Programmation par objets
IUP-MIAGe 3^{eme} année
Université Bordeaux I

Alain Griffault

22 octobre 2002

Chapitre 1

Introduction

1.1 Bibliographie

Ce document est fortement inspiré des documents suivants :
- Les Objets. M. Bouzeghoub, G. Gardarin et P. Valduriez (Eyrolles).
- Design Patterns : Catalogue de modèles de conception réutilisables. E. Gamma, R. Helm, R. Johnson et J. Vlissides (Vuibert).
- Des solutions objets : Gérer les projets orientés objets. G. Booch (International Thomson Publishing).
- Java : de l'esprit à la méthode. M. Bonjour, G. Falquet, J. Guyot et A. Le Grand (International Thomson Publishing).
- JDBC et JAVA : Guide du programmeur. G. Reese (O'Reilly).

1.2 Les origines

Un objet associe données et traitements dans une même entité en ne laissant visible que son interface.

L'approche objet est née en 1966 avec le langage SIMULA, dont l'objectif était la simulation de processus parallèle.

Le concept d'abstraction utilisé a été formalisée dans les années 70 avec la théorie des types abstraits. L'idée est de spécifier formellement des ensembles de fonctions sans se soucier de l'implémentation.

En 1976, le langage Smalltalk implémente la notion de type abstrait sous forme de classe, structure les classes en hiérarchie avec un mécanisme d'héritage, et utilise l'envoi et la réception de message entre les objets en lieu et place des appels de procédure. Smalltalk est l'inspiration essentielle de l'approche objet.

1.3 Limite de la programmation structurée

La programmation structurée (Algol, Pascal), a été conçue pour mieux organiser les programmes complexes en associant les traitements aux structures de données qu'ils manipulent.

Cette approche s'est révélée bien adaptée au développement d'applications complexes et peu évolutives. Elle atteint ses limites lorsque les structures de données et/ou les procédures doivent être partagées par différents programmes.

1.4 Pourquoi les objets ?

En conception, l'objet correspond à un concept du monde réel qui peut être spécialisé. Avec une programmation structurée, cette correspondance disparaît.

En programmation, du point de vue structurel, l'objet est une instance d'un type de données caractérisé par une structure cachée par des opérations. Une modification de la structure cachée est toujours possible.

En programmation, dans un contexte multi-processus et/ou de type IHM, l'objet est une entité autonome et active qui répond et/ou émet des messages. Le prototypage rapide et la simulation d'algorithmes distribués est facilité.

1.5 Perspectives

Depuis 1992, et des premières expériences réussies, l'approche objet s'est répandue notamment grâce aux applications client-serveur.

Les gains en productivité semblent bons.

Attention :

- Il n'y a toujours pas de théorie complète de l'orientée objet.
- Les définitions des différents concepts varient un peu d'un livre à un autre.
- Les objets ne sont pas la panacée qui résout tous les problèmes de conception et de développement.

Chapitre 2

Les concepts objets

2.1 Les objets et les relations inter-objets

Définition 2.1 *Un objet est une abstraction d'une donnée du monde réel caractérisée par :*
Objet = identité + état + comportement

Attention, l'identité et l'égalité de deux objets sont deux notions différentes.
$(o1 == o2) \Leftrightarrow (o1.identité = o2.identité)$
$(o1 = o2) \Leftrightarrow (o1.état = o2.état)$

Les liens entre les objets peuvent être représentés par des références aux identités des objets. Un lien mono-valué est représenté par un attribut dans l'objet source, dont la valeur est la référence de l'objet cible. Un lien multivalué est représenté par un attribut dans l'objet source, dont la valeur est une collection de références d'objets.

Les liens entre objets sont donc facilement représentables par un graphe orienté d'objets.

Un chemin dans un tel graphe $o.A1.A2...An$ est ainsi un objet ou une collection d'objets.

Exercice 2.1 *Soient les objets :*

```
<o1, employé, (Martin, o10)>
<o2, employé, (Dupont, o10)>
<o3, employé, (Martin, o11)>
<o4, employé, (Machin, o11)>
<o5, employé, (Dupond, o11)>
<o10, site, (Paris, o1, (o2, o3))>
<o11, site, (Grenoble, o5, (o3, o4))>
```

Dessinez le graphe des objets associé.

2.2 Les classes et les relations interclasses

2.2.1 Les classes

Définition 2.2 *Une classe factorise les caractéristiques communes des objets :*
Classe = instanciation + attributs + opérations

La classe fournit un mécanisme d'instanciation (l'opération new), un objet est ainsi une instance d'une classe.

Les attributs d'une classe sont aussi appelés les variables d'instances.

Différents niveaux de visibilité pour les opérations : public et privé.

L'interface de la classe est constitué des opérations publiques. C'est le principe de l'encapsulation des attributs.

Remarque : les langages objets diffèrent quelque peu sur cet aspect.

La spécification d'une opération est sa signature : type des arguments d'entrée, type de l'objet receveur et type de l'argument de sortie.

Les objets sont accédés soit par un appel de procédure (langage compilé), soit par envoi de messages (langage interprété).

Exercice 2.2 *Donnez les classes* `Employé` *et* `Site` *de l'exercice précédent.*

2.2.2 Le graphe de généralisation

La relation de généralisation qui traduit la relation naturelle *est-un (cas particulier de)* est à la base de la hiérarchie des classes et de la relation d'héritage.

Définition 2.3 *Généralisation : fonction qui, à une classe origine, dite sous-classe, fait correspondre une classe plus générale, dite super-classe.*

La fonction inverse est appelée fonction de spécialisation.

Exercice 2.3 *Ajouter les classes nécessaires pour avoir quatre catégories d'employés : des administratifs, des ingénieurs, des chercheurs et des stagiaires qui sont à la fois des chercheurs et des étudiants. Dessinez le graphe associé.*

2.2.3 Le graphe d'instanciation

Définition 2.4 *Instanciation : relation entre un objet et sa classe d'appartenance qui a permis de la créer.*

Une des limites de la distinction entre objet et classe est l'impossibilité de créer des classes pendant l'exécution d'un programme.

Définition 2.5 *Méta-classe : classe générateur de classes (ses instances), regroupant en particulier les opérations applicables aux classes et les structures de données communes à toutes les classes.*

Remarque : la méta-classe est sa propre instance, et la classe object est la racine de la hiérarchie

Exercice 2.4 *Ajoutez au graphe précédant les classes* `méta-classe` *et* `object`

2.2.4 Le graphe d'association et d'agrégation

L'association est un concept issu du modèle entité-association, défini par Chen en 76 pour la modélisation sémantique des données.

Définition 2.6 *Association : relation entre plusieurs classes, caractérisée par un verbe, décrivant conceptuellement les liens entre les objets de ces classes.*

Une caractéristique importante d'une association est sa cardinalité : 1-1, 1-N ou N-M.
Parmi les associations, deux types particulier :

Binaire : le verbe peut être utilisé dans les deux sens, le lien est donc bidirectionnel.

Partie-de : utile pour modéliser les objets complexes qui sont composés d'autres objets.

Définition 2.7 *Agrégation : relation entre deux classes spécifiant que des objets d'une classe sont les composants de l'autre classe.*

Remarque : L'agrégation est transitive

Exercice 2.5 *Dessinez un graphe d'association et d'agrégation entre* `Employé, Site` *et quatre nouvelles classes* `Bâtiment, Bureau, Salle-Réunion` *et* `Parking`.

2.2.5 Les collections

Lorsqu'un attribut est multivalué, sa valeur est une collection. Le langage peut fournir un constructeur de type pour les collections (une liste), mais il est préférable de fournir une classe.

Définition 2.8 *Classe collection : classe générique (une classe paramétrée) permettant d'imposer une structure à une collection d'objets et de les manipuler par des opérations associées.*

Exercice 2.6 *Ajouter dans le graphe des classes, les classes* `Collection, Ensemble, Tableau`, *et* `Liste`.

2.3 L'héritage

L'héritage capture la relation de généralisation, il facilite ainsi la modélisation des objets et la modularité et la réutilisation du code.

Définition 2.9 *Héritage simple : mécanisme de transmission des propriétés d'une classe vers une sous-classe.*

Les objets suivants sont corrects

```
<o1, employé, (Martin, o10)>
<o2, administratif, (Dupont, o10)>
<o3, chercheur, (Martin, o11)>
<o4, employé, (Machin, o11)>
<o5, ingenieur, (Dupond, o11)>
```

Définition 2.10 *Héritage multiple : mécanisme par lequel une sous-classe hérite des propriétés de plus d'une super-classe.*

L'héritage permet d'éviter la redondance des définitions des attributs et des opérations.
Remarques :
- *Attention aux conflits dans le cas de l'héritage multiple.*
- *Il peut être pratique de ne pas définir d'opération.*
- *Il peut être pratique de redéfinir une opération.*

Exercice 2.7 *Quel est le conflit éventuel avec l'objet :*

`<o6, stagiaire, (X, o10), (maitrise, Bordeaux)>`

Définition 2.11 *Classe abstraite = attributs + signature des opérations.*

Une classe abstraite ne peut pas être instanciée. Elle doit être la super-classe d'une autre, dans laquelle les opérations devront être implémentées.

Définition 2.12 *Surcharge : redéfinition d'une opération avec un code différent.*

Définition 2.13 *Polymorphisme : faculté d'une opération à pouvoir s'appliquer à des objets de classes différentes.*

Remarques : le polymorphisme complique le déroulement d'un programme. Pour une opération polymorphe, quelle est l'implémentation à exécuter ? (c-a-d quelle est la classe réelle de l'objet ?). Ce choix peut être décidé à la compilation où à l'exécution.

Définition 2.14 *Liaison dynamique : mécanisme de sélection de code d'une opération à l'exécution en fonction de la classe de l'objet receveur.*

Remarque : la liaison dynamique est difficilement compatible avec un typage statique fort, pourtant garant de sûreté à l'exécution.

