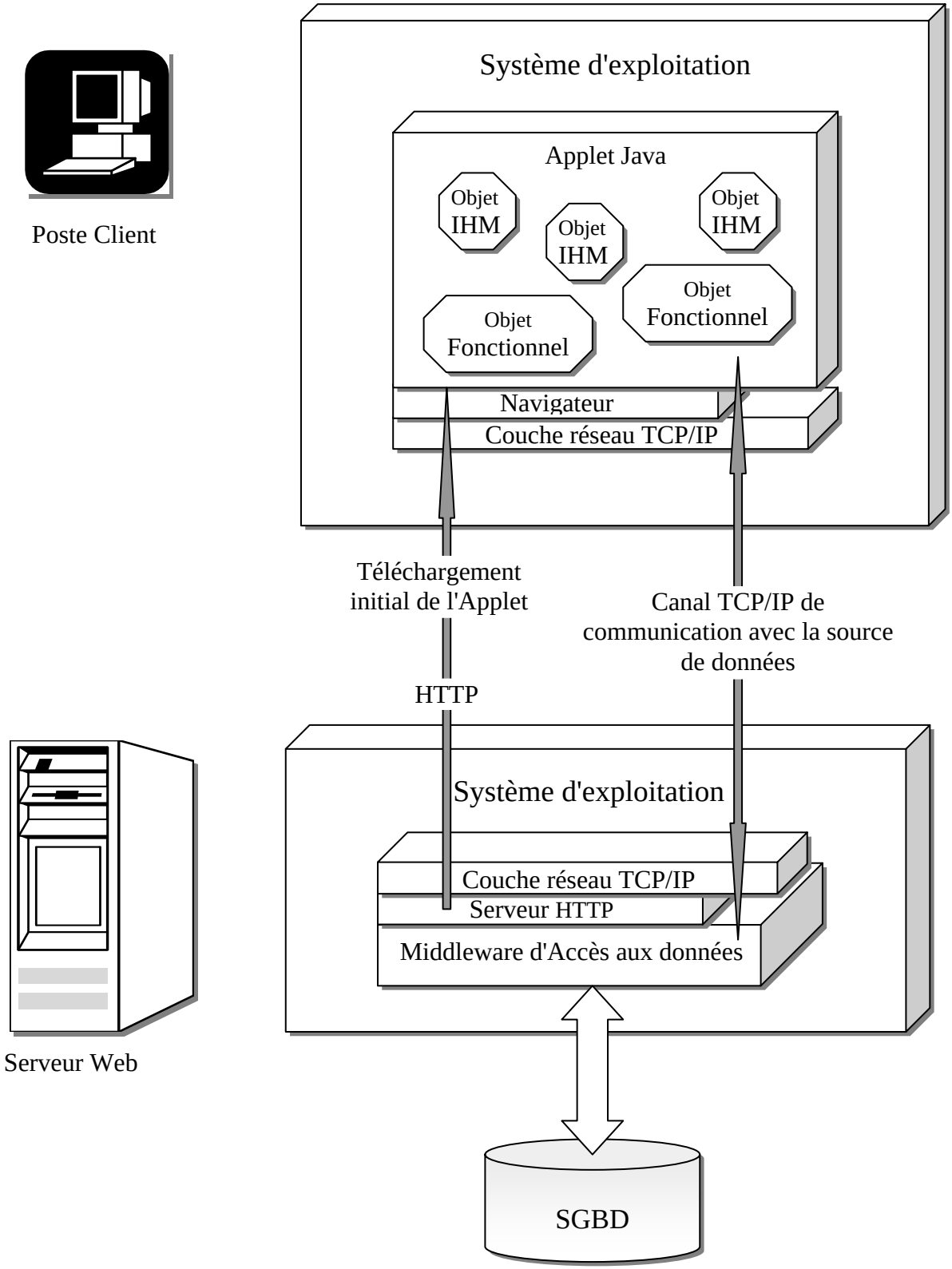


Fig. n°4.2 : Exemple d'architecture deux niveaux avec Java [2].

Avantages :

- Interface utilisateur beaucoup plus riche.
- Meilleure sécurité.

Inconvénients :

- Maîtrise de la programmation objet nécessaire.
- Pas de séparation nette entre IHM (Interface Homme Machine) et traitements.

Cadre d'utilisation :

- Tout type d'application d'entreprise, sur des projets de petite ou moyenne taille.

L'utilisation d'un langage de programmation objet comme Java nécessite l'adoption d'une méthodologie de conception d'applications à partir d'un certain niveau de complexité. On peut considérer que la nécessité d'une méthode de conception s'impose dés qu'il y a plus de deux développeurs sur le projet.

En effet le formalisme imposé par la méthode est un bon moyen de s'assurer de la conformité de l'application.

