CENTRE INTERUNIVERSITAIRE DE RECHERCHE
PLURIDISCIPLINAIRE (CIREP)
STATUT : UNIVERSITE PUBLIQUE
Web : www.cirep.ac.cd
Email : info@cirep.ac.cd

NOTES DE COURS D'INFORMATIQUE 5

OBJECTIFS DU COURS

Objectif général :

Le cours vise à fournir aux étudiants une compréhension approfondie des principes fondamentaux de l'informatique, des technologies de l'information et de la programmation, ainsi que des applications pratiques dans divers domaines.

Objectifs spécifiques du cours :

- ✓ Maîtriser les langages de programmation couramment utilisés et être capable de développer des applications informatiques simples.
- ✓ Acquérir des compétences en résolution de problèmes informatiques et en analyse de données.
- ✓ Développer des compétences en gestion de projet informatique et en collaboration au sein d'équipes techniques.
- ✓ Être capable de suivre les évolutions technologiques et d'adapter ses compétences en conséquence.

Introduction à la programmation

L'informatique est avant tout une science de l'abstraction — il s'agit de créer le bon modèle pour un problème et d'imaginer les bonnes techniques automatisables et appro- priées pour le résoudre. Toutes les autres sciences considèrent l'univers tel qu'il est. Par exemple, le travail d'un physicien est de comprendre le monde et non pas d'inventer un monde dans lequel les lois de la physique seraient plus simples et auxquelles il serait plus agréable de se conformer. À l'opposé, les informaticiens doivent créer des abstractions des problèmes du monde réel qui pourraient être représentées et manipulées dans un ordinateur.

1.1 Informatique = mécanisation de l'abstraction

Un *programme* est une suite d'instructions définissant des opérations à réaliser sur des données. Les instructions du programme sont exécutées les unes après les autres, le plus souvent dans l'ordre séquentiel dans lequel elles sont données dans le programme – on dit que le flot d'exécution, ou flot de contrôle, est séquentiel. Pour écrire des programmes, on se sert d'une notation, appelée *langage de programmation*. Les langages de programmation les plus rudimentaires sont ceux qui « collent » au jeu d'instructions de l'ordinateur ; on les appelle les langages d'assemblage (cf. annexe A). Un langage d'assemblage est par définition propre à un processeur donné (ou au mieux à une famille de processeurs) ; l'évolution de la programmation vers des projets à la fois complexes et portables a créé le besoin de langages de plus haut niveau. Ces langages permettent de gérer la complexité des problèmes traités grâce à la *structuration*, qu'on définira dans un premier temps, de manière très générale, comme le regroupement d'entités élémentaires en entités plus complexes (enregis- trements regroupant plusieurs données, modules ou procédures regroupant plusieurs instructions élémentaires...).

Un bon programme doit être facile à écrire, à lire, à comprendre et à modifier. Il faut savoir que la mémoire humaine est limitée par le nombre plus que par la taille des structures à manipuler. Il devient donc vite difficile pour un programmeur de maîtriser de nombreuses lignes de code, si celles-ci ne sont pas regroupées en modules ou autres structures.

Les avantages de l'utilisation d'un langage de haut niveau sont multiples :

- Un programme n'est pas seulement destiné à être lu une fois ; plusieurs intervenants risquent de se pencher sur lui au cours de son existence, d'où la nécessité de la lisibilité.
- L'existence de langages évolués a permis la création de logiciels par des programmeurs qui n'auraient jamais appris un langage d'assemblage. Il faut noter à ce propos la puissance et la popularité croissante des langages dits de scriptage (Visual Basic, TCL/TK...) et des générateurs de programmes de gestion.
- Un programme en langage évolué peut être « porté » sur différentes architectures et réutilisé au fil de l'évolution du matériel.
- Un module clairement spécifié peut être réutilisé comme une brique de construction, dans divers contextes.
- Un langage peut choisir de laisser transparaître la machine sous-jacente, ou au contraire four- nir un autre modèle de calcul (cf. parallélisme par exemple, même s'il y a un seul processeur physique, ainsi que la programmation fonctionnelle ou la programmation logique).
- Les langages de programmation offrent des outils pour l'abstraction, la modélisation et la structuration des données et des opérations. Ils permettent aussi la *vérification de type*, qui évite beaucoup d'erreurs (elle permet par exemple de refuser un programme qui voudrait additionner 1 et "z").

1.2 Traduction

À partir du moment où l'on utilise un langage différent de celui de la machine, il devient nécessaire de lui traduire les programmes écrits dans ce langage. Deux approches sont possibles pour cela. La première consiste à lancer un programme de traduction simultanée, appelé *interprète*, qui traduit et exécute au fur et à mesure les instructions du programme à exécuter. La deuxième approche consiste à analyser l'ensemble du programme et à le traduire d'avance en un programme en langage machine, qui est ensuite directement exécutable. C'est la *compilation*.

Java fait partie des langages qui choisissent une approche hybride : un programme en Java est analysé et compilé, mais pas dans le langage de la machine physique. Pour des raisons de portabilité, le compilateur Java

traduit dans un langage intermédiaire, universel et portable, le *byte-code*, qui est le langage d'une machine virtuelle, la JVM (*Java Virtual Machine*). Cette « machine » est exécutée sur une machine physique et un interprète le *byte-code*.

Pour traduire un langage, il faut tenir compte de ses trois niveaux :

- le niveau *lexical*, qui concerne le vocabulaire du langage, les règles d'écriture des mots du langage (identificateurs de variables ou fonctions), les mots clés, c'est-à-dire les éléments de structuration du discours, et les caractères spéciaux, utilisés par exemple pour les opérateurs ;
- le niveau *syntaxique*, qui spécifie la manière de construire des programmes dans ce langage, autrement dit les règles grammaticales propres au langage ;
- le niveau *sémantique*, qui spécifie la signification de la notation.