Exercice 2.8 *Les primes des employés dépendent de leur statut : les administratifs sont intéressés au recouvrement des factures, les ingénieurs au chiffre d'affaires des projets gérés, les chercheurs au nombre de brevets déposés, et les stagiaires à la bonne volonté de leur responsable. Proposez une solution.*

Chapitre 3

Les langages objets

3.1 Introduction

Il faut être conscient que la plupart des concepts de la programmation objet sont connus depuis 1970.

La période 1960-1980 est sans doute celle pendant laquelle les informaticiens ont "inventé" le plus de langage de programmation. La période actuelle étant plus dévolue aux outils connexes.

Dans ce chapitre, nous allons comparer plusieurs langages objets, non pas en fonction de leurs performances respectives, (performance de l'exécutable généré, taux d'utilisation dans le monde industriel), mais en fonction des choix effectués vis-à-vis des concepts objets présentés au chapitre précédent, notamment vis-à-vis des relations interclasse.

3.2 Les langages comparés

ADA Créé en 1977 pour uniformiser les développements de l'armée américaine, ADA intègre la généricité et les types abstraits. La version 1995 en fait un langage objet.

SIMULA Créé en 1966 pour simuler des processus parallèles.

C++ Créé en 1986 comme une extension de C avec les concepts de SIMULA.

Object Pascal Créé au début des années 80, comme une extension de Pascal avec les concepts de SIMULA.

SMALLTALK Créé en 1972, basé sur les seuls concepts d'objet, de classe, d'héritage simple et d'envoi de messages. SMALLTALK a permis l'inter-activité des objets telle qu'elle fut intégrée dans les Macintosh.

EIFFEL Créé en 1992 dans un contexte de génie logiciel, afin de faciliter le développement et la mise au point. Il peut être compilé, mais aussi générer du code C.

Objective-C Créé en 1986 comme une extension de C avec les concepts empruntés à SMALLTALK.

JAVA Créé en 1995 pour le développement de composants pour des applications en réseau.

3.3 Les critères de comparaison

La modélisation Comment est gérer l'identité de l'objet, et existe t-il des attributs et/ou des méthodes de la classe ?

L'encapsulation Comment est défini l'interface d'un objet ?

La modularité Comment est gérer le partage entre les applications ?

La hiérarchie des classes héritage simple/multiple, généricité, classe abstraite, méta-classe ...

Le typage est-il statique ou dynamique ?

Le parallélisme est-il permis, et si oui, comment est-il gérer ?

La gestion mémoire est-elle manuelle ou automatique ?

TAB. 3.1 – Une comparaison des langages objets

	Ada	Simula	C++	O. Pascal	Smalltalk	Eiffel	O.-C	Java
Modélisation								
Variables de classes		oui	non	non	oui	non	non	oui
Méthodes de classes		oui	non	non	oui	non	oui	oui
Pointeurs visibles		oui	oui	oui	non		oui	non
Encapsulation								
Des attributs		lect, ecr	pub, pro, pri	pub, pri	pri	pri	pri	pub, pro, pri
Des opérations		pub, pri	pub, pro, pri	pub, pri	pub	pub	pub	pub, pro, pri
Modularité								
Type de modules	package	module	fichier	unit	fichier		fichier	package
Hiérarchie								
Héritage		simple	multiple	simple	simple	multiple	simple	simple
Généricité	oui	oui	oui	non	non	oui	non	non
Classe abstraite	oui	oui	oui		non		non	oui
Classe object		oui	non		oui		oui	oui
Métaclasse		non	non	non	oui	non	non	oui
Typage								
Typage dynamique		non	non	non	oui		oui	non
Liaison dynamique		virtuel	virtuel	virtuel	oui		oui	oui
Parallélisme								
Multitâche	oui	oui	non	non	oui		non	oui
Synchronisation	RdV	Moniteur	non	non	message		non	Moniteur
Gestion mémoire								
Chargeur dynamique		non	non	oui	non			oui
Ramasse-miettes		oui	non	oui	oui	oui	non	oui

3.4 Lequel choisir ?

Bien évidemment, le tableau ?? ne répond pas à la question, tout juste permet-il de comprendre certains choix.

Chapitre 4

Architecture d'une application Java

4.1 La modularité : les packages

Un package Java est un groupe de classes, et une classe ne peut appartenir qu'à un seul package. Ils peuvent être organisés hiérarchiquement, l'organisation est en fait l'organisation du système de fichiers.

```
package [<nomDePackage>.]*<nomDuPackage>
```

Si cette déclaration est absente, cela est équivalent à :

```
package anonymous
```

Le regroupement des classes en package est utile dès que les applications deviennent importantes en nombre de classes. Les critères de regroupement peuvent être fonctionnel, de domaine où d'implantation.

4.1.1 La visibilité des classes

Une classe peut être déclarée publique ou pas. Si elle est publique, elle est utilisable par des classes d'autres packages, sinon, seules les classes du même package peuvent l'utiliser.

Soit le début de source

```
1  package MonPackage;
2
3  class MaClasse {
```

et cet autre

```
1  package MonPackage;
2
3  public class MaClassePublique {
```

Le source suivant est alors correct.

```
1  package MonPackage;
2
3  class MaClasseUtilisatrice {
4      public static void main(String[] args){
5          MaClassePublique mcp = new MaClassePublique();
6          MaClasse mc = new MaClasse();
```

alors que celui-ci :

```
1  import MonPackage.*;
2
3  class ClasseUtilisatriceA {
4      public static void main(String[] args){
5          MaClassePublique mcp = new MaClassePublique();
6          MaClasse mc = new MaClasse();
```

provoque les erreurs suivantes à la compilation.

```
ClasseUtilisatriceA.java:6: MonPackage.MaClasse is not public in MonPackage; cannot be accessed from ou
MaClasse mc = new MaClasse();
       ^
ClasseUtilisatriceA.java:6: MonPackage.MaClasse is not public in MonPackage; cannot be accessed from ou
MaClasse mc = new MaClasse();
                         ^
ClasseUtilisatriceA.java:6: MaClasse() in MonPackage.MaClasse is not defined in a public class or inter
MaClasse mc = new MaClasse();
                     ^
3 errors
```

Les packages servent à regrouper des classes, afin qu'elles forment un tout cohérent, mais en restreignant l'accès aux utilisateurs par des interfaces clairement définies.

4.2 La modélisation

4.2.1 Les attributs d'un objet

Le mot clef `static` détermine si un attribut est une variable d'instance (appartient à l'objet), où une variable de classe (appartient à la classe). Par défaut, la variable est d'instance.

Dans la classe `MaClassePublique`

```
4       static int nbInstance = 0;
5       int ident;
```

Les instructions suivantes de `MaClasseUtilisatrice` sont alors correctes.

```
7           System.out.println("Nombres d'instances :"+MaClassePublique.nbInstance);
8           System.out.println("Mon instance :"+mcp.ident);
```

4.2.2 Les opérations d'un objet

Le mot clef `static` détermine si une opération (une méthode) est une méthode d'instance (appartient à l'objet), où une méthode de classe (appartient à la classe). Par défaut, la méthode est d'instance.

Dans la classe `MaClassePublique`

```
11      static public int getNbInstance(){
12          return nbInstance;
13      }
14
15      public int getIdent(){
16          return ident;
17      }
```

Les instructions suivantes de `MaClasseUtilisatrice` sont alors correctes.

```
9           System.out.println("Nombres d'instances :"+MaClassePublique.getNbInstance());
10          System.out.println("Mon instance :"+mcp.getIdent());
```

4.2.3 La référence aux objets

En Java, les pointeurs n'existent pas. En fait, à l'exception des types primitifs (boolean, char, byte, short, int, long, float, double), tout est implanté avec des pointeurs. Il faut distinguer la création d'un attribut, et la création de l'objet référencé par l'attribut.

```
1   import MonPackage.*;
2
3   class ClasseUtilisatriceB {
```

```
4       public static void main(String[] args){
5           MaClassePublique mcp;                   // création d'une variable
6           System.out.println("Mon instance :"+mcp.getIdent());
7           mcp = new MaClassePublique();           // création de l'objet
8           System.out.println("Mon instance :"+mcp.getIdent());
```

provoque l'erreur suivante à la compilation.

```
ClasseUtilisatriceB.java:6: variable mcp might not have been initialized
System.out.println("Mon instance :"+mcp.getIdent());
                                                 ^
1 error
```

4.3 L'encapsulation des objets

4.3.1 L'encapsulation des attributs et des méthodes

Trois mots clefs (`public, protected, private`) détermine les règles de visibilité sur les attributs et les méthodes d'un objet.

- visibilité restreinte à la classe : `private`
- visibilité restreinte au package : aucun mot clef
- visibilité non restreinte : `public`
- visibilité étendue au sous-classes du même package :`private protected`
- visibilité étendue à toutes les sous-classes : `protected`

4.4 L'héritage

A la distinction déjà vue entre héritage simple et héritage multiple, on ajoute également celle-ci :

Définition 4.1 *Héritage d'implantation : une sous-classe hérite de comportement de sa super-classe, où d'une partie de ses attributs.*

Définition 4.2 *Héritage d'interface : une sous-classe n'hérite d'aucun comportement de sa super-classe, et d'aucun de ses attributs.*

En Java, il existe un héritage simple d'implantation et un héritage multiple d'interface. Ainsi le conflit d'opérations sur un objet n'existe pas.

```
class SC{}
Interface I1{}
Interface I2{}
Interface I3{}
class C extends SC implements I1, I2, I3{}
```

Attention à l'héritage des constructeurs, notamment avec les constructeurs sans arguments et l'utilisation de super().