3.1.2. Outils de conception utilisés :

Pour la réalisation de notre application nous avons eu recours aux outils suivants :

- Le JDK(Java Development Kit) version 1.1 [12]: en utilisant son compilateur **javac** pour la compilation des classes et de l'Applet et l'Appletviewer pour le visionnement de l'Applet sans les détails appartenant à la page HTML.

- Un éditeur de texte pour l'écriture de code Java.

- Microsoft Access [4] : pour la création de notre base de données et la préparation de nos procédures stockées.

- Microsoft FrontPage pour la création de pages HTML attrayantes.

- Netscape Communicator (à partir de la version 4.5) [14] pour le visionnement d'Applet à l'intérieur des pages HTML

- Un middleware d'accès aux données [15,16](pilote de type 3 : Qui fournit au client une API réseau générique et, une fois traduit au niveau du serveur, permet l'accès aux bases de données).

Nous avons également voulu insérer l'outil Internet Explorer cependant cet outil n'est pas compatible à 100% avec Java et donc il nous a été impossible de faire tourner notre application sur IExplorer.

Tandis que Netscape effectue la connexion avec la base de données, dessine l'interface utilisateur sans aucun problème et réceptionne normalement les résultats des requêtes ; IExplorer se limite au dessin de l'interface graphique et n'effectue pas la connexion avec la source de données. Pour cette raison, nous avons opté pour l'outil Netscape Communicator.

3.1.3. Etapes suivies dans la Réalisation :

Nous avons choisi de décomposer la phase de réalisation en plusieurs étapes afin qu'elle puisse être bien assimilée et qu'on puisse détailler une à une les opérations effectuées.

a- Préparation de la Base de Données :

Après avoir conçu le Système d'Information relatif à la gestion du Portefeuille au Trésor public de Constantine, nous allons réaliser notre Base de Données (BD) à partir du Modèle Logique de Données (MLD) qu'on a pu tirer en utilisant les règles de passage MCD Entités/Relations au MLD relationnel et conditionner les tables conformément au règles établies dans le dictionnaire de données.

La préparation des tables se fait comme le montre la Fig. n° 4.3 :

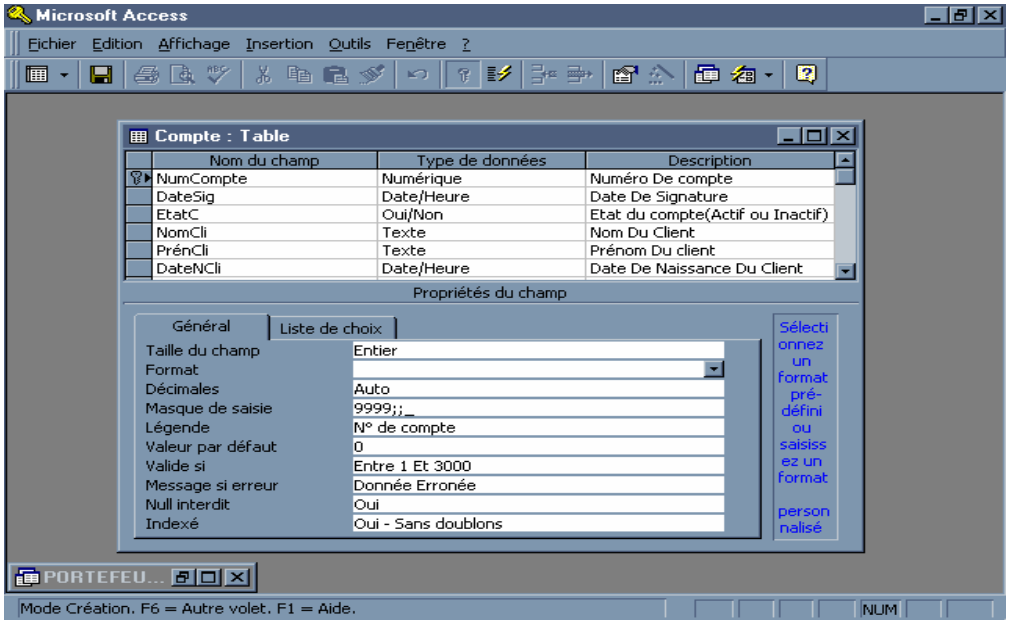


Fig. n° 4.3 montrant la préparation d'une table dans Access.

Grâce aux possibilités qu'offre Access il a été possible de déterminer minutieusement les propriétés de nos attributs : Son type, la taille du champ qui lui est associé, un masque de saisie, le message à afficher en cas d'erreurs et aussi nous procure le choix de l'indexé ou non afin de faciliter la recherche de celui-ci. Toutes ces propriétés font que le risque d'introduire une donnée erronée est réduit au maximum.