1.3 La programmation : du problème au programme

Nous attaquons ici toute la problématique de la programmation. On peut distinguer les étapes suivantes dans la vie d'un logiciel [1] :

- Définition du problème et spécification, incluant éventuellement plusieurs itérations de spé- cification avec les futurs utilisateurs du logiciel, un prototypage du produit final, et une modélisation des données.
- Conception, avec création de l'architecture de haut niveau du système, réutilisant si possible des composants déjà disponibles.
- Réalisation des composants et test de chacun d'eux pour vérifier qu'ils se comportent comme spécifié.
- Intégration et test du système : une fois chaque composant testé et validé, encore faut-il que le système intégré le soit.
- Installation et test « sur le terrain », c'est-à-dire en situation réelle.
- Maintenance : elle représente souvent plus de la moitié du coût de développement. Le système doit être purgé d'effets imprévisibles ou imprévus, ses performances doivent être corrigées ou améliorées, le monde réel dans lequel il est implanté n'étant pas immuable, il faut modifier le programme ou lui ajouter de nouvelles fonctionnalités, le système doit souvent être porté sur de nouvelles plateformes physiques, etc. Tous ces aspects de maintenance soulignent

l'importance d'écrire des programmes lisibles, corrects, robustes, efficaces, modifiables et portables !

Nous voyons donc que l'activité d'analyse et de programmation, bien qu'elle soit au cœur de notre cours, ne représente qu'une des étapes dans le cycle logiciel. Ceci étant, nous allons essentiellement nous arrêter sur les phases qui permettent d'aller d'un problème bien spécifié à un programme, en passant par la réalisation d'un algorithme précis, c'est-à-dire de la conception d'une démarche opératoire pour résoudre le problème donné, cette démarche étant ensuite transcrite dans un langage de programmation, en l'occurrence Java.

Programmation objet

Les méthodes d'analyse qui vont de pair consistent à *diviser pour régner*, c'est-à-dire à découper la tâche à effectuer en un ensemble de modules indépendants, considérés comme des boîtes noires. Cette technique a fait ses preuves depuis longtemps et continue à être à la base de la programmation structurée. Mais elle atteint ses limites lorsque l'univers sur lequel opèrent les programmes évolue, ce qui est en fait la régle générale plutôt que l'exception : de nouveaux types de données doivent être pris en compte, le contexte applicatif dans lequel s'inscrit le programme change, les utilisateurs du programme demandent de nouvelles fonctionnalités et l'interopérabilité du pro- gramme avec d'autres programmes, etc. Or dans un langage comme Pascal ou C, les applications sont découpées en procédures et en fonctions ; cela permet une bonne décomposition des traite- ments à effectuer, mais le moindre changement de la structuration des *données* est difficile à mettre en œuvre, et peut entraîner de profonds bouleversements dans l'organisation de ces procédures et fonctions.

C'est ici que la programmation objet apporte un « plus » fondamental, grâce à la notion d'*encapsulation* : les données et les procédures qui manipulent ces données sont regroupées dans une même entité, appelée l'*objet*. Les détails d'implantation de l'objet restent cachés : le monde extérieur n'a accès à ses données que par l'intermédiaire d'un ensemble d'opérations qui constituent l'*interface* de l'objet. Le programmeur qui utilise un objet dans son programme n'a donc pas à se soucier de sa représentation physique ; il peut raisonner en termes d'abstractions.

Java est l'un des principaux représentants actuels de la famille des

langages à objets. Dans ce chapitre, nous allons nous familiariser progressivement avec les principales caractéristiques de cette famille, en nous appuyant comme dans les chapitres précédents sur le langage Java pour illustrer notre propos. Mais il faut savoir qu'à certains choix conceptuels près, on retrouvera le même style de programmation dans d'autres langages de la famille.

Modèles et structures de données

Un *modèle de données* est une abstraction utilisée pour décrire des problèmes. Cette abstraction spécifie les valeurs qui peuvent être attribuées aux objets, et les opérations valides sur ces objets. La *structure de données*, quant à elle, est une construction du langage de programmation,
permettant de représenter un modèle de données.

Dans ce chapitre, nous allons illustrer la différence entre ces deux notions en parlant assez longuement de la *liste* (modèle de données), et des structures de données qui permettent habituel- lement de la représenter. Cela nous donnera aussi l'occasion d'introduire la notion d'*interface*, telle qu'elle est définie en Java. Nous verrons aussi plus rapidement quelques autres modèles de données classiques.

Les listes

La liste est un modèle de données qui permet de décrire une séquence d'objets contenant chacun une valeur d'un type donné. À la base, on peut définir le type L = *liste de* V à partir d'un type V donné, comme une suite finie d'éléments de V.

En plus de ces opérations de base, on peut souhaiter définir des opérations plus complexes – qui s'expriment en termes de combinaison de ces opérations de base – comme l'adjonction ou la suppression à un endroit quelconque, la recherche de l'existence d'une valeur dans la liste, voire l'adjonction et la suppression en queue ou le tri...

Programmation (un peu) plusavancée

Portée

La portée d'une variable est le bloc de code au sein de laquelle cette variable est accessible ; la portée détermine également le moment où la variable est créée, et quand elle est détruite.

Les variables d'instance et variables de classe sont visibles et accessibles directement à l'intérieur de l'ensemble de la classe. Par exemple, la variable tabComptes est accessible dans l'ensemble de la classe AgenceBancaire, et nous avons bien entendu profité largement de cette visibilité pour l'utiliser dans les méthodes de cette classe. Si une variable de classe ou d'instance est déclarée publique, elle est également accessible de l'extérieur de la classe, mais en employant la notation pointée.

Espaces de nommage : les packages

On retrouve la notion de *bibliothèque* dans la grande majorité des langages de programmation. Pour beaucoup d'entre eux, la définition du langage est d'ailleurs accompagnée d'une définition normalisée de la bibliothèque de base, ensemble de fonctions fournissant les services fondamentaux tels que les entrées-sorties, les calculs mathématiques, etc. Viennent s'y ajouter des bibliothèques quasiment «standard » pour l'interface utilisateur, le graphisme, etc. D'autres bibliothèques accompagnent des produits logiciels. Enfin, certaines bibliothèques peuvent elles-mêmes être des produits logiciels, commercialisés en tant que tels.