En Java, les classes abstraites sont possibles, les classes `Object`, `Class` et `ClassLoader` modifient la hiérarchie des classes. Ce qui suit est extrait de la documentation.

```
java.lang
Class Object

java.lang.Object

public class Object

Class Object is the root of the class hierarchy.
Every class has Object as a superclass.
All objects, including arrays, implement the methods of this class.
```

```
java.lang
Class Class

java.lang.Object
  |
  +--java.lang.Class

public final class Class
extends Object
implements Serializable

Instances of the class Class represent classes and interfaces in a running Java application.
Every array also belongs to a class that is reflected as a Class object
that is shared by all arrays with the same element type and number of dimensions.
The primitive Java types (boolean, byte, char, short, int, long, float, and double),
and the keyword void are also represented as Class objects.

Class has no public constructor.
Instead Class objects are constructed automatically by the Java Virtual Machine
as classes are loaded and by calls to the defineClass method in the class loader.
```

```
java.lang
Class ClassLoader

java.lang.Object
  |
  +--java.lang.ClassLoader

Direct Known Subclasses:
      SecureClassLoader

public abstract class ClassLoader
extends Object

The class ClassLoader is an abstract class.
A class loader is an object that is responsible for loading classes.
Given the name of a class, it should attempt to locate or generate data
that constitutes a definition for the class.
A typical strategy is to transform the name into a file name
and then read a "class file" of that name from a file system.

Every Class object contains a reference to the ClassLoader that defined it.
```

4.5 Liaison dynamique et polymorphisme

Définition 4.3 *surcharge : dans une classe, un même nom est utilisé pour différents attributs où différentes méthodes (les signatures doivent différées).*

Définition 4.4 *redéfinition : dans une sous-classe, l'implantation d'une méthode est changée (spécialisée).*

```
class C {
  int x;

  int ajouter(int a){return x+a;}
  void ajouter(int a){x = x+a;} //surcharge
```

```
  void ajouter(int a; int b){x = x+a+b;}        //surcharge
}
class SC extends C {
  float x;                                              //surcharge
  void ajouter(int a; int b){ajouter(a+2*b);}        //redéfinition
  void ajouter(int a; int b; int c){ajouter(a+b,c);} //surcharge
}
```

Pour les attributs : la surcharge est interdite dans une classe et permise dans les sous-classes, la redéfinition n'a pas de sens.

Pour les méthodes : la surcharge est permise dans la classe et les sous-classes, la redéfinition est interdite dans la classe et permise dans les sous-classes.

4.6 Les exceptions

Java offre un mécanisme d'exceptions pour traiter les situations exceptionnelles. Une exception est un objet de la classe `Exception` ou d'une de ses sous-classes, qui est créé au moment où une situation anormale est détectée. Une fois créée, l'exception est levée à l'aide de l'instruction `throw`, propager jusqu'au premier bloc d'instructions protégé par un `catch`. Si aucun bloc ne peut capturer l'exception, celle-ci est propagée jusqu'à la JVM qui interrompt le programme.

4.6.1 Un premier exemple

```
1   import java.util.*;
2
3   public class SansException {
4
5       private static final int max = 5;
6       private static Random r = new Random();
7
8       public void division (int a, int b) {
9           System.out.println("a :"+a+" b :"+b+" a/b :"+a/b);
10      }
11
12      public static void main(String args[]){
13          SansException se = new SansException();
14          System.out.println("Debut");
15          for(int i = 0; i < 10; i++)
16              se.division(r.nextInt(max),r.nextInt(max));
17          System.out.println("Fin");
18      }
19
20  }
```

Ce programme peut produire la sortie suivante à l'écran :

```
Debut
a :1 b :3 a/b :0
a :1 b :2 a/b :0
a :1 b :4 a/b :0
a :3 b :4 a/b :0
a :0 b :4 a/b :0
a :3 b :3 a/b :1
a :3 b :1 a/b :3
java.lang.ArithmeticException
```

```
at SansException.division(SansException.java:9)
at SansException.main(SansException.java:5)
```

La première division par zéro arrête l'exécution. Dans la nouvelle version, le programme termine normalement.

```
1  import java.util.*;
2
3  public class AvecException {
4
5      private static final int max = 5;
6      private static Random r = new Random();
7
8      public void division (int a, int b) {
9          try {
10             System.out.println("a :"+a+" b :"+b+" a/b :"+a/b);
11         }
12         catch (java.lang.ArithmeticException e){
13             System.out.println("a :"+a+" b :"+b+" a/b : impossible");
14         }
15     }
16
17     public static void main(String args[]){
18         AvecException ae = new AvecException();
19         System.out.println("Debut");
20         for(int i = 0; i < 10; i++)
21             ae.division(r.nextInt(max),r.nextInt(max));
22         System.out.println("Fin");
23     }
24
25 }
```

Ce programme peut produire la sortie suivante à l'écran :

```
Debut
a :2 b :2 a/b :1
a :2 b :4 a/b :0
a :0 b :4 a/b :0
a :4 b :0 a/b : impossible
a :3 b :3 a/b :1
a :2 b :0 a/b : impossible
a :0 b :3 a/b :0
a :4 b :1 a/b :4
a :0 b :4 a/b :0
a :3 b :2 a/b :1
Fin
```

4.6.2 Gérer ses propres exceptions

Il peut être utile d'avoir ses propres classes pour des exceptions.

```
1  public class Vacances extends Exception {
2      public Vacances(){super("En vacances");}
3      public Vacances(String st){super(st);}
4  }
```

Une utilisation possible :

```
1  import java.util.*;
2
3  public class Agenda {
4
5      Calendar calendrier;
6
7      public Agenda(){
8          calendrier = new GregorianCalendar();
9      }
10
11     public void planning (int nbJours) {
12         for (int i=0; i < nbJours; i++){
13             try {
14                 afficher();
15             }
16             catch(Vacances v){
17                 System.out.println(v.getMessage());
18             }
19             calendrier.roll(Calendar.DATE, true);
20         }
21     }
22
23     public void afficher () throws Vacances {
24         if (calendrier.get(Calendar.DAY_OF_WEEK)==Calendar.SUNDAY)
25             throw new Vacances("Jamais le dimanche");
26         if (calendrier.get(Calendar.DAY_OF_MONTH)==1)
27            throw new Vacances("Jamais le premier jour du mois");
28         System.out.println(calendrier.getTime());
29         for (int h=8; h < 17; h++){
30            System.out.println(h + " heures :");
31         }
32     }
33
34     public static void main(String args[]){
35         Agenda ag = new Agenda();
36         System.out.println("Debut");
37         ag.planning(Integer.parseInt(args[0]));
38         System.out.println("Fin");
39     }
40
41 }
```

4.7 Le parallélisme

Ce paradigme fait l'objet d'un cours au second semestre. Les threads seront abordés également plus tard dans ce cours.

4.8 La gestion mémoire

Java est muni d'un chargeur dynamique et d'un ramasse-miettes.

Chapitre 5

Les méthodes de développement logiciel

5.1 Les phases classiques du développement

Traditionnellement, le développement logiciel distingue les étapes suivantes :

La définition des besoins qui aboutit à un cahier des charges. C'est le document contractuel entre le client et le réalisateur.

La conception qui aboutit à différents documents décrivant les structures de données et les solutions techniques retenues.

L'implémentation qui aboutit au code source commenté et au code exécutable.

Les tests qui aboutissent à la recette par le client. Par les tests, le réalisateur montre au client que le contrat est respecté.

Les manuels d'utilisation et de maintenance qui permettent l'un la prise en main par le client du produit, l'autre un gain de temps d'une part sur les éventuelles corrections à faire, d'autre part sur les ajouts ultérieurs de fonctionnalités.

5.2 Les cycles classiques du développement

Le modèle en cascade date des années 1970, il enchaîne sans retour arrière les quatre étapes précédentes. Il est bien adapté pour le développement de type (entrées, transformations, sorties), mais pas du tout au développement de systèmes complexes.

Le modèle en V est une variante du précédent (la conception est modulaire) qui permet de distinguer la vérification des composants et la validation du système, mais celle-ci apparaît bien trop tard.

Le modèle en spirale essaie de valider au plus tôt un système. Il est basé sur une suite de prototypes dont le dernier est le système. La notion de prototype est malheureusement floue, et bien que décrit comme le cycle le plus adapté aux objets, aucune méthodes objets n'y fait explicitement référence.

Le modèle tridimensionnel possède trois axes : le premier est le modèle de cascade, le second une distinction de niveaux (physique, logique, conceptuel) et un troisième est un axe temporel relatif à la place du système dans le système d'information global de l'entreprise, malheureusement sans trop décrite cette sémantique. Il est très orienté bases de données.

Tous ces modèles souffrent d'un manque : elles ne parlent pas des techniques de modélisation et de réalisation, qui sont certainement les principaux facteurs de la qualité d'un système, et de la productivité des développeurs.

5.3 Les méthodes de conception

Les approches cartésiennes considèrent un système comme une fonction que l'on peut hiérarchiquement décomposer en fonctions. Elles sont performantes, mais atteignent leurs limites pour les problèmes de cohérences des données manipulées par plusieurs fonctions, et pour les problèmes d'interactions non hiérarchique dans le cas des systèmes complexes.

Les approches systémiques distinguent nettement la modélisation des données et celle des traitements. Cela permet de traiter correctement des données (niveaux externe, conceptuel et interne), de traiter des systèmes complexes, mais malheureusement l'absence de concept commun entre les modèles de données et de traitements rendent leurs utilisations difficiles hors du contexte BD.

Les approches objets essaient d'ajouter de la cohérence entre le modèle des données et le modèles des traitements.

5.4 L'approche objets

5.4.1 Les méthodes objets

Les méthodes objets sont basées sur une triple perception du système : une dimension statique qui décrit les objets du système, une dimension dynamique qui contrôle les changements d'états des objets et une dimension fonctionnelle qui décrit les processus de transformation.

Les méthodes objets sont plus ou moins complètes vis-à-vis de ces trois dimensions.

5.4.2 Les modèles et techniques objets

Chaque méthode utilisent ses propres modèles formels, la plupart utilise une hiérarchie de modèles (conceptuel, logique et physique), mais c'est au niveau conceptuel que les modèles utilisés caractérisent le mieux la méthode.