Une fois que tous les tables sont crées et préparées correctement il nous reste plus qu' à les relier entre elles en utilisant les cardinalités du MCD validé. Le résultat obtenu est un schéma qui nous offre une assurance de plus que le résultat et conforme au MCD validé comme le montre la Fig. n° : 4.4.

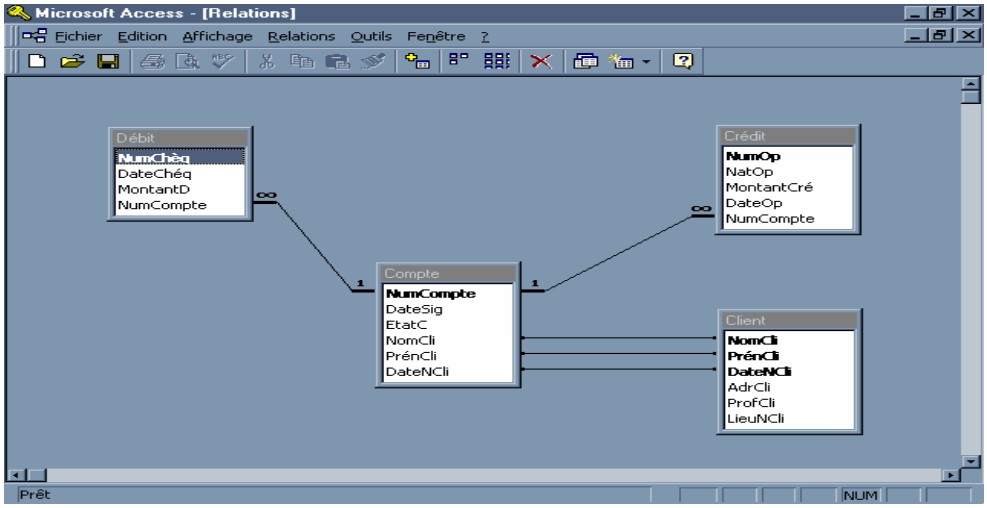


Fig. n° : 4.4 montrant les tables reliées entre elle ainsi que les cardinalités.

b- Configuration du pilote :

Une fois la source de données créer nous attaquons la configuration du pilote et qui est, à notre avis, une démarche très importante car sans elle la connexion aux bases de données serait impossible, c'est pourquoi nous avons jugé préférable d'en parler encore une fois ici dans ce chapitre au risque de nous répété(elle été décrite au chapitre précédent).

La configuration passe par les deux étapes suivantes :

- **La configuration de la source de donnée ODBC:**

  - Cliquer sur l'icône "ODBC 32bit" dans le panneau de configuration, la fenêtre suivante apparaît :

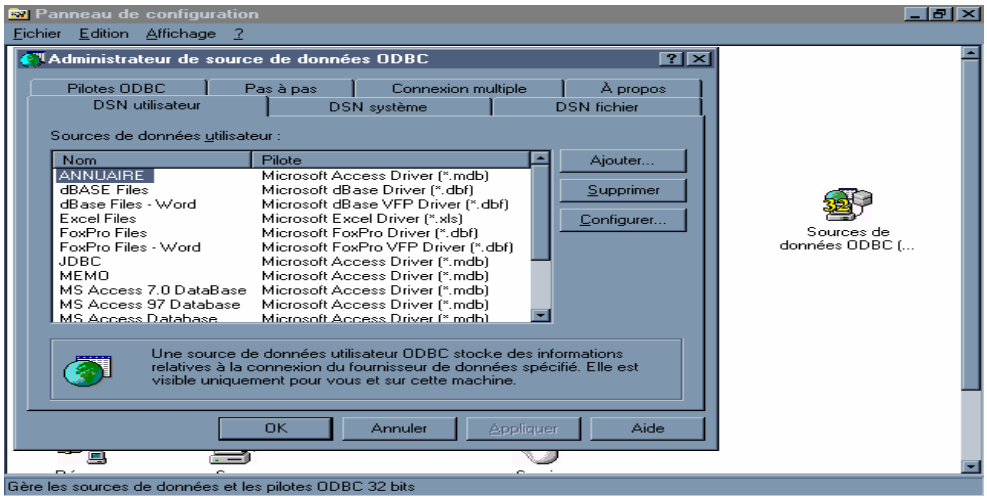


Fig. n° 4.5 : La fenêtre de l'administrateur de source de données ODBC.