Avec la notion de *package*, Java étend et modifie un peu, mais généralise aussi, le concept de bibliothèque.

La problématique du *nommage* est liée à celle de l'utilisation de bibliothèques. Si on peut théoriquement garantir l'absence de conflits de nommage quand un seul programmeur développe une application, c'est déjà plus difficile quand une équipe travaille ensemble sur un produit logiciel, et cela nécessite des conventions ou constructions explicites quand on utilise des bibliothèques en provenance de tiers.

De ce point de vue, Java systématise l'emploi des *packages*, qui sont entre autres des espaces de nommage permettant de réduire fortement les conflits de nommage et les ambiguités. Toute classe doit appartenir à un *package* ; si on n'en indique pas, comme cela a été le cas jusqu'à maintenant pour tous nos programmes, la classe est ajoutée à un *package* par défaut.

L'organisation hiérarchique en packages est traduite à la compilation en une organisation hié- rarchique équivalente des répertoires, sous la racine indiquée comme le « réceptacle » de vos classes Java. La

recommandation est faite d'utiliser pour ses noms de *packages* une hiérarchisation cal- quée sur celle des domaines Internet. Ainsi, les classes que j'écris pour les corrigés de mon cours devraient être déclarées dans le package fr.inpl-nancy.mines.tombre.ens.cours1A si je décide d'organiser mes programmes d'enseignement dans la catégorie *ens*.

Introduction sommaire au monde desbases de données

On doit être préoccupé uniquement de l'impression ou de l'idée à traduire. Les yeux de l'esprit sont tournés au dedans, il faut s'efforcer de rendre avec la plus grande fidélité possible le modèle intérieur. Un seul trait ajouté (pour briller, ou pour ne pas trop briller, pour obéir à un vain désir d'étonner, ou à l'enfantine volonté de rester classique) suffit à compromettre le succès de l'expérience et la découverte d'une loi. On n'a pas trop de toutes ses forces de soumission au réel, pour arriver à faire passer l'impression la plus simple en apparence, du monde de l'invisible dans celui si différent du concret où l'ineffable se résout en claires formules.

Nous avons abordé la notion de modèle de données, que nous avons définie comme une *abstraction* d'un type donné d'information, spécifiant les valeurs que peut prendre cette information, et les opérations qui permettent de la manipuler. Si les langages de programmation tels que Java offrent des mécanismes de représentation permettant de construire des structures de données et des classes représentant certains modèles de données, ils manquent cependant de souplesse quand les informations à manipuler comportent de nombreuses relations entre elles, par exemple. Nous avons vu au chapitre 6 comment la représentation d'un ensemble de comptes bancaires – problème que nous avons pourtant beaucoup simplifié par rapport à la réalité – nécessite la mise en place de fichiers, de mécanismes de sauvegarde et de chargement, etc. On imagine aisément que dans la réalité, on ne s'amuse pas, pour chaque application nécessitant la gestion d'informations un peu complexes, à remettre en place de tels mécanismes au moyen de lignes de code telles que nous les avons vues en Java. Au contraire, on recourt alors aux systèmes de gestion de bases de données.

On peut définir sommairement une *base de données* comme un ensemble de données représentant l'information sur un domaine d'application particulier, pour lequel on aura pris soin de spécifier les propriétés de ces données, ainsi que les relations qu'elles ont les unes avec les autres.

Un *système de gestion de bases de données* (SGBD) est un ensemble logiciel qui permet de créer, de gérer, d'enrichir, de maintenir de manière rémanente, et d'interroger efficacement des bases de données. Les SGBD sont des outils génériques, indépendants des domaines d'application des bases de données. Ils fournissent un certain nombre de services :

- capacité de spécifier les informations propres à une base de données, en s'appuyant sur un modèle de base – dans notre cas, nous nous intéresserons au modèle relationnel ;
- gestion de la rémanence des données – les opérations explicites de sauvegarde et récupération que nous avons programmées en Java au chapitre 6 sont fournies en standard par un SGBD,
- possibilité de partage des données entre différents utilisateurs, voire différents logiciels ;
- confidentialité, intégrité et cohérence des données – dans un contexte réel, la gestion des droits d'accès, en consultation ou en modification, aux données d'une base est bien évidemment une contrainte absolue, tout comme l'assurance de la préservation de l'intégrité des données entre deux accès[2] ou de la cohérence entre les différentes informations que comporte la base ;
- possibilité de consultation des données (interrogation de la base), soit par des langages d'accès (nous allons voir dans ce chapitre le langage SQL), soit par le biais d'une page Internet, soit directement à partir d'un programme ;
- capacité de stockage élevée, afin de permettre la gestion de données volumineuses.

De nombreux SGBD existent, de complexité et de capacités variables. On peut citer des produits tels que Oracle dans le haut de gamme, le logiciel Access de Microsoft, plus rudimentaire, mais omniprésent en micro-informatique, ou des logiciels libres tels que MySQL.

Le modèle relationnel

Le modèle relationnel a été proposé en 1970 par Ted Codd, un chercheur chez IBM, sur la base de la théorie des ensembles et de l'algèbre relationnelle. Il correspond à une vision tabulaire des données : les attributs des données, et les relations entre elles, sont représentés dans des tables. Présentons rapidement ses principaux éléments :

- Un *domaine* est un ensemble de valeurs possibles. Chaque valeur du domaine est atomique. Le domaine correspond à la notion de type de données, tel que nous l'avons vue dans les chapitres précédents – mais restreint à des types élémentaires (atomiques).
- Un *schéma de relation* $R(A_1 : D_1, A_2 : D_2, \ldots A_n : D_n)$ permet de regrouper un ensemble d'attributs A_i, qui sont « typés » par leur domaine D_i. Ainsi, on pourra définir le schéma de relation Client(idClient : entier, nom : chaîne(30), prénom : chaîne(30)). Pour chaque schéma, on peut avoir un ensemble de *relations*, c'est-à-dire d'«instances » du schéma, par exemple :

idClient	nom	Prénom
27384	Tombre	Karl
39827	Bonfante	Guillaume
73625	Lamiroy	Bart
98362	Chirac	Jacques
99375	Gates	Bill

  Certaines valeurs peuvent prendre une valeur spéciale nulle qui indique qu'il n'y a pas d'in- formation pour cet attribut.
- Chaque relation est caractérisée de manière unique par une ou plusieurs *clés*, qui sont des sous-ensembles d'attributs tels qu'il ne peut pas y avoir deux relations identiques avec les mêmes valeurs pour ce sous-ensemble. Parmi les clés possibles, on choisit une *clé primaire* – dans notre exemple, il s'agira assez naturellement de l'attribut idClient.