Deux tendances, l'accent est mis soit sur les interactions entre objets, soit sur les associations statiques et la structuration interne des données.

Les modèles de données utilisent soit les modèles entité-association et Agrégation-Généralisation avec une approche sémantique, soit le modèle relationnel avec une approche relationnelle. Le premier est mieux adapté aux objets complexes.

Les modèles dynamiques ont pour objectif de décrire les règles d'évolution des objets au cours du temps. Différents modèles formels sont utilisés :

- **Les diagrammes états/transitions** sont des graphes dans lequel les sommets sont des valeurs d'états, et les arcs des événements déclencheurs d'un changement d'états. Ils vont des simples graphes orientés, aux graphes orientés avec des expressions de conditions sur les arcs.
- **Les règles actives** décrivent les réactions du système (comme les triggers dans les SGBD), dans un formalisme qui généralise le "switch". Un formalisme textuel pour décrire des diagrammes états/transitions.
- **Les statecharts** sont des diagrammes états/transitions hiérarchisés. Un état est lui-même un diagramme états/transitions.
- **Les réseaux de Petri** permettent comme les diagrammes états/transitions de représenter l'évolution d'un objet, mais également de représenter dans un même modèle, les états successifs de plusieurs objets différents. Les réseaux de Petri sont des graphes bipartites. Les nœuds sont soit des places pouvant contenir des jetons, soit des transitions. Une transition est tirable si toutes les places amonts contiennent au moins un jeton ; après la transition le contenu des places amonts est décrémenté, celui des places aval est incrémenté.

Les modèles fonctionnels utilisent fréquemment les diagrammes de flux de données. Les DFD sont des graphes tripartites dont les nœuds sont soit des acteurs, soit des processus de transformation, soit des récipients de données ; et dont les arcs sont des flux de données. L'obtention des DFD se fait généralement par affinages successifs (méthode descendante).

5.4.3 Limites des approches objets

Aucune des méthodes ne couvre entièrement le cycle de développement, chacune ayant ses qualités et ses défauts. Plus grave, elles ne donnent pas forcement le même sens aux différents concepts objets, ce qui rend le passage d'une méthode à une autre difficile.

C'est dans ce contexte qu'est apparue la volonté de standardiser les terminologies et les notations : UML (Unified Modeling Language) est le fruit de ce travail.

La notation UML est l'objet d'un cours spécifique du second semestre.

Exercice 5.1 *Imaginez des modèles dynamiques pour décrire les évolutions possibles entre les situations suivantes : célibataire, marié, veuf, divorcé.*

Chapitre 6

Les catalogues de conception

6.1 Les interfaces Homme-machine

Le développement et la généralisation des écrans graphiques et des environnements de développement d'interfaces conviviales permet de répondre aux besoins des utilisateurs-développeurs qui souhaitent développer des applications complexes avec des outils simples.

Mais là encore, ce concept (interagir avec un système à l'aide de questions-réponses, de menus à choix multiples et d'icônes graphiques représentant des métaphores des objets quotidiens) date du début des années 70 avec Smalltalk.

Les métaphores ont été très développés dans le domaine de la bureautique, principalement par Apple. Depuis, de nombreux standards existent.

6.1.1 Réalisation d'interfaces dans Smalltalk

Les principes de base

Smalltalk offre une hiérarchie de classes prédéfinies et un modèle de conception d'interfaces (MVC pour Model-View-Controller).

Le modèle est l'objet sur lequel on veut travailler ; il représente l'application de l'utilisateur.

La vue est l'interface externe servant à présenter un objet à l'utilisateur ; elle peut être constituée de menus, d'icônes ou de boîtes de dialogue par exemple.

Le contrôleur est une interface interne, réalisant l'association dynamique entre l'objet et sa représentation ; il contrôle la sélection dans les menus où l'activité de la souris.

FIG. 6.1 – Le modèle MVC de Smalltalk

FIG. 6.2 – Les liens dans le modèle MVC de Smalltalk

L'utilisateur agit sur la vue, le contrôleur récupère l'événement et réalise la connexion avec l'objet concerné dans l'application. Si l'application veut afficher un résultat, elle peut soit passer par le contrôleur, soit transmettre directement le message à la vue.

A un modèle peuvent être associées plusieurs vues différentes, contrôlées par un contrôleur unique, ou bien chacune par un contrôleur.

La construction d'une vue

Une vue est une fenêtre dans laquelle les objets s'affichent. En Smalltalk, il y a trois types de fenêtres :
- les fenêtres normales avec décoration (label, ascenseur,...) ;
- les fenêtres de dialogue ;
- les fenêtres de type menu (Pop-up windows).

Les liens entre Vue, Contrôleur et Modèle

Du point de vue de Smalltalk, Vue, Contrôleur et Modèle sont des objets. Les méthodes associées à une vue sont des méthodes d'accès au contrôleur et des méthodes d'affichage. Les méthodes associées à un contrôleur sont des méthodes d'accès à la vue, des méthodes d'accès au modèle et des méthodes d'accès au détecteur d'événement. Les méthodes associées à un modèle sont des méthodes d'accès aux objets dépendants.

6.2 Les modèles de conception

6.2.1 Qu'est ce qu'un modèle de conception

Le nom de modèle décrit en un ou deux mots un problème de conception.

Le problème décrit les situations où le modèle s'applique.

La solution décrit les éléments qui constituent la conception, les relations entre eux, leurs parts dans la solution, leur coopération.

Les conséquences sont les effets résultants de la mise en œuvre du modèle et les variantes de compromis que celle-ci entraîne.

6.2.2 Catalogue et classification des modèles de conception

		Rôle		
		Créateur	Structurel	Comportement
Domaine	Classe	Fabrication	Adaptateur	Interprète Patron de méthode
	Objet	Fabrique abstraite Monteur Prototype Singleton	Adaptateur Pont Composite Décorateur Façade Poids mouche Procuration	Chaîne de responsabilité Commande Itérateur Médiateur Mémento Observateur État Stratégie Visiteur

6.2.3 Description des modèles de conception

Noms de modèle et classification ...

Intention ...

Alias ...

Motivation ...

Indications d'utilisation ...

Structure ...

Constituants ...

Collaborations ...

Conséquences ...

Implémentation ...

Exemples de code ...

Utilisations remarquables ...

Modèles apparentés ...

6.2.4 Les modèles créateurs

Ils permettent de rendre un système indépendant de la façon dont ses objets ont été créés, combinés et concrétisés.

Ils encapsulent la connaissance des classes concrètes que le système utilise, de sorte que tout ce que le système connaît de ces objets, est leurs interfaces, telles que définies par des classes abstraites.

Des briques de base

Les sources suivantes décrivent des briques de base permettant la construction d'un labyrinthe.

La classe Dédale

```
1  package brique;
2  import java.util.*;
3
4  public class Dedale {
5      private Vector plan;
```

La classe Direction

```
1  package brique;
2
3  public class Direction {
4      public static final int est = 0;
5      public static final int nord = 1;
6      public static final int ouest = 2;
7      public static final int sud = 3;
```

L'interface Lieu

```
1  package brique;
2
3  public interface Lieu {
4
5      public boolean entrer();
```

L'interface Mur et quelques classes l'implémentant

```
1  package brique;
2
3  public interface Mur {
```

```
1  package brique;
2
3  public class MurNormal implements Mur, Lieu {
```

```
1  package brique;
2
3  public class MurBombarde extends MurNormal {
```

L'interface Porte et quelques classes l'implémentant

```
1  package brique;
2
3  public interface Porte {
```

```
1  package brique;
2
3  public class PorteNormale implements Porte, Lieu {
```

L'interface Salle et quelques classes l'implémentant

```
1  package brique;
2
3  public interface Salle {
```

```
1  package brique;
2
3  public class SalleNormale implements Salle, Lieu {
```

```
1  package brique;
2  import java.util.*;
3
4  public class SallePiege extends SalleNormale {
```

Exercice 6.1 *Dessinez le graphe des classes.*

Une première version de labyrinthe

Le source suivant construit *manuellement* un labyrinthe.

La classe JeuDuLabyrinthe

```
1  import brique.*;
2  import java.util.*;
3
4  public class JeuDuLabyrinthe {
5      private Random r = new Random();
6
7      public Dedale CreeDedale() {
8          Dedale unDedale = new Dedale();
9          Salle s;
10         Mur me,mo,ms,mn;
11         for (int i=1; i<4; i++) {
12             s = new SalleNormale(i);
13             unDedale.ajouteSalle(s);
14             me = new MurNormal(Direction.est,s);
15             mo = new MurNormal(Direction.ouest,s);
16             ms = new MurBombarde(Direction.sud,s);
17             mn = new MurNormal(Direction.nord,s);
18         }
19         Porte laPorte = new PorteNormale(Direction.est,
20                                          unDedale.salleNumero(1),
21                                          unDedale.salleNumero(2),
22                                          r.nextBoolean());
23         unDedale.acolleSalles(Direction.nord,
24                               unDedale.salleNumero(1),
25                               unDedale.salleNumero(3));
26         System.out.println("dedale cree");
27         return unDedale;
28     }
29
30     public static void main(String args[]){
31         JeuDuLabyrinthe jeu = new JeuDuLabyrinthe();
32         jeu.CreeDedale();
33     }
34
35 }
```

Une version utilisant une fabrique abstraite

Intention La fabrique abstraite fournit une interface, pour créer des familles d'objets apparentés ou dépendants, sans avoir à spécifier leurs classes concrètes.

Exemples de code Les sources suivantes utilisent le modèle d'une fabrique abstraite pour la création d'un labyrinthe.