- Choisissez l'anglet "DSN utilisateur".
- Sélectionnez un pilote pour lequel vous souhaitez définir une source de donnée (dans notre cas "**driver pour Access**").
- La fenêtre suivante apparaît :

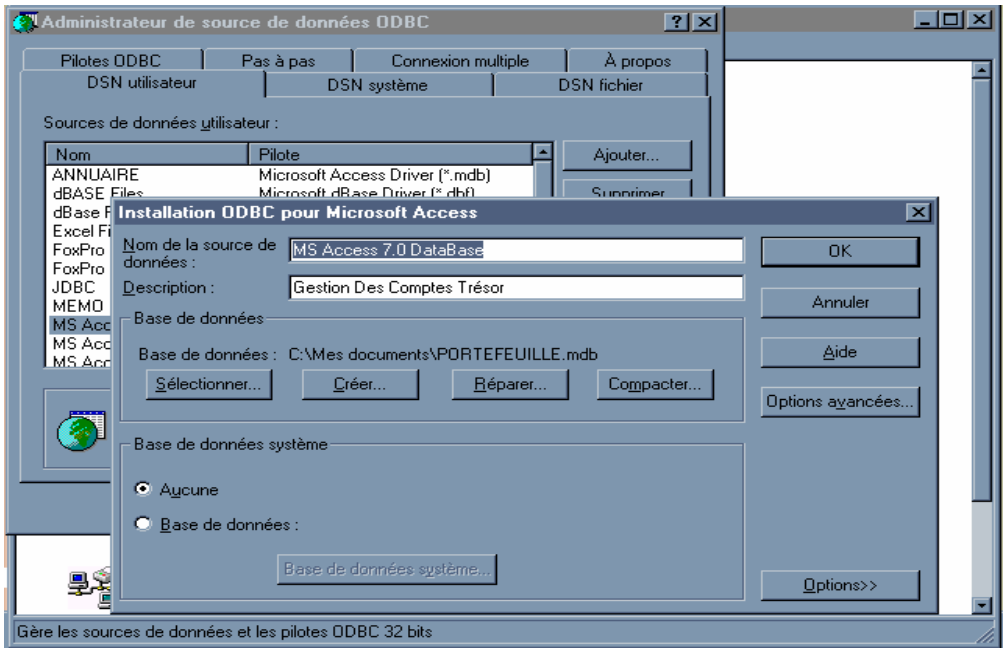


Figure n° 4.6 : Fenêtre d'installation de ODBC pour Microsoft Access.

- Remplir le champ "Nom de la base" (dans notre cas PORTEFEUILLE).
- Remplir le champ "Description" (optionnelle).
- Cliquez sur le bouton "Sélectionner" pour indiquer le chemin exact ou se trouve votre base de donnée.
- Validez en cliquant sur le bouton "OK".

- **<u>Installation d'un pilote de type 3 :</u>**

La configuration de la source de données ODBC ne suffit pas pour que l'Applet puisse interroger des bases de données distantes. Dans ce cas l'emploi d'un pilote de type 3 utilisant les mécanismes d'invocation d'objets à distance (l'API RMI) et compatible avec notre base de données crée à l'aide d'Access, devient indispensable. En faite, ce pilote n'est qu'un pont permettant l'accès à distance aux pilotes JDBC.

La première étape consiste à reconfigurer le fichier **Autoexec.bat** dans le serveur en rajoutant la ligne suivante :

SET CLASSPATH=%CLASSPATH%; c:\MonRépertoire\MonPiloteType3;

Ensuite, on enregistre le pilote dans la liste des pilotes JDBC (qui se trouve dans le serveur). En ce qui concerne les lignes de codes qu'il faut modifier dans l'Applet nous le verrons dans la section qui va suivre.

Maintenant que la base de données est prête et le pilote bien installé nous pouvons entamer l'élaboration de notre application (Applet) qui implémente l'API JDBC.

c- L'Applet :

Il est sûr que nous n'allons pas entrer dans les détails concernant l'écriture du code mais au moins expliquer certains mécanismes propres à JDBC (des détails ont été fournis dans le chapitre précédent).

- **Connexion :**

Une connexion s'obtient par les lignes suivantes :

```
try {

    Class.forName(MonRépertoire.MonPiloteType3) ;    //Le chargement du pilote et l'établissement
                                                     //de la connexion avec la BD.

    String url=new String("jdbc:PiloteProtocole://<HostNam[:Port]>/<SourceDeDonnéesOdbc>");

//PiloteProtocole : le protocole propre au pilote utilisé. Exp. : Pilote RmiJdbc à comme protocole rmi

//HostName : le nom du poste dans lequel se trouve l'application.

//Port : étant le port utilisé par le pilote(optionnel). Exp. : rmiPort=1099.

    Connection  connexion=DriverManager.getConnection(url,"Admin", "Password") ;

catch(SQLException e)  {}
```

- **Exécution d'une requête :**

Il y a trois manières de formuler une requête avec l'API JDBC :

1- Une requête simple en utilisant un objet ***Statement*** :

```
// On suppose la connexion établie

    Statement statement = connexion.createStatement() ;

//connexion étant un objet Connection

    ResultSet rs = statement.executeQuery (" SELECT * FROM ……") ;

//ResultSet étant l'objet qui serre à récupérer les résultats des requêtes.
```

2- Une requête prècompilée en utilisant un objet ***PreparedStatement*** :

On l'utilise quant une requête a tendance à ce répéter et ou il n'y a que l'argument qui change :

```
PreparedStatement preparer = connexion.prepareStatement("DELETE FROM" +

                    "  Compte WHERE NumCompte =  ? ") ;
```

Le symbole "?" étant l'argument à fixer à chaque appel et qui se fait par les méthodes *setXXX()* ou XXX est le type de l'argument(Int, Double, Date, String…etc.).