- Un *schéma de base de données* est un ensemble de schémas de relations assorti d'un ensemble de contraintes, dites contraintes d'intégrité. Une contrainte d'intégrité est une propriété, par exemple l'interdiction pour un attribut donné de prendre une valeur nulle, ou – et c'est là que les choses deviennent intéressantes – l'obligation pour une valeur d'attribut dans une première relation R_1 d'apparaître comme valeur de clé dans une autre relation R_2.
  Ainsi, on si on définit un schéma de relation Compte(numCompte : entier, idClient :
  entier, solde : réel), on pourra donner comme contrainte que toute relation instanciée doit correspondre à un client existant parmi les relations du schéma Client.
  On dispose ainsi d'un ensemble d'*instances de base de données*, qui sont des instances des relations définies dans le schéma de la base de données, chaque relation de ces instances respectant les contraintes d'intégrité.

Complétons maintenant notre schéma de base de données exemple, en nous inspirant de l'exemple bancaire qui nous accompagne tout au long de ce polycopié. On peut imaginer les schémas de rela- tion suivants (la clé primaire est mise en exergue pour chaque schéma). Il va sans dire que l'exemple reste très sommaire par rapport à la réalité bancaire. . .

```
Client(idClient : entier, nom : chaîne(30), prénom : chaîne(30))
Compte(numCompte : entier, idClient : entier, solde : réel, codeType : caractère)
Adresse(idClient : entier, rue : chaîne(50), codePostal : entier, commune : chaîne(40))
TypeCompte(code : caractère, nom : chaîne(40), tauxAgios : réel, tauxIntérêts : réel)
```

On pourrait alors avoir les instances suivantes pour les autres schémas de relation (les noms des attributs ont été abrégés pour des questions de place sur la page) :

Client Compte

id	nom	prénom
27384	Tombre	Karl
39827	Bonfante	Guillaume
73625	Lamiroy	Bart
98362	Chirac	Jacques
99375	Gates	Bill

num	id	solde	code
12536	39827	345,40	D
16783	99375	63703,40	E
18374	39827	2500,45	E
26472	98362	3746,23	D
28374	73625	837,23	D
37468	27384	-981,34	D
37470	27384	1345,34	E
48573	73625	1673,87	E
57367	27384	938,34	E

Adresse

id	rue	cp	com
27384	53 Grande rue	54280	Sornéville
39827	3 avenue des tandems	54000	Nancy
73625	Bureau 498 à l'ENSMN	54000	Nancy
98362	Palais de l'Élysée	75000	Paris
99375	1 rue des fenêtres	99999	Seattle

TypeCompte

code	nom	txAg	txInt
D	Dépôts	14,3	0,0
E	Épargne	0,0	4,4

SQL

Introduction

SQL (*Structured Query Language*) est un langage conçu à partir des études sur le modèle relationnel, pour définir et accéder de manière normalisée aux bases de données relationnelles. Il est actuellement supporté par la plupart des SGBD du commerce et fait l'objet d'une norme. On peut donc le considérer comme une référence essentielle dans le domaine des bases de données.

SQL est un langage structuré, permettant d'exprimer de manière simple et lisible (proche de la langue anglaise) des requêtes qui traduisent les opérations de manipulation de données dans le modèle relationnel. Il permet d'une part de définir des schémas de bases de données relationnelles, ainsi que les données qui composent une base de données, et d'autre part de définir des requêtes, c'est-à-dire des manipulations dans une base de données pour en extraire les informations recher- chées.

Création de schémas et insertion de données

Plutôt que de grandes explications théoriques, illustrons l'emploi de

```
CREATE TABLE CLIENT(IDCLIENT INTEGER NOT NULL, NOM CHAR(30), PRENOM CHAR(30),
                    PRIMARY KEY (IDCLIENT))
CREATE TABLE COMPTE(NUMCOMPTE INTEGER NOT NULL, IDCLIENT INTEGER NOT NULL,
                    SOLDE DECIMAL(15,2), CODETYPE CHAR,
                    PRIMARY KEY (NUMCOMPTE))
CREATE TABLE ADRESSE(IDCLIENT INTEGER NOT NULL, RUE CHAR(50), CODEPOSTAL INTEGER,
                     COMMUNE CHAR(40),
                     PRIMARY KEY (IDCLIENT))
CREATE TABLE TYPECOMPTE(CODE CHAR NOT NULL, NOM CHAR(40), TAUXAGIOS DECIMAL(4,2),
                        TAUXINTERETS DECIMAL(4,2),
                        PRIMARY KEY (CODE))
```

ce langage en créant tout d'abord les tables données précédemment, grâce à l'instruction CREATE TABLE. À noter que pour détruire une table, on utilise l'instruction DROP, et que pour modifier un schéma de relation, on utilise l'instruction ALTER TABLE.