La classe FabriqueDedale

```
1  package fabriqueAbstraite;
2
3  import brique.*;
4
5  public interface FabriqueDedale {
6
7      public Dedale creeDedale();
8      public Mur creeMur();
9      public Mur creeMur(int d, Salle s);
10     public Salle creeSalle(int n);
11     public Porte creePorte();
12     public Porte creePorte(int d1, Salle s1, Salle s2, boolean ouverte);
13     public String getName();
14
15 }
```

La classe FabriqueDedaleNormal

```
1  package fabriqueAbstraite;
2
3  import brique.*;
4
5  public class FabriqueDedaleNormal implements FabriqueDedale {
```

La classe FabriqueDedalePiege

```
1  package fabriqueAbstraite;
2
3  import brique.*;
4
5  public class FabriqueDedalePiege implements FabriqueDedale {
```

La classe JeuDuLabyrinthe

```
1  import fabriqueAbstraite.*;
2  import brique.*;
3  import java.util.*;
4
5  public class JeuDuLabyrintheFA {
6      private Random r = new Random();
7
8      public Dedale CreeDedale(FabriqueDedale fabrique) {
9          Dedale unDedale = fabrique.creeDedale();
10         Salle s;
11         Mur me,mo,ms,mn;
12         for (int i=1; i<4; i++) {
13             s = fabrique.creeSalle(i);
14             unDedale.ajouteSalle(s);
15             me = fabrique.creeMur(Direction.est,s);
16             mo = fabrique.creeMur(Direction.ouest,s);
17             ms = fabrique.creeMur(Direction.sud,s);
```

```
18              mn = fabrique.creeMur(Direction.nord,s);
19          }
20          Porte laPorte = fabrique.creePorte(Direction.est,
21                                             unDedale.salleNumero(1),
22                                             unDedale.salleNumero(2),
23                                             r.nextBoolean());
24          unDedale.acolleSalles(Direction.nord,
25                                unDedale.salleNumero(1),
26                                unDedale.salleNumero(3));
27          System.out.println(fabrique.getName());
28          return unDedale;
29      }
30
31      public static void main(String args[]){
32          JeuDuLabyrintheFA jeu = new JeuDuLabyrintheFA();
33          Dedale dedaleN = jeu.CreeDedale(new FabriqueDedaleNormal());
34          Dedale dedaleP = jeu.CreeDedale(new FabriqueDedalePiege());
35      }
36
37  }
```

Exercice 6.2 *Dessinez le graphe des classes.*

Une version utilisant un monteur

Intention Le monteur dissocie, dans un objet complexe, sa construction de sa représentation, de sorte que, le même procédé de construction puisse engendrer des représentations différentes.

Exemples de code Les sources suivantes utilisent le modèle d'un monteur pour la création d'un labyrinthe.

La classe MonteurDedale

```
1   package monteur;
2
3   import brique.*;
4
5   public interface MonteurDedale {
6
7       public void creeDedale();
8       public void creeMur(int d, Salle s);
9       public void creeSalle(int n);
10      public void creePorte(int d1, Salle s1, Salle s2, boolean ouverte);
11      public void acolleSalles(int d1, Salle s1, Salle s2);
12      public Dedale acqDedale();
```

```
13      public String getName();
14  }
```

La classe MonteurDedaleNormal

```
1   package monteur;
2
3   import brique.*;
4
5   public class MonteurDedaleNormal implements MonteurDedale {
```

La classe MonteurDedalePiege

```
1   package monteur;
2
3   import brique.*;
4
5   public class MonteurDedalePiege implements MonteurDedale {
```

La classe JeuDuLabyrinthe

```
1   import monteur.*;
2   import brique.*;
3   import java.util.*;
4
5   public class JeuDuLabyrintheM {
6       private Random r = new Random();
7
8       public Dedale CreeDedale(MonteurDedale monteur) {
9           monteur.creeDedale();
10          for (int i=1; i<4; i++) {
11              monteur.creeSalle(i);
12              monteur.creeMur(Direction.est,monteur.acqDedale().salleNumero(i));
13              monteur.creeMur(Direction.ouest,monteur.acqDedale().salleNumero(i));
14              monteur.creeMur(Direction.sud,monteur.acqDedale().salleNumero(i));
15              monteur.creeMur(Direction.nord,monteur.acqDedale().salleNumero(i));
16          }
17          monteur.creePorte(Direction.est,
18                            monteur.acqDedale().salleNumero(1),
19                            monteur.acqDedale().salleNumero(2),
20                            r.nextBoolean());
21          monteur.acolleSalles(Direction.nord,
22                               monteur.acqDedale().salleNumero(1),
23                               monteur.acqDedale().salleNumero(3));
24          System.out.println(monteur.getName());
25          return monteur.acqDedale();
26      }
27
28      public static void main(String args[]){
29          JeuDuLabyrintheM jeu = new JeuDuLabyrintheM();
30          Dedale dedaleN = jeu.CreeDedale(new MonteurDedaleNormal());
31          Dedale dedaleP = jeu.CreeDedale(new MonteurDedalePiege());
32      }
33
34  }
```

Exercice 6.3 *Dessinez le graphe des classes.*

6.2.5 Les modèles structuraux

Ils étudient la façon de composer des classes et des objets pour réaliser des structures plus importantes. Les modèles de classes structurelles utilisent l'héritage pour composer interfaces et implémentations. Les modèles objets structuraux décrivent les moyens de composer des objets pour réaliser de nouvelles fonctionnalités.

Un prototype d'application

Un individu développe une application, la conception le conduit aux classes suivantes :

La classe MonApplication

```
1  package prototype;
2
3  public class MonApplication {
4
5      private MonObjectif monObjectif;
6
7      public MonApplication(){
8          monObjectif = new MonObjectif();
9      }
10
11     public void run(){
12         monObjectif.requete1();
13         monObjectif.requete2();
14     }
```

La classe MonObjectif

```
1  package prototype;
2
3  public class MonObjectif {
4
5      public void requete1(){
6          System.out.println("requete 1 non implémentée");
7      }
8
9      public void requete2(){
10         System.out.println("requete 2 non implémentée");
11     }
12
13 }
```

Une application propriétaire

Il trouve sur le marche un package fonctionnellement intéressant, mais malheureusement cela demande une refonte de sa conception.

La classe ApplicationPrivee

```
1  package prive;
2
3  public class ApplicationPrivee {
4      public ApplicationPrivee(){
5          System.out.println("création de l'application privee");
6      }
7      public void operation1 () {
8          System.out.println("opération 1 de l'application privee");
9      }
10     public void operation2 () {
11         System.out.println("opération 2 de l'application privee");
12     }
13     public void operation3 () {
14         System.out.println("opération 3 de l'application privee");
15     }
16 }
```

L'adaptateur

Intention convertit l'interface d'une classe en une autre conforme à l'attente du client. L'adaptateur permet à des classes de collaborer, qui n'auraient pu le faire du fait d'interfaces incompatibles.

Un exemple de code Deux applications dont une propriétaire que l'on ne peut modifier. De même il n'est pas souhaitable de modifier profondément notre application pour qu'elle puisse utiliser directement l'application propriétaire.

Les conséquences sur l'application

La classe MonApplication est inchangée
La classe MonObjectif devient un interface

```
1  package adaptateurClasse;
2
3  public interface MonObjectif {
4      public void requete1();
5      public void requete2();
6  }
```

L'application avec un adaptateur de classe

La classe Adaptateur

```
1  package adaptateurClasse;
2
3  import prive.*;
4
5  public class Adaptateur extends ApplicationPrivee implements MonObjectif {
6
7      public void requete1 () {
8          operation1();
9          operation2();
10     }
```

```
11
12      public void requete2 () {
13          operation2();
14          operation3();
15      }
16  }
```

L'application avec un adaptateur d'objet

La classe Adaptateur

```
1   package adaptateurObjet;
2
3   import prive.*;
4
5   public class Adaptateur implements MonObjectif {
6       private ApplicationPrivee appli;
7
8       public Adaptateur(){
9           appli = new ApplicationPrivee();
10      }
11
12      public void requete1 () {
13          appli.operation1();
14          appli.operation2();
15      }
16      public void requete2 () {
17          appli.operation2();
18          appli.operation3();
19      }
20  }
```

Exercice 6.4 *Dessinez le graphe des classes.*

6.2.6 Les modèles comportementaux

L'exemple d'un ordonnanceur de processus

Une première classe pour définir des exceptions.

La classe Impossible

```
1   package mesExceptions;
2
3   public class Impossible extends Exception {
```

```
4       public Impossible(){super(".");}
5       public Impossible(String st){super(st);}
6   }
```

Une première version de l'application.

La classe Processus

```
1   package versionInitiale;
2
3   import mesExceptions.*;
4
5   public class Processus {
6       static int numero = 0;
7       private int state;
8       private int ident;
9
10      public Processus(){
11          ident = numero++;
12          state = 0;
13          System.out.println("Création du processus "+ident);
14      }
15
16      public void elire () throws Impossible {
17          if (state == 0){
18              System.out.println("Election du processus "+ident);
19              state = 1;
20          }
21          else throw new Impossible("IMPOSSIBLE d'élire "+ident);
22      }
23
24      public void suspendre () throws Impossible {
25          if (state == 1){
26              System.out.println("Suspension du processus "+ident);
27              state = 0;
28          }
29          else throw new Impossible("IMPOSSIBLE de suspendre "+ident);
30      }
```

La classe Ordonnanceur

```
1   package versionInitiale;
2
3   import mesExceptions.*;
4   import java.util.*;
5
6   public class Ordonnanceur {
7       Processus[] processus;
8       int nbProcessus;
9
10      public Ordonnanceur(int nb){
11          processus = new Processus[nb];
12          for (int i=0; i<nb; i++)
13              processus[i] = new Processus();
14          nbProcessus = nb;
15      }
16
17      public void ordonne(){
```

```
18          int max = 6;
19          Random r = new Random();
20          for(int i=0; i<100; i++){
21              int p = r.nextInt(nbProcessus);
22              int n = r.nextInt(max);
23              try {
24                  switch(n){
25                  case 0 : processus[p].elire(); break;
26                  case 1 : processus[p].suspendre(); break;
27                  case 2 : processus[p].run(); break;
28                  case 3 : processus[p].termine(); break;
29                  case 4 : processus[p].debutES(); break;
30                  case 5 : processus[p].finES(); break;
31                  default :
32                  }
33              }
34              catch(Impossible e){
35                  System.out.println("\t"+e.getMessage());
36              }
37          }
38      }
39
40  }
```

La classe Test

```
1   import versionInitiale.*;
2
3   public class TestVI {
4
5       public static void main(String args[]){
6           Ordonnanceur ord = new Ordonnanceur(Integer.parseInt(args[0]));
7           System.out.println("Debut");
8           ord.ordonne();
9           System.out.println("Fin");
10      }
11
12  }
```

L'état

Intention L'état permet à un objet de modifier son comportement, quand son état interne change. Tout se passera comme si l'objet changeait de classe.