3- Une procédure stockée avec l'objet ***CallableStatement*** :

C'est le même principe que celui des requêtes prècompilées sauf que la requête se trouve dans la Base de Données et de ce fait **le traitement se fait dans BD** donc plus rapidement que les précédentes.

Les lignes qui suivent montrent comment l'utiliser :

```
try {
    //préparation de l'objet CallabelStatement avec la méthode prepareCall()
      CallabelStatement cs=connexion.prepareCall("{call Feuille_De_Journée (?,?)}") ;
    //On fixe la valeur du premier paramètre en entrée c'est à dire le numéro de compte
      cs.setInt (1, "1500") ;    //le chiffre "1" représente l'emplacement du paramètre
      cs.setDate(2, 12/03/2000 ) ;
    //On a plus qu'à exécuter la procédure stockée
      ResultSet  rs = cs.executeQuery () ;  //il nous reste plus qu'a extraire le résultat
catch(SQLException e)  {}
```

Remarque :

Afin d'assimiler tous ces concepts et de bien les maîtriser nous les avons utilisés tous les trois tout en ayant une préférence pour ceux qui utilisent les procédures stockées (*CallableStatement*) pour sa rapidité et qui nous semble être un détail à ne pas négliger dans une utilisation sous réseau.

- **Récupération des résultats :**

Les résultats d'une requête sont récupérés dans un objet ***ResultSet*** et le parcourt des ses tuples se fait par une boucle *while*, une méthode *next()* pour passer d'un tuple à un autre et une méthode *getXXX(int)* pour extraire les attributs de chaque tuple chacun selon son type (*getInt, getDouble, getString*…etc.).

Quelques lignes de codes s'imposent :

```
//Création d'un objet Statement
    Statement  statement = connexion.createStatement() ;
//Exécution d'une requête avec la méthode executeQury()
```

```
ResultSet rs=statement.executeQuery(" SELECT NumCompte, DateS, BalE FROMCompte ") ;

    // Le résultat de la requête est dans l'objet ResultSet (dans la variable rs)

    //pour parcourir les tuples on utilise la méthode next() associer à une boucle while

        while (rs.next()) {

            int numéroC = rs.getInt (1) ;

            //on peut désigner le champ par son nom : rs.getInt("NumCompte")

             Date dateSign = rs.getDate (2) ;
             float  balance =  rs.getFloat (3) ;

           System.out.println (numéroC+dateSign+balance) ;   }    //pour afficher les résultats.
    }

  catch (SQLException e) {
```

- **Libération des ressources :**

Il impératif de libérer les ressources si on vient à quitter l'application ou quant on a plus à s'en servir. Chacun des objets *Connection, Statement* et *ResultSet* arrivé à sa fin d'utilisation, doit appeler la méthode *close().*

Il est certain que tous ces mécanismes décrit ci-dessus se cachent derrière une Interface Utilisateur Graphique riche en composants : Fenêtres, Barres d'outils, Menus, Boutons…etc.

Paquetages utilisés :

Il est impensable de citer toutes les classes Java utilisées mais au moins les paquetages aux quels elles appartiennent.

- Le paquetage java.awt.* :

  Pour la création des composants graphiques (Frame, MenuBar, Button…etc.) et la gestion des événements relatifs à ses derniers.

- Le paquetage java.sql.* :

  Pour l'établissement de la Connexion(l'interface *Connection*), l'exécution des requêtes (l'interface *Statement, PreparedStatement, CallableStatement*) et récupération des résultats(l'interface *ResultSet*).

- Le paquetage java.util.* :

  Ce paquetage propose une série d'utilitaires.

3.2. 2ème Version : Une Application

La deuxième version est une Application Java utilisant l'API JDBC avec ses mêmes mécanismes décrits auparavant et s'exécutant sur une machine monoposte, ayant pour seule différence avec l'Applet, la méthode ***main().***

En effet, une Application Java élémentaire comporte une classe dotée d'une méthode ***main()*** (qui joue le même rôle que la méthode *init()* pour les Applets, -décrite au premier chapitre-) dont la signature est caractéristique.