Peuplons maintenant ces tables avec l'instruction INSERT (l'instruction UPDATE permet ulté- rieurement de modifier ces valeurs, et l'instruction DELETE de supprimer certaines relations d'une table – dans les deux cas, ces instruction se servent de la clause WHERE, que nous illustrons par la suite) :

```
INSERT INTO CLIENT VALUES(27384, "Tombre", "Karl")
INSERT INTO CLIENT VALUES(39827, "Bonfante", "Guillaume")
INSERT INTO CLIENT VALUES(73625, "Lamiroy", "Bart")
INSERT INTO CLIENT VALUES(98362, "Chirac", "Jacques")
INSERT INTO CLIENT VALUES(99375, "Gates", "Bill")

INSERT INTO COMPTE VALUES(12536, 39827, 345.40, 'D')
INSERT INTO COMPTE VALUES(16783, 99375, 63703.40, 'E')
INSERT INTO COMPTE VALUES(18374, 39827, 2500.45, 'E')
INSERT INTO COMPTE VALUES(26472, 98362, 3746.23, 'D')
INSERT INTO COMPTE VALUES(28374, 73625, 837.23, 'D')
INSERT INTO COMPTE VALUES(37468, 27384, -981.34, 'D')
INSERT INTO COMPTE VALUES(37470, 27384, 1345.34, 'E')
INSERT INTO COMPTE VALUES(48573, 73625, 1673.87, 'E')
INSERT INTO COMPTE VALUES(57367, 27384, 938.34, 'E')

INSERT INTO ADRESSE VALUES(27384, "53 Grande rue", 54280, "Sornéville")
INSERT INTO ADRESSE VALUES(39827, "3 avenue des tandems", 54000, "Nancy")
INSERT INTO ADRESSE VALUES(73625, "Bureau 498 à l'ENSMN", 54000, "Nancy")
INSERT INTO ADRESSE VALUES(98362, "Palais de l'Élysée", 75000, "Paris")
INSERT INTO ADRESSE VALUES(99375, "1 rue des fenêtres", 99999, "Seattle")

INSERT INTO TYPECOMPTE VALUES('D', "Dépôts", 14.3, 0.0)
INSERT INTO TYPECOMPTE VALUES('E', "Épargne", 0.0, 4.4)
```

Projections et sélections

Passons maintenant à la manipulation de données. On exprime habituellement une requête de la manière suivante : SELECT ... FROM

... WHERE La clause SELECT indique que l'on effectue une opération sur la base de données, en lui appliquant certains opérateurs ou fonctions, afin d'obtenir un résultat. La clause FROM donne la liste des tables utilisées dans l'opération en cours. La clause optionnelle WHERE, quant à elle, donne une liste de conditions que doivent respecter les données sélectionnées.

L'opération la plus simple consiste à faire une *projection*, c'est-à-dire d'extraire un ou plusieurs attributs d'un schéma. Ainsi, voici une requête simple de type projection, et le résultat obtenu sur l'exemple développé :

```
SELECT CODEPOSTAL, COMMUNE FROM ADRESSE

54280, Sornéville
54000, Nancy
54000, Nancy
75000, Paris
99999, Seattle
```

Un cas particulier est le caractère *, qui correspond à l'extraction de tous les attributs :

```
SELECT * FROM TYPECOMPTE

D, Dépôts, 14.30, 0.00
E, Épargne, 0.00, 4.40
```

La deuxième opération est la *sélection*, dans laquelle on limite la projection aux données qui vérifient une certaine condition (spécifiée grâce à la clause WHERE). Ainsi, pour extraire les identi- ficateurs de tous les clients qui habitent à Nancy, on pourra écrire :

```
SELECT IDCLIENT FROM ADRESSE WHERE COMMUNE="NANCY"

39827
73625
```

La présence de plusieurs lignes identiques dans le résultat ci-dessus n'est pas une erreur, mais résulte du produit cartésien, qui a trouvé

```
SELECT DISTINCT NOM, CODEPOSTAL, COMMUNE FROM CLIENT, COMPTE, ADRESSE
  WHERE CLIENT.IDCLIENT = ADRESSE.IDCLIENT AND CLIENT.IDCLIENT = COMPTE.IDCLIENT
  AND CODETYPE='E'

Gates, 99999, Seattle
Bonfante, 54000, Nancy
Tombre, 54280, Sornéville
Lamiroy, 54000, Nancy
```

plusieurs comptes d'épargne pour le même client. Si on souhaite éliminer de tels doublons, on peut utiliser la clause DISTINCT :

Quelques autres clauses et fonctions

Une fois de plus, nous n'avons pas la prétention de donner un cours complet sur les bases de données relationnelles, ni même sur SQL, mais simplement de vous faire goûter à la puissance de ce type de modèle et de langage. Nous illustrons donc sans beaucoup de commentaires quelques autres constructions possibles en SQL...

Pour connaître le nombre de comptes de chaque type, on peut utiliser la

```
SELECT NOM, COUNT(*) FROM COMPTE, TYPECOMPTE
  WHERE COMPTE.CODETYPE = TYPECOMPTE.CODE
  GROUP BY CODE

Dépôts, 4
Épargne, 5
```

fonction COUNT, associée à la clause GROUP BY pour regrouper les comptes de même type dans le produit cartésien :

Pour connaître le nombre de clients dans les différentes communes de Meurthe-et-Moselle, on pourra écrire

```
SELECT COMMUNE, COUNT(*) FROM CLIENT, ADRESSE
  WHERE CLIENT.IDCLIENT = ADRESSE.IDCLIENT
  AND CODEPOSTAL >= 54000 AND CODEPOSTAL < 55000
  GROUP BY COMMUNE

Nancy, 2
Sornéville, 1
```

```
SELECT COMMUNE, COUNT(*) FROM CLIENT, ADRESSE
  WHERE CLIENT.IDCLIENT = ADRESSE.IDCLIENT
  AND CODEPOSTAL >= 54000 AND CODEPOSTAL < 55000
  GROUP BY COMMUNE
  HAVING COUNT(*) > 1

Nancy, 2
```

Si maintenant on veut limiter la recherche précédente aux communes ayant plus d'un client, on ne peut plus recourir simplement à la clause WHERE, car ce genre de condition ne porte pas sur les lignes individuelles sélectionnées, mais sur les groupes constitués par la clause GROUP BY. C'est à cela que sert la clause HAVING :

Pour calculer le solde cumulé de tous les comptes de chaque client, on utilisera la fonction SUM