Implémentation Il existe une grande variété d'implantations : Qui décident du changement interne ? Un seul état avec création et destruction ou bien un ensemble et un état courant qui change ?

Une première version de l'ordonnanceur avec des états

La classe Processus

```
1   package etatVersion1;
2
3   import mesExceptions.*;
4
5   public class Processus {
6       static int numero = 0;
```

```
7       private int ident;
8       private Etat[] etats;
9       private Etat state;
10
11      public Processus(){
12          ident = numero++;
13          etats = new Etat[4];
14          etats[0] = new Etat0();
15          etats[1] = new Etat1();
16          etats[2] = new Etat2();
17          etats[3] = new Etat3();
18          state = etats[0];
19          System.out.println("Création du processus "+ident);
20      }
21
22      public void changerEtat(int s) {
23          state = etats[s];
24      }
25
26      public void elire () throws Impossible {
27          state.elire(this);
28      }
29
30      public void suspendre () throws Impossible {
31          state.suspendre(this);
32      }
```

La classe État

```
1   package etatVersion1;
2
3   import mesExceptions.*;
4
5   public class Etat {
6
7       public void elire (Processus p) throws Impossible {
8           throw new Impossible("IMPOSSIBLE d'élire "+p.getIdent());
9       }
10
11      public void suspendre (Processus p) throws Impossible {
12          throw new Impossible("IMPOSSIBLE de suspendre "+p.getIdent());
13      }
```

La classe État2 (un exemple de sous classe)

```
1   package etatVersion1;
2
3   public class Etat2 extends Etat{
4
5       public void finES (Processus p) {
6           System.out.println("Fin de l'appel système "+p.getIdent());
7           p.changerEtat(0);
8       }
9
10  }
```

La classe Ordonnanceur est inchangée

```
1  package etatVersion1;
```

La classe Test est inchangée

```
1  import etatVersion1.*;
```

Une seconde version de l'ordonnanceur avec des états

La classe Processus

```
1  package etatVersion2;
2
3  import mesExceptions.*;
4
5  public class Processus {
6      static int numero = 0;
7      private int ident;
8      private Etat state;
9
10     public Processus(){
11         ident = numero++;
12         state = new Etat0();
13         System.out.println("Création du processus "+ident);
14     }
15
16     public void changerEtat(Etat s) {
17         state = s;
18     }
19
20     public void elire () throws Impossible {
21         state.elire(this);
22     }
23
24     public void suspendre () throws Impossible {
25         state.suspendre(this);
26     }
```

La classe État est inchangée

```
1  package etatVersion2;
```

La classe État2 (un exemple de sous classe)

```
1  package etatVersion2;
2
3  public class Etat2 extends Etat{
4
5      public void finES (Processus p) {
6          System.out.println("Fin de l'appel système "+p.getIdent());
7          p.changerEtat(new Etat0());
8      }
9
10 }
```

La classe Ordonnanceur est inchangée

```
1  package etatVersion2;
```

La classe Test est inchangée

Exercice 6.5 *Dessinez le graphe des classes.*

L'observateur

Intention : définit une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendant en soient notifiés automatiquement et mis à jour.

Structure

Remarque : Ce modèle est suffisamment utilisé pour qu'il soit présent dans l'environnement Java.

Le texte suivant est extrait de la documentation sur l'interface `Observer`.

```
java.util
Interface Observer

public interface Observer

A class can implement the Observer interface when it wants to be informed of
changes in observable objects.

Method
        public void update(Observable o, Object arg)

        Parameters:
                o - the observable object.
                arg - an argument passed to the notifyObservers method.

        This method is called whenever the observed object is changed.
An application calls an Observable object's notifyObservers method
to have all the object's observers notified of the change.
```

et celui-ci de la documentation sur la classe `Observable`.

```
java.util
Class Observable

java.lang.Object
  |
```

```
  +--java.util.Observable

public class Observable
extends Object

This class represents an observable object, or "data" in the model-view
paradigm. It can be subclassed to represent an object that the application
wants to have observed.

An observable object can have one or more observers.
An observer may be any object that implements interface Observer.
After an observable instance changes, an application calling the
Observable's notifyObservers method causes all of its observers
to be notified of the change by a call to their update method.

Some methods
        public void addObserver(Observer o)
                Adds an observer to the set of observers for this object.

        public boolean hasChanged()
                Tests if this object has changed.

        public void notifyObservers(Object arg)
                If this object has changed, as indicated by the hasChanged
method, then notify all of its observers and then call the clearChanged
method to indicate that this object has no longer changed.
```

Un exemple de code : Une pyramide est posée sur un parallélépipède. On s'intéresse au volume global, ainsi qu'aux surfaces des trois projections suivant les trois axes. L'utilisateur peut à tout instant changer les points qui définissent la pyramide et le parallélépipède.

La classe Point3D

```
1   public class Point3D {
2       public int x,y,z;
```

La classe Parallelepipede

```
1   public class Parallelepipede
2       extends java.util.Observable
3       implements Volume {

19      public Integer getVolume(){
20          return new Integer(sommet.x * sommet.y * sommet.z) ;
21      }
22
23      public Integer getSurface(char axe){
24          if (axe=='x')
25              return new Integer(sommet.y * sommet.z);
26          else if (axe=='y')
27              return new Integer(sommet.x * sommet.z);
28          else if (axe=='z')
29              return new Integer(sommet.x * sommet.y);
30          else
31              return new Integer(0);
32      }
33
34      public void setEtat(int x, int y, int z){
```

```
35          sommet.x = x;
36          sommet.y = y;
37          sommet.z = z;
38          setChanged();
39          notifyObservers();
40      }
41
42      public void setX(int x){
43          sommet.x = x;
44          setChanged();
45          notifyObservers(new Character('x'));
46      }
```

La classe Pyramide

```
1   import java.awt.*;
2
3   public class Pyramide
4       extends java.util.Observable
5       implements Volume {
```

La classe ObserverPlan

```
1   import java.awt.*;
2   import java.util.*;
3
4   public class ObserverPlan implements Observer {
5       private Hashtable table;
6       private char axe;
7
8       public ObserverPlan(char axe){
9           this.axe = axe;
10          table = new Hashtable();
11      }
12
13      public void update(Observable o, Object arg){
14          Volume v = (Volume)o;
15          if ((arg==null) || (((Character)arg).charValue()!=axe)){
16              table.put(Integer.toString(v.hashCode()),v.getSurface(axe));
17              System.out.println("surface perpendiculaire à l'axe " +
18                                 axe + " = " + surfaceGlobale());
19          }
20      }
21
22      private int surfaceGlobale(){
23          int res = 0;
24          for (Enumeration e = table.elements() ; e.hasMoreElements() ;) {
25              res += ((Integer)e.nextElement()).intValue();
26          }
27          return res;
28      }
29
30  }
```

La classe ObserverVolume

```
1   import java.awt.*;
2   import java.util.*;
```

```
3
4   public class ObserverVolume implements Observer {
5       private Hashtable table;
6
7       public ObserverVolume(){
8           table = new Hashtable();
9       }
10
11      public void update(Observable o, Object arg){
12          Volume v = (Volume)o;
13          table.put(Integer.toString(v.hashCode()),v.getVolume());
14          System.out.println("volume = "+volumeGlobal());
15      }
16
17      private int volumeGlobal(){
18          int res = 0;
19          for (Enumeration e = table.elements() ; e.hasMoreElements() ;) {
20              res += ((Integer)e.nextElement()).intValue();
21          }
22          return res;
23      }
24
25  }
```

La classe Test

```
1   import java.util.*;
2
3   public class Test {
4
5       public static void main(String args[]){
6           if (args.length > 0){
7               int max = 6;
8               Random r = new Random();
9
10              Pyramide pyr = new Pyramide();
11              Parallelepipede par = new Parallelepipede();
12              Observer obsVol = new ObserverVolume();
13              Observer obsX = new ObserverPlan('x');
14              Observer obsY = new ObserverPlan('y');
15              Observer obsZ = new ObserverPlan('z');
16
17              pyr.addObserver(obsVol);
18              pyr.addObserver(obsX);
19              pyr.addObserver(obsY);
20              pyr.addObserver(obsZ);
21              par.addObserver(obsVol);
22              par.addObserver(obsX);
23              par.addObserver(obsY);
24              par.addObserver(obsZ);
25
26              System.out.println("Debut");
27              pyr.setEtat(2,3,4);
28              par.setEtat(4,5,6);
29              for(int i = 0; i < Integer.parseInt(args[0]); i++){
```

```
30                  int n=r.nextInt(max);
31                  switch(n){
32                  case 0 : pyr.setX(r.nextInt(max)); break;
33                  case 1 : pyr.setY(r.nextInt(max)); break;
34                  case 2 : pyr.setZ(r.nextInt(max)); break;
35                  case 3 : par.setX(r.nextInt(max)); break;
36                  case 4 : par.setY(r.nextInt(max)); break;
37                  case 5 : par.setZ(r.nextInt(max)); break;
38                  default :
39                  }
40              }
41              System.out.println("Fin");
42          }
43          else
44              System.out.println("java Test <nombreDeModifications>");
45      }
46  }
```

Exercice 6.6 *Dessinez le graphe des classes.*

6.2.7 Résumé

Fabrique Abstraite fournit une interface, pour créer des familles d'objets apparentés ou dépendants, sans avoir à spécifier leurs classes concrètes.