A l'exécution de l'application, la méthode *main()*est détectée et exécutée. Elle sert à mettre en place le comportement nécessaire au bon déroulement du programme.

A la différence des Applets qui sont exécutées au sein d'un navigateur, une Application Java s'exécute en se servant seulement de l'interpréteur Java à partir d'une ligne de commande.

3.2.1. Architecture :

L'architecture de notre application s'exécutant sur une machine monoposte se présente comme suit (Fig. n°4.7) :

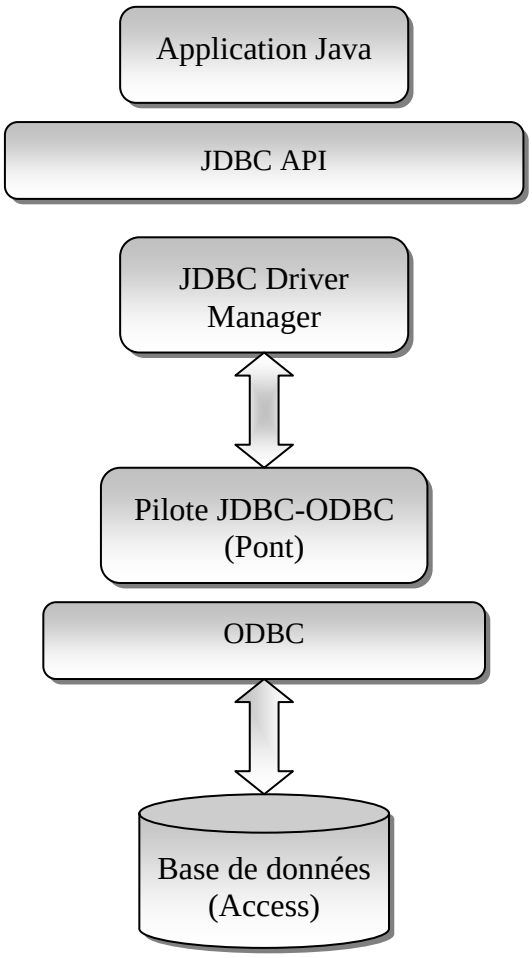


Fig. n°4.7: Architecture d'une application Java monoposte utilisant JDBC.

3.2.2. Etapes suivies dans la réalisation :

Il faudra noter que les étapes suivies au cours de la réalisation sont les mêmes que celles décrites dans la section précédente à savoir : la préparation de la base de données et la configuration de la source de données.

a- L'application :

La seule différence entre l'application et l'applet réside dans l'écriture de la classe principale, le code concernant la connexion, l'interaction avec la base et la libération des ressources étant le même.

La déclaration de la classe aura la forme suivante :

```
 public class MyMainClass {

                //…….
                //……

        public static void  main (String args[ ]) {    // main() qui joue le rôle de init() dans l'applet
                        //…..
        }
}
```

au lieu de celle-ci si c'était une Applet :

```
                public class MyMainClass extends java.applet.Applet {

                    //……..
                    //…….

                  public void init(){

                      //qui joue le même rôle que main() dans l'application
                 }

                  public void start(){
                     //pour le démarrage du thread et autres
                 }

                  public void stop(){
                      //pour le stoppage du thread
                 }

                 public void destroy(){
                     //pour la liberation des ressources
                 }
            }
```

//il faut noter que la redéfinition de start(), stop(), destroy() et paint() n'est obligatoire

Enfin il ne nous reste plus qu'a visualisé le résultat dans une fenêtre de Netscape Communicator (V4.5)