- à noter que d'autres fonctions existent, notamment MAX, MIN et AVG :

```
SELECT PRENOM, NOM, SUM(SOLDE) FROM CLIENT, COMPTE
  WHERE CLIENT.IDCLIENT = COMPTE.IDCLIENT
  GROUP BY CLIENT.IDCLIENT

Karl, Tombre, 1302.34
Guillaume, Bonfante, 2845.85
Bart, Lamiroy, 2511.10
Jacques, Chirac, 3746.23
Bill, Gates, 63703.40
```

Enfin, si je veux présenter la liste ci-dessus dans l'ordre alphabétique des noms de famille, on peut ajouter la clause ORDER BY :

```
SELECT PRENOM, NOM, SUM(SOLDE) FROM CLIENT, COMPTE
  WHERE CLIENT.IDCLIENT = COMPTE.IDCLIENT
  GROUP BY CLIENT.IDCLIENT
  ORDER BY NOM

Guillaume, Bonfante, 2845.85
Jacques, Chirac, 3746.23
Bill, Gates, 63703.40
Bart, Lamiroy, 2511.10
Karl, Tombre, 1302.34
```

JDBC

Jusqu'ici, nous avons vu les bases de données de manière complètement déconnectée de la programmation. On peut bien évidemment accéder directement à une base de données *via* un interprète du langage SQL ou par une interface Web, par exemple. Cependant, l'intérêt principal reste bien entendu d'intégrer directement l'accès à ces bases aux programmes écrits par ailleurs pour remplacer en particulier les manipulations lourdes de fichiers illustrées par exemple au § 6.3.2, par une gestion directe des données au moyen d'une base de données.

En Java, l'interface (API) JDBC a été justement conçue pour permettre aux applications écrites en Java de communiquer avec les SGBD, en exploitant notamment SQL. Nous nous contentons ici d'illustrer très

```
import java.sql.*;
```

sommairement l'emploi de cette API dans un programme Java ; le lecteur qui voudrait en savoir plus se reportera à l'un des nombreux ouvrages techniques disponibles, tel que [9]. À noter que les classes JDBC se

trouvent dans le *package* java.sql. On n'oubliera donc pas de commencer par dire :

```
Class.forName("org.gjt.mm.mysql.Driver").newInstance();
```

Il faut noter que le SGBD peut très bien s'exécuter sur une autre machine que celle sur laquelle s'exécute l'application qui l'utilise. Il faut donc examiner ce qui se passe aussi bien du côté du *serveur*, en l'occurrence le SGBD, que du côté du *client*, à savoir l'application. En supposant qu'un SGBD s'exécute bien du côté serveur, il faut un *pilote*[4] pour y accéder. Les pilotes sont bien entendu propres à chaque SGBD ; si on imagine par exemple que l'on accède au SGBD MySql, on invoquera l'un des pilotes disponibles pour ce SGBD. Chaque pilote a ses règles propres pour le lancement ; celui que nous utilisons en TD à l'École, par exemple, est lancé par la commande suivante :

```
try {
    Connection c = DriverManager.getConnection(
        "jdbc:mysql://cesar.mines.inpl-nancy.fr/cours_1a?
        user=cours_1a_user&
        password=mines");
} catch (SQLException err) {
    System.out.println("SQLException: " + err.getMessage());
    System.out.println("SQLState:      " + err.getSQLState());
    System.out.println("VendorError:   " + err.getErrorCode());
}
```

La première chose à effectuer, une fois le pilote lancé, est de se connecter à la base de données. On utilise pour cela une méthode de la classe DriverManager du *package* java.sql, dont le rôle est de tenir à jour une liste des implantations de pilotes et de transmettre à l'application toute information utile pour se connecter aux bases. Ainsi, on pourra écrire : à condition bien entendu que le SGBD tourne bien sur la machine cesar, avec les informations d'authentification données ici. Vous noterez au passage que comme nombre de méthodes des classes du *package* java.sql, getConnection est susceptible de provoquer une exception, et que je prévois d'afficher un maximum d'informations dans ce cas.

La méthode getConnection rend une référence à un objet de type

```
try {
    Statement stmt = c.createStatement();
} catch (SQLException err) { ... }
```

Connection ; cette dernière classe symbolise une transaction avec une base de données. C'est en utilisant une méthode de cette classe que l'on crée un objet de type requête SQL, représenté par la classe Statement :

Nous sommes maintenant prêts pour récupérer le résultat d'une

```
try {
    ResultSet rs = stmt.executeQuery(
    "SELECT * FROM COMPTE, TYPECOMPTE WHERE COMPTE.CODETYPE = TYPECOMPTE.CODE");
} catch (SQLException err) { ... }
```

requête, grâce à une méthode de cette dernière classe :

Vous comprendrez aisément qu'à la place de la chaîne de caractères constante ci-dessus, la requête peut être constituée d'une chaîne construite par le programme d'application, à partir par exemple des données fournies par un utilisateur dans une partie interactive, ou de la logique propre à l'application. Nous vous laissons d'ailleurs le soin de le vérifier dans l'exercice qui vous est proposé dans le TD accompagnant ce cours.

Le résultat de la requête est une instance de la classe ResultSet ; les différents éléments peuvent être récupérés pour affichage, ou pour exploitation dans les calculs propres à l'application, par l'intermédiaire de l'interface de cette classe. Le résultat de la requête est une table, et les méthodes getXXX de la classe ResultSet permettent de lire le contenu des différentes colonnes de cette table, colonnes qui sont numérotées à partir de 1. Ainsi, dans l'exemple donné ci-dessus, si on veut afficher dans l'ordre le type de compte (dépôts ou épargne), le numéro de compte, et le solde, on pourra par exemple écrire :

```
try {
    while (rs.next()) {
        System.out.print("Compte de type " + rs.getString(6) + " numéro ");
        System.out.print(rs.getInt(1) + " -- solde : ");
        System.out.println(rs.getDouble(3));
    }
    // On ferme proprement
    rs.close();
} catch (SQLException err) {
    ...
}
```

ce qui donnera une sortie du genre :