Adaptateur convertit l'interface d'une classe en une autre conforme à l'attente du client. L'adaptateur permet à des classes de collaborer, qui n'auraient pu le faire du fait d'interfaces incompatibles.

Pont découple une abstraction de son implémentation afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre.

Monteur dissocie, dans un objet complexe, sa construction de sa représentation, de sorte que, le même procédé de construction puisse engendrer des représentations différentes.

Chaîne de responsabilité d'une part évite le couplage de l'émetteur d'une requête à ses récepteurs, en donnant à plus d'un objet la possibilité d'entreprendre la requête; d'autre part chaîne les objets récepteurs et fait passer la requête tout au long de la chaîne, jusqu'à ce qu'un objet la traite.

Commande encapsule une requête comme un objet, autorisant ainsi le paramétrage des clients par différentes requêtes, files d'attente et récapitulatifs de requêtes, et de plus, permettant la réversion des opérations.

Composite compose des objets en des structures arborescentes pour représenter des hiérarchies composant/composé. Il permet au client de traiter de la même et unique façon les objets individuels et les combinaisons de ceux-ci.

Décorateur attache dynamiquement des responsabilités supplémentaires à un objet. Les décorateurs fournissent une alternative souple à la dérivation, pour étendre les fonctionnalités.

Façade fournit une interface unifiée, à l'ensemble des interfaces d'un sous-systèmes. La façade fournit une interface de plus haut niveau, qui rend le sous-systèmes plus facile à utiliser.

Fabrication définit une interface pour la création d'un objet, tout en laissant à des sous-classes le choix de la classe à instancier. Une fabrication permet de déférer à des sous-classes les instanciations d'une classe.

Poids Mouche utilise une technique de partage qui permet la mise en œuvre efficace d'un grand nombre d'objets de fine granularité.

Interpréteur définit, pour un langage donné, une représentation de sa grammaire, en même temps qu'un interpréteur utilisant cette représentation pour interpréter les phrases du langage.

Itérateur fournit un moyen d'accès séquentiel, aux éléments d'un agrégat d'objets, sans mettre à découvert la représentation interne de celui-ci.

Médiateur définit un objet qui encapsule les modalités d'interaction d'un certain ensemble d'objets. Le médiateur favorise le couplage faible en dispensant les objets de se faire explicitement référence, et il permet donc de faire varier indépendamment les relations d'interaction.

Mémento saisi et transmet à l'extérieur d'un objet, sans violation de l'encapsulation, l'état interne de celui-ci, dans le but de pouvoir ultérieurement le restaurer dans cet état.

Observateur définit une interdépendance de type un à plusieurs, de façon telle que, quand un objet change d'état, tous ceux qui en dépendant en soient notifiés automatiquement et mis à jour.

Prototype spécifie les espèces d'objets à créer, en utilisant une instance de type prototype, et crée de nouveaux objets par copies de ce prototype.

Procuration fournit à un tiers objet un mandataire ou un remplaçant, pour contrôler l'accès à cet objet.

Singleton garantit qu'une classe n'a qu'une seule instance, et fournit à celle-ci, un point d'accès de type global.

État permet à un objet de modifier son comportement, quand son état interne change. Tout se passera comme si l'objet changeait de classe.

Stratégie définit une famille d'algorithmes, encapsule chacun d'entre eux, et les rend interchangeables. Le modèle stratégie permet aux algorithmes d'évoluer indépendamment des clients qui les utilisent.

Patron de méthode définit, dans une opération, le squelette d'un algorithme, en en déléguant certaines étapes à des sous-classes. Le patron de méthode permet de redéfinir par des sous-classes, certaines parties d'un algorithme, sans avoir à modifier la structure de ce dernier.

Visiteur fait la représentation d'une opération applicable aux éléments d'une structure d'objet. Il permet de définir une nouvelle opération, sans qu'il soit nécessaire de modifier la classe des éléments sur lesquels elle agit.

6.3 Mode d'emploi

6.3.1 Trouver les objets appropriés

Les méthodologies classiques de conception conduisent souvent à une modélisation rigoureuse du monde réel (c-à-d le système étudié) qui traduit les réalités du moment.

Les modèles de conception permettent d'identifier des concepts moins évidents (ie un processus, un algorithme). Dans une approche traditionnelle, ces objets n'apparaissent en général que lors du remaniement d'une conception pour la rendre plus souple et plus réutilisable.

6.3.2 Spécification des interfaces d'objets

Les objets ne sont connus qu'à travers leurs interfaces. Deux objets, dotés d'implémentation complètement différentes, peuvent avoir des interfaces identiques. La liaison dynamique permet l'association à l'exécution d'une requête à un objet.

Les modèles de conception d'une part aident à la définition des interfaces ; d'autre part précisent également les relations entre interfaces.

6.3.3 Comparaison entre héritage de classe et héritage d'interface

L'héritage de classe définit l'implémentation d'un objet à partir de l'implémentation d'un autre objet. C'est un mécanisme de partage de code et de la représentation.

L'héritage d'interface (sous-typage), décrit comment un objet peut être utilisé à la place d'un autre. Il réduit de façon significative l'interdépendance des implémentations d'un sous-systèmes à l'autre.

Principe : Programmer pour une interface, non pour un développement.

Cela conduit d'une part à ne pas déclarer les variables comme instances de classes concrètes, mais comme se conformant à une interface ; d'autre part à l'utilisation des modèles créateurs.

Les deux techniques les plus courantes pour la réutilisation sont :

- L'héritage de classe qui est de la réutilisation **boîte blanche**, car le contenu des classes parentes est généralement visible et souvent modifiable.
- La composition d'objets qui est de la réutilisation **boîte noire** car aucun détail des objets n'est visible. La composition n'utilise que les interfaces, qui doivent par conséquent être bien définies. La délégation (un objet récepteur délègue les opérations à son délégué) est un moyen qui fait de la composition un outil de réutilisation aussi puissant que l'héritage.

Principe : Préférer la composition d'objets à l'héritage de classes.

6.3.4 Concevoir évolutif

Il n'y a pas vraiment de solution, mais des erreurs à éviter :

- Créer un objet en spécifiant explicitement une classe. *modèles créateurs.*
- Assujettissement à une opération particulière. *modèles chaîne de responsabilités et commande*
- Dépendance vis à vis des plates-formes matérielles et logicielles.
- Assujettissement à la représentation d'un objet ou à son code.
- Assujettissement à un algorithme. *modèles itérateur, stratégie ...*
- Couplage fort entre classes.
- Extension des fonctionnalités par sous-classes.
- Impossibilité de modifier convenablement les classes. *modèle adaptateur ...*

Chapitre 7

Les objets distribués

7.1 Développement Client/Serveur

7.1.1 Le modèle deux parties

La forme la plus élémentaire d'une architecture client/serveur comporte deux parties :
- un système de stockage des données centralisé.
- des systèmes clients.

Exemple : le logiciel de navigation de la machine cliente extrait des informations entreposées sur un serveur web.

Limites : Il est difficile au client de manipuler les données envoyés par le serveur. Le client est très souvent limité à un rôle de présentation de données.

7.1.2 Le modèle trois parties

Une telle architecture se compose de :
- un système de stockage des données centralisé : BD, fichiers...
- un serveur d'application.
- des systèmes clients.

Avantages : seul le serveur manipule les données, car les clients n'ont accès qu'aux applications. Les données peuvent être déplacées et peuvent provenir de plusieurs sources de stockages sans que les clients soient avertis.

7.2 Les modèles d'objets

Le fait qu'un même objet puisse être manipulé à distance et simultanément par plusieurs clients demande que cet objet possède des propriétés particulières afin que le système reste cohérent.

7.2.1 La sérialisation des objets

Lorsqu'un objet doit franchir les limites d'une machine virtuelle, le passage se fait par valeur, à travers sa sérialisation.

7.2.2 Le modèle d'observation

Un objet, dit observateur, doit guetter les modifications apportées à un autre objet, dit observable.

Remarque : le modèle qui existe en Java à travers l'interface *java.util.Observer* et et la classe *java.util.Observable* doit être adapté pour les objets distants.

7.2.3 Le modèle de persistance

Il est nécessaire de préserver les données d'une instance d'une application afin d'en faire profiter de nouvelles instances. Les Bases de données et les fichiers garantissent généralement la persistance. La notion de transaction est généralement introduite.

7.2.4 Le modèle peer

Les objets partagés par les clients ne doivent en aucun cas être dépendants des plates-formes. Une fois déplacé chez un client, il doivent par liaison dynamique utilisé la bonne classe "peer".

Remarque : mécanisme de base en Java.

7.2.5 Le modèle de fabrique

Les clients souhaitent que ce soient les objets qui s'adaptent à eux et non l'inverse. Notamment, la langue, les couleurs . . ..

Le modèle de fabrique et l'utilisation de *ressources locales* permet de réaliser ce souhait.

7.3 l'API RMI

Le but de cette API est de rendre transparent l'accès aux objets distribués sur un réseau, afin que les appels de méthodes distantes s'effectuent comme les appels de méthodes sur les objets locaux.

Le principal problème est l'obtention d'une référence d'un objet qui se trouve dans une autre machine virtuelle.

7.3.1 Accéder aux objets distants

Sur la machine qui détient l'objet, deux applications particulières :

- un programme (rmiregistry) dédié dont le rôle est d'écouter sur un port particulier d'une machine (1099 par défaut pour RMI) les requêtes des clients.
- le serveur d'applications qui va lié l'objet en sa possession (en indiquant sa localisation) à ce même port.