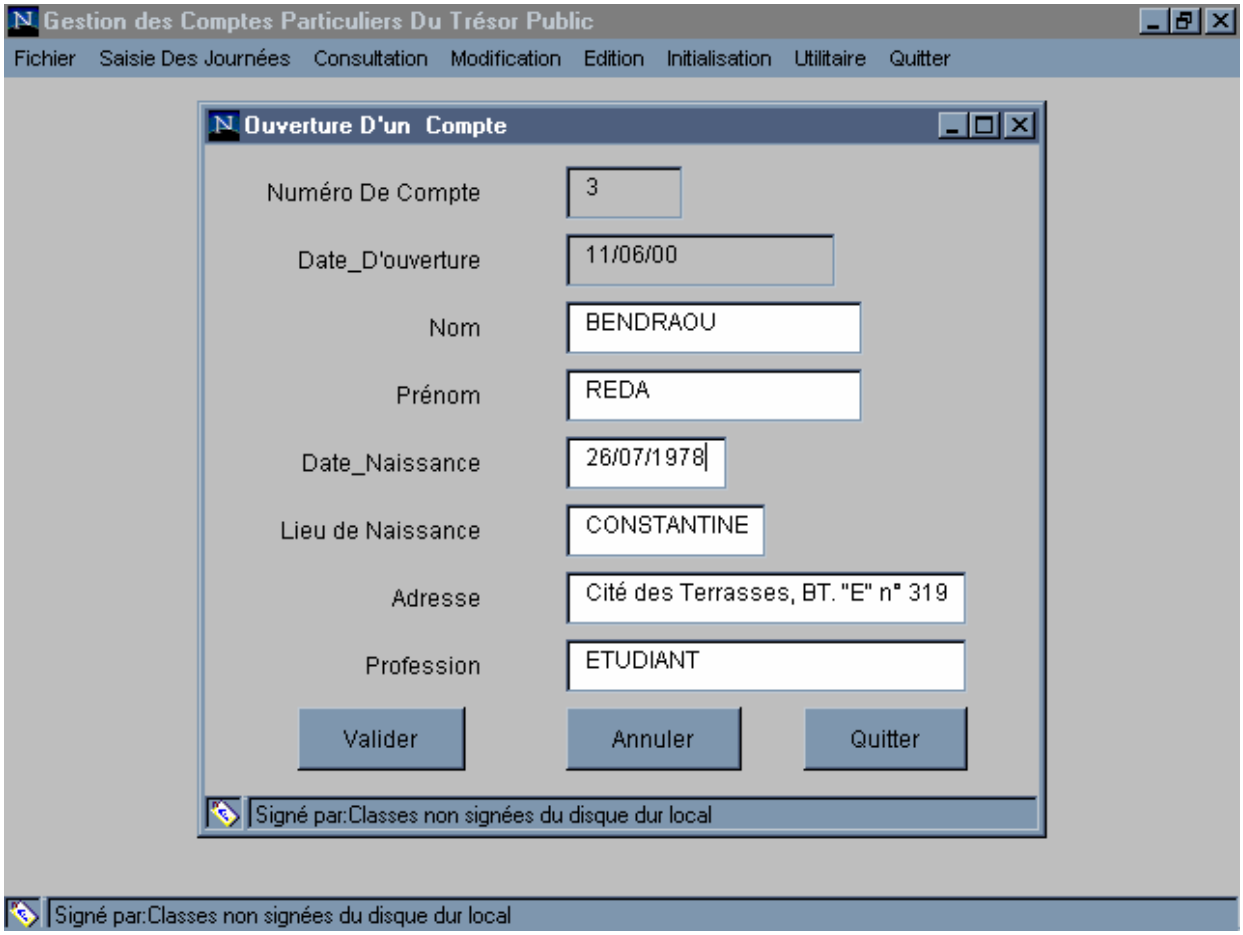


Fig. n° 4.8 : Une vue du résultat obtenu dans Netscape Communicator (version. 4.5).

4. Conclusion :

Après présentation des différentes étapes nécessaires à la réalisation d'une application de type Intranet permettant l'accès aux bases de données a travers une Applet dans une architecture deux niveaux, nous pensons qu'une bonne analyse et une maîtrise des concepts liée à java en général et à l'Api JDBC en particulier reste le moyen le plus sûr qui permettra d'atteindre les objectifs en termes de fonctionnalités.

Nous soulignons que pour une application de type Intranet un pilote de type 3 utilisant les mécanismes propres a RMI est indispensable afin de permettre l'interaction avec les bases de données et l'invocation d'objets distants ; certes leurs manipulation reste complexe mais une fois bien configurer il ne devrais pas y avoir de problèmes.

Une architecture à deux niveaux reste indiquée pour des projet d'entreprise de petite envergure et il faudra envisager d'utiliser une architecture a trois niveaux si le projet et plus complexe.

En ce qui concerne l'application sur une machine mono poste, elle reste relativement facile à réaliser, la configuration de la source de données étant simplifier.

Conclusion :

De par les différents aspects et concepts que nous avons détaillés au cours de cette étude, il est claire que Java a devant lui un avenir très prometteur.

Sa portabilité, sa conception Orientée Objet et sa simplicité ont fait de lui un atout de taille pour le développement d'applications Internet (multi-plate-formes).

Il est clair que l'API JDBC apporte beaucoup de perspectives aux développeurs d'applications de nature distribuées. Néanmoins nous rappelons que l'API JDBC n'est qu'à ses débuts et promet des fonctionnalités plus évoluées dans les versions à venir.

La conception d'un Système d'Information relatif à la gestion du PORTEFEUILLE au Trésor Public de Constantine nous a apportée l'assurance d'une base de données cohérente et par la même occasion, une maîtrise des concepts propre à MERISE.

Enfin, la mise en œuvre de notre application nous a amener à penser, qu'en héritant du meilleur de ses prédécesseurs (le langage C) et par sa portabilité qui n'est plus à démontrer, que Java est prêt à affronter les exigences imposées par les systèmes d'information les plus complexes et notamment à offrir une réactivité au changement toujours constante.