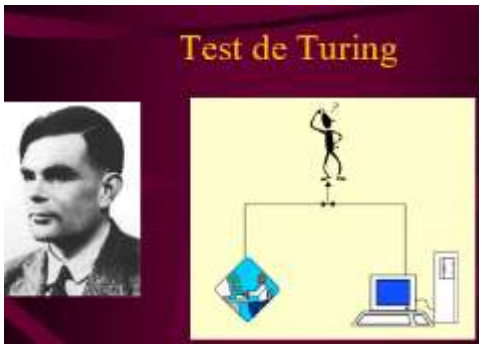
```
Compte de type Dépôts numéro 12536 -- solde : 345.40
Compte de type Dépôts numéro 26472 -- solde : 3746.23
Compte de type Dépôts numéro 28374 -- solde : 837.23
Compte de type Dépôts numéro 37468 -- solde : -981.34
Compte de type Épargne numéro 16783 -- solde : 63703.40
Compte de type Épargne numéro 18374 -- solde : 2500.45
Compte de type Épargne numéro 37470 -- solde : 1345.34
Compte de type Épargne numéro 48573 -- solde : 1673.87
Compte de type Épargne numéro 57367 -- solde : 938.34
```

Introduction à L'Intelligence Artificielle

Définition de l'intelligence artificielle

Le but de l'Intelligence Artificielle (IA) est de concevoir des systèmes capables de reproduire le comportement de l'humain dans ses activités de raisonnement.

L'IA se fixe comme but la modélisation de l'intelligence prise comme phénomène (de même que la physique, la chimie ou la biologie qui ont pour but de modéliser d'autres phénomènes).



Test de Turing (1950)

Considérons trois joueurs

A: Homme, B: Femme, C: Homme ou Femme.

C doit déterminer qui est l'homme et qui est la femme.

A a pour rôle d'induire C en erreur, B a pour rôle de l'aider.

Le test de Turing : (on communique par clavier)

Question : si on faisait jouer le rôle de A à une machine est-ce que C se tromperait aussi souvent ? => Les machines sont-elles capables de penser ?

(Si le comportement de la machine est indistinguable de celui d'un humain alors on pourrait en conclure que la machine est Intelligente)

La chambre chinoise ``Minds, Brains and Programs'' 1980

John Searle (professeur en philosophie a Berkeley) est enfermé dans une pièce ne communiquant avec l'extérieur que par un guichet et contenant un (très) gros livre dans lequel est écrit une succession de questions et de réponses pertinentes à ces questions, et rédigées en

chinois.

Searle précise qu'il ne connaît rien au chinois et que l'anglais est sa langue maternelle.

Un expérimentateur lui transmet des messages par le guichet, tantôt en anglais, tantôt en chinois.

Searle répond directement aux messages rédigés en anglais alors que pour ceux rédigés en chinois, il est obligé de consulter le livre jusqu'à trouver une question identique au message ; il recopie alors la réponse associée.

Exemple : Le jeu d'échèc

Dans le premier cas, on essaiera avant tout d'obtenir un programme efficace. Peu importe alors que la machine fasse des calculs inaccessibles à l'homme, comme explorer quelques centaines de millions (ou milliards) de positions à la seconde.

Dans le second, on essaiera d'abord de comprendre comment l'homme joue aux échecs.

Pour cela, on interviewera des maîtres, on essaiera de dégager les règles plus ou moins consciemment suivies par les joueurs : tenter d'occuper le centre, de dominer une couleur de cases, etc. Le programme réalisé validera.

Historique de l'IA

La préhistoire 1945-1955

Traduction automatique du langage => problème de représentation et d'extraction des connaissances et problème de rédaction générique.

Intelligence artificielle dans la littérature et le cinéma de science fiction (Films: Odyssée 2001 de Kubrick, IA de Spielberg)

Apparition du mot *robot* pour la première fois en 1923 dans la pièce de theâtre

« R.U.R » (*Rossum's Universal Robots)* écrite par Karel Capek.

En 1950, Isaac Asimov (auteur de Science fiction avec un backround scientifique) propose ses trois *Lois de la robotique*.

Un robot ne doit pas attenter a la vie d'un humain

Un robot doit obeir aux ordres d'un humain sauf si cela contredit la première loi

Un robot doit préserver sa propre existence sauf si cela contredit aux

deux premières lois)

(Film I, Robot avec Will Smith, 2004)

1955-1970

Les débuts 1955-1970

Le terme intelligence artificielle est apparu en 1956 à la rencontre de Minsky, McCarthy, Newell et Simon au collège de Darmouth (New Hampshire, USA).

•Époque de l'enthousiasme absolu (Simon en 1958) : *en moins de dix ans un programme d'échec arrivera au niveau d'un champion du monde et qu'un programme de démonstration automatique de théorème découvrira un théorème mathématique.*

♦ Pourtant, Kasparov n'a été battu par la machine Deep Blue qu'en 1997 !

•Développement de travaux : jeux d'échec, démonstration de théorèmes en géométrie

•Apparition du premier programme le LOGIC THEORIST (démonstration automatique de théorème) en 1956 et du langage IPL1. Apparition des langage Lisp en 1960 par MacCarthy, et Prolog en 1971 par Alan Colmerauer.

• **Eliza** est construit au MIT en 1965 , un système intelligent qui dialogue en anglais et qui joue au psychotherapiste.

•**Le système ELIZA** (de Joseph Weizenbaum, au MIT), en repérant des expressions clés dans des phrases et en reconstruisant à partir d'elles des phrases toutes faites, était capable, dès 1965, de dialoguer en langage naturel en trompant un moment des interlocuteurs qui croyaient avoir affaire à un psychologue humain !

•Pourtant, ELIZA ne détenait aucune compréhension réelle des phrases qu'il traitait. C'est sans doute le système SHRDLU de Terry Winograd qui, en 1970, fut le premier à « comprendre » quelque chose à la langue naturelle et à exploiter cette compréhension dans des dialogues qui portaient sur un monde simplifié fait de blocs.

•1970 : SCHRDLU, logiciel conçu par Terry Winograd. Il simule la manipulation de blocs géométriques (cubes, cylindres, pyramides, ...) posés sur une

•table. Le logiciel génère automatiquement des plans (<< Pour déplacer le cube bleu sur le sommet du cylindre jaune, je dois d'abord enlever la

pyramide qui se trouve sur le cube et ...>>) et est muni d'une interface en langage naturel.

- Les systèmes experts, parmi lesquels :
- 1969 : DENDRAL : analyse des résultats d'une spectrographie de masse 1967 : MACSYMA (logiciel de calcul formel)
- 1977 : MYCIN (maladies infectieuses).
- HEARSAY-II en compréhension de la parole, PROSPECTOR en géologie.
- **La spécialisation 1970-1980 (spécialisation et théorisation)**
- L'IA est le carrefour de plusieurs disciplines : informatique, logique, linguistique, neurologie et psychologie). Naissance du langage Smalltalk en 80
- Simon recoit le prix Nobel en économie en 1978
- **La reconnaissance 1980-1990 (crédibilité et audience)**
- Projet de cinquième génération par MITI (3 alphabets pour les japonais : le Katakana, l'Hiragana et le Kanji => idéogrammes)
- MITI est l'ancien acronyme du nom du ministère de l'économie japonais remplacé aujourd'hui par METI.

Des techniques spécifiques à l'informatique ont été développées à partir des années 1980 : les réseaux de neurones qui simulent l'architecture du cerveau humain, les algorithmes génétiques qui simulent le processus de sélection naturelle des individus, la programmation logique inductive qui fait « marcher à l'envers » le processus habituel de déduction, les réseaux bayésiens qui se fondent sur la théorie des probabilités pour choisir, parmi plusieurs hypothèses, la plus satisfaisante.

Fin des années 1980

L'IA s'est essentiellement focalisée sur les théories et techniques permettant la réalisation d'intelligences individuelles. Dans la nature, il existe toutefois une autre forme d'intelligence – collective celle-là, comme les êtres multicellulaires simples, les colonies d'insectes sociaux, les sociétés humaines. Ces sources d'inspiration montrent qu'une forme d'intelligence supérieure peut résulter de l'activité corrélée d'entités plus simples. Dans les systèmes artificiels, ce champ porte le nom d'« IA distribuée » ou de « systèmes multiagents ».

Années 1990-2000

L'avènement d'Internet a ouvert la voie au partage et à la communication de connaissances. Comment organiser et traiter ces gigantesques masses d'information ? Comment en extraire les connaissances pertinentes pour les problèmes posés ? Les moteurs de recherche comme *Google* ont intégré dans leurs tâches des techniques avancées de recherche d'information (*information retrieval*) et d'intelligence artificielle (*data mining*).

En 1994, une équipe française met au point dans le cadre de ses recherches en vie artificielle les « Jardins des hasards » Ce sont des jardins virtuels dont l'évolution est fonction de données numériques reçues par modem en temps réel. Ils sont composés de plusieurs familles de formes qui naissent, grandissent, meurent et interagissent entre elles suivant des comportements inspirés de la vie. Ils constituent des écosystèmes de vie artificielle.

Les couleurs des objets changent avec les données météo et avec le temps chronologique au fil des jours et des saisons.

En 1995, le système automatique de vision *ALVINN* de Carnegie Mellon University a permis la conduite automatique d'un véhicule appelé *Navlab5* de Pittsburgh à San Diego, pendant que les opérateurs humains s'occupaient du frein et de l'accélérateur.

En 1997, à Philadelphie, le champion du monde aux échecs, GarryKasparov, a été battu par *Deep Blue,* un ordinateur d'IBM, en six manches. Kasparov a gagné la première, a perdu la seconde et a très mal joué le reste. Furieux, il a dû s'incliner devant la machine.

Kasparov got wiped off the board, a déclaré le grand maître Ilya Gurevich.

L'apprentissage en ligne, ou *e-learning*, est en pleine expansion. L'IA a permis grâce à ses techniques de mettre en œuvre des systèmes d'éducation à distance de plus en plus performants.

Il y a une meilleure prise en compte du profil de l'apprenant (cognitif, affectif et inférentiel), une construction le plus souvent collaborative de la base de connaissances (curriculum), une interaction plus intelligente entre le système et l'apprenant, etc.

Domaines de recherche en IA

Réalité virtuelle

Ce domaine propose de nouvelles formes d'interaction entre l'homme et la

machine. L'arrivée d'ordinateurs plus puissants, dotés d'impressionnantes capacités graphiques en trois dimensions, couplés à des périphériques de visualisation et d'interaction (casque, gant, etc.), permet de fournir les informations sensorielles nécessaires pour convaincre des utilisateurs qu'ils sont en immersion.

Larry Hedges du Georgia Institute of Technology utilise depuis longtemps la réalité virtuelle pour guérir certaines phobies comme celles de l'ascenseur ou celles des araignées.

Reconnaissance des formes

Les recherches dans ce domaine visent à automatiser le discernement de situations typiques sur le plan de la perception. Ses méthodes trouvent des applications nombreuses. Citons la vision, la reconnaissance de la parole, la lecture optique de documents et la synthèse d'images.

Les progrès en reconnaissance d'images vidéo permettent déjà à la police de repérer une cible dans une foule.

Vie artificielle

Ce domaine s'intéresse à l'étude des écosystèmes et à la reproduction, par des systèmes artificiels, de caractéristiques propres aux systèmes vivants (depuis les mécanismes de fonctionnement cellulaire jusqu'aux dynamiques de peuplement, en passant par des modèles de développement individuel).

Applications de l'IA

Diagnostic médical, thérapie, surveillance d'appareils Synthèse d'images, vision par ordinateur Classifications naturelles (biologie, minéralogie, …) Planification de tâches (prédictions financières, …) Architecture (conception assistée par ordinateur)

Détection de pannes (Sherlock pour les avions F16)

Éducation (Systèmes Tutoriels Intelligents, e-Learning) Génie (vérification de règles de design)

Prospection géologique (gisements miniers)

Centrales nucléaires, feux de forêts (systèmes à temps réel) Simulateurs de vols (CAE, Bombardier, …)