Ses propriétés font de Java un langage sûr et fiable et de ce fait, il pourra répondre aux besoins des applications de type Intranet/Internet mais aussi aux besoins du Client/Serveur plus traditionnels.

L'étape suivante serait de développer la même application sous forme de composants "*Java Beans* " ce qui faciliterait la réutilisation de chacun des composants ici présents et permettra au langage Java d'offrir un large éventail de possibilités. Ainsi, grâce à ce langage, le problème de portabilité posé par le modèle Client/Serveur est résolu.

Etant donner que Microsoft Access n'est pas vraiment adapté à une utilisation en réseau, une autre perspective serait d'adapter notre système avec le SGBD "*Oracle 8* " et de mettre en place une architecture de type Intranet destinée aux entreprises.

Bibliographie

[1] Bradley F . Burton and Victor W . Marek, Application of Java Programming Language to database Management, SIGMOD Record, Vol. 27, N° 1, March 1998.

[2] C. NICOLAS, C. Avare, F. Najman, JAVA Client-Serveur, EYROLLES 1998, 3ème T.

[3] D. Banos, M. Mouyssinat, de MERISE aux Bases de Données : SGBD : Réseau, hiérarchique, relationnel, EYROLLES 1991.

[4] F. Austermûhl et A. Rosskamp, ACCESS 97, Micro Application 1997.

[5] J. Hugues, A. Collongues, B. Laroche, MERISE : Méthode de Conception, DUNOD 1989.

[6] J. P. Matheron, Comprendre MERISE : Outils conceptuels et organisationnels, EYROLLES 3ème T 1996.

[7] L. Lemay et C. L. Perkins, Apprenez JAVA 1.1 en 21 jours, Simon & Schuster Macmillan 1997.

[8] M. Makarévitch, JDBC et Java : Guide du Programmeur, O'REILLY 1998.

[9] M. Bonjour, G. Falquet, J. Guyot, A. le Grand, JAVA : De l'esprit à la méthode : Distribution d'applications sur Internet, THOMSON Publishing 1996.

[10] P. Sridharan, Advanced JAVA networking, Prentice Hall PTR 1997.

[11] R. Werle, HTML, Micro Application 1996.

Ressources Internet

[12] Java Development Kit : http://www.javasoft.com/products/jdk/1.1/

[13] JDBC : http://www.javasoft.com/products/jdk/1.1/docs/guide/jdbc/index.html

[14] Netscape (Version 4.5) : http://home.fr.netscape.com/fr/

[15] Pilote JDBC-ODBC Bridge 2000 easySoft : http://www.easysoft.com/

[16] Pilote RmiJdbc : http://dyade.inrialpes.fr/mediation/download/RmiJdbc/

[17] Sun MicroSystem : http://www.sun.com/

Conclusion :

De par les différents aspects et concepts que nous avons détaillés au cours de cette étude, il est claire que Java a devant lui un avenir très prometteur.

Sa portabilité, sa conception Orientée Objet et sa simplicité ont fait de lui un atout de taille pour le développement d'applications Internet (multi-plate-formes).

Il est clair que l'API JDBC apporte beaucoup de perspectives aux développeurs d'applications de nature distribuées. Néanmoins nous rappelons que l'API JDBC n'est qu'à ses débuts et promet des fonctionnalités plus évoluées dans les versions à venir.

La conception d'un Système d'Information relatif à la gestion du PORTEFEUILLE au Trésor Public de Constantine nous a apportée l'assurance d'une base de données cohérente et par la même occasion, une maîtrise des concepts propre à MERISE.

Enfin, la mise en œuvre de notre application nous a amené à penser, qu'en héritant du meilleur de ses prédécesseurs (le langage C) et par sa portabilité qui n'est plus à démontrer, que Java est prêt à affronter les exigences imposées par les systèmes d'information les plus complexes et notamment à offrir une réactivité au changement toujours constante.

Ses propriétés font de Java un langage sûr et fiable et de ce fait, il pourra répondre aux besoins des applications de type Intranet/Internet mais aussi aux besoins du Client/Serveur plus traditionnels.

L'étape suivante serait de développer la même application sous forme de composants "*Java Beans* " ce qui faciliterait la réutilisation de chacun des composants ici présents et permettra au langage Java d'offrir un large éventail de possibilités. Ainsi, grâce à ce langage, le problème de portabilité posé par le modèle Client/Serveur est résolu.

Etant donner que Microsoft Access n'est pas vraiment adapté à une utilisation en réseau, une autre perspective serait d'adapter notre système avec le SGBD "*Oracle 8* " et de mettre en place une architecture de type Intranet destinée aux entreprises.