Sur la machine cliente, il suffit alors de préciser le triplet (machine, port, objet) pour l'obtenir.

7.3.2 Interfaces distantes

Tout objet accédé à distance doit être déclaré comme tel. Pour cela il doit implémenter une interface particulière *java.rmi.remote*

La seule contrainte que cela implique est que toute les méthodes doivent prévoir les pannes possibles du réseau *throws RemoteException.*

7.3.3 Stubs et squelettes

Les objets exportés doivent posséder deux classes particulières *objet_Stub.class et objet_Skel.class* qui doivent être présentes sur les deux sites.

- objet_Stub.class est un objet spécial, qui implémente toutes les interfaces de l'objet distant de la façon suivante :
  1. convertit les paramètres de la méthode,
  2. construit un message suivant un certain protocole (TCP/IP, UDP/IP)
  3. transmet le message via le réseau à l'adresse (machine, port).
- objet_Skel.class est un objet spécial qui :
  1. réceptionne le message,

2. reconstruit les paramètres en tant qu'objet.
3. appelle la méthode visée de l'objet.

Les rôles sont inversés lors de l'acheminement des valeurs de retour.
Remarque : le programme *rmic* fabrique les classes *Stub et Skel*.

7.4 Un exemple

L'accès à un objet *Personne* et la possibilité de modifier son *nom*

L'interface Personne

```
1  import java.rmi.*;
2
3  public interface Personne extends Remote {
4
5      public void setName(String s) throws RemoteException;
6      public String getName() throws RemoteException;
7      public int getIdent() throws RemoteException;
8
9  }
```

L'implantation de Personne

```
1  import java.rmi.*;
2  import java.rmi.server.UnicastRemoteObject;
3
4  public class PersonneImpl
5      extends UnicastRemoteObject
6      implements Personne {
7
8      private static int numero=0;
9      private String name;
10     private int ident;
11
12     public PersonneImpl(String name) throws RemoteException {
13         this.name = name;
14         ident = numero;
15         numero++;
16     }
17
18     public void setName(String name) throws RemoteException {
19         this.name = name;
20     }
21
22     public String getName() throws RemoteException {
23         return name;
24     }
25
26     public int getIdent() throws RemoteException {
27         return ident;
28     }
29
30 }
```

L'interface Serveur

```
1  import java.rmi.*;
2
3  public interface Serveur
4      extends Remote {
5
6      public Personne getPersonne(String s) throws RemoteException;
7
8  }
```

L'implantation de Serveur

```
1  import java.net.*;
2  import java.rmi.*;
3  import java.rmi.server.UnicastRemoteObject;
4
5  public class ServeurImpl
6      extends  UnicastRemoteObject
7      implements Serveur {
8
9      Personne [] pers;
10
11     public ServeurImpl(String port) throws RemoteException {
12         super();
13         initialiseBD();
14         try {
15             // Get the inet address
16             String host=InetAddress.getLocalHost().getHostAddress();
17             // Create and install a security manager
18             //if (System.getSecurityManager() == null) {
19             //System.setSecurityManager(new RMISecurityManager());
20             //}
21             Naming.rebind("rmi://"+host+":"+port+"/Serveur", this);
22             System.out.println("Serveur prêt");
23         }
24         catch(Exception e) {
25             System.out.println("exception : " +e);
26         }
27     }
28
29     public void initialiseBD() throws RemoteException {
30         pers = new Personne[2];
31         pers[0] = new PersonneImpl("Domenger");
32         pers[1] = new PersonneImpl("Griffault");
33     }
34
35     public Personne getPersonne(String s) throws RemoteException {
36         for (int i =0; i < pers.length; i++)
37             if (s.equalsIgnoreCase(pers[i].getName())) return pers[i];
38         return null;
39     }
40
41     static public void main(String [] argv ) {
42         boolean pasArgument = argv.length != 1;
43
44         if (pasArgument)
```

```
45              System.out.println("java ServeurImpl NumeroPort");
46          else {
47              try {
48                  new ServeurImpl(argv[0]);
49              }
50              catch(Exception e) {
51                  System.out.println("exception : " +e);
52              }
53          }
54      }
55  }
```

La classe Client

```
 1  import java.rmi.*;
 2  import java.rmi.server.UnicastRemoteObject;
 3
 4  public class Client {
 5
 6      Serveur sref;
 7      Personne pers;
 8
 9      public Client(String host, String port) {
10          try{
11              sref = (Serveur)Naming.lookup("//"+host+
12                                            ":"+port+
13                                            "/Serveur");
14          }
15          catch(Exception e){
16              System.out.println("exception : " +e);
17          }
18      }
19
20      void replacePersonne(String ancien, String nouveau) {
21          try{
22              pers = sref.getPersonne(ancien);
23              if (pers != null){
24                  System.out.println("Old : "+
25                                     "\n\tident : "+pers.getIdent()+
26                                     "\n\tNom   : "+pers.getName());
27                  pers.setName(nouveau);
28                  System.out.println("New : "+
29                                     "\n\tident : "+pers.getIdent()+
30                                     "\n\tNom   : "+pers.getName());
31              }
32              else
33                  System.out.println(ancien + " : Inconnu.");
34          }
35          catch(Exception e){
36              System.out.println("exception : " +e);
37          }
38      }
39
40      static public void main(String [] argv) {
41          boolean pasArgument = argv.length != 4;
```

```
42
43          if (pasArgument)
44              System.out.println("java Client host port AncienNom NouveauNom");
45          else {
46              Client client = new Client(argv[0],argv[1]);
47              client.replacePersonne(argv[2],argv[3]);
48          }
49      }
50  }
```

Un applet utilisateur

```
1     <HTML>
2       <title>Client</title>
3       <center> <h1>Client</h1> </center>
4       <applet code="Application"
5               width=300 height=400>
6       </applet>
7     </HTML>
8
9
```

7.4.1 Mise en œuvre

```
javac *java
rmic ServeurImpl PersonneImpl
```

Sur la machine “serveur” :

```
rmiregistry <port> &
java ServeurImpl <port> &
```

Sur la machine “client” :

```
java Client <host> <port> <ancienNom> <nouveauNom>
```

Chapitre 8

Les accès aux bases de données

8.1 Applications OO et SGBD

Développement client/serveur trois parties.

Le modèle d'observation pour avoir des vues cohérentes.

Le modèle de persistance : en général, on utilise celui des SGBD, mais il est possible de faire ses propres classes d'objets persistant :

- abort()
- remove()
- restore()
- save()

8.2 Accéder aux bases de données grâce à JDBC

L'API JDBC permet de formuler des requêtes SQL au sein des sources JAVA.

JDBC propose un ensemble d'interfaces et de classes. L'ensemble des classes d'implantation pour un SGBD est appelé pilote JDBC. On distingue quatre classes de pilote (plus ou moins évolué, le code est plus ou moins portable).

type 1 : les pilotes accèdent aux données grâce à une technologie de ponts. Les ponts requièrent en général d'installer du code natif coté client. Exemple : le pont ODBC.

type 2 : les pilotes appellent des méthodes C/C++ natives livrées par les éditeurs de SGBD. Il faut donc du code natif chez le client.

type 3 : les pilotes fournissent au client une API réseau générique qui, traduite au niveau du serveur, permet d'accéder aux bases de données. Le pilote utilise des sockets avec une application intermédiaire sur le serveur; cette application convertit les requêtes en appel à l'API spécifique du pilote souhaité. Ces pilotes ont l'avantage de pouvoir interagir avec plusieurs SGBD différents.

type 4 : les pilotes interagissent directement avec le SGBD. Ils sont en pur Java, portables et représentant souvent la solution "optimale". Ils sont exclusivement fournis par les éditeurs de SGBD.

8.2.1 les interfaces de JDBC

Driver : tous les pilotes doivent respecter cet interface.

Statement : la méthode `execute` sert pour exécuter les requêtes SQL. Deux sous-interfaces `PreparedStatement et CallableStatement` pour les instructions prédéfinies et les procédures stockées.

Connection : pour établir la connexion à l'instance de la base de données.

ResultSet : pour stocker le résultat (un ensemble) d'une requête SQL.

ResultSetMetaData et DatabaseMetaData : pour accéder aux MetaTables du SGBD.

8.2.2 Se connecter et se déconnecter à la base de données

De l'utilisation des méthodes `Class.forName()`, `DriverManager.getConnection()` et de méthode `close()` de la classe `Connection`.

La classe TablesBD

```
 8      public static Connection etablirConnexion(String pilote,
 9                                                String url,
10                                                String uid,
11                                                String mdp) {
12          Connection connexion = null;
13
14          try {
15              Class.forName(pilote);
16          }
17          catch(Exception e){
18              System.out.println("Impossible de charger le pilote");
19              return null;
20          }
21          try {
22              connexion = DriverManager.getConnection(url, uid, mdp);
23              System.out.println("Connexion etablie.");
24          }
25          catch(SQLException e){
26              System.out.println("Impossible d'etablir la connexion");
27              return null;
28          }
29          return connexion;
30      }
31
32      public static void fermerConnexion(Connection c) {
33          try {
34              c.close();
35              System.out.println("Connexion terminee");
36          }
37          catch(SQLException e){
38              System.out.println("Impossible de fermer la connexion...Tant pis");
39              return;
40          }
41      }
```

8.2.3 Les interactions simples

De l'utilisation des méthodes `createStatement()` et `getResultSet()`.

La classe TablesBD

```
121     public static void traiterRequete(Connection c,
122                                       String requete) {
123         try {
124             Statement s = c.createStatement();
125             if (s.execute(requete)) {
126                 ResultSet res = s.getResultSet();
127                 TablesBD.sortieResultSet(res);
128             }
129             else {
```

```
130                 int res = s.getUpdateCount();
131                 System.out.println("Modification reussie de "+res+" lignes.");
132             }
133         }
134         catch(SQLException e){
135             System.out.println("Impossible de traiter la requete");
136         }
137     }
```

8.2.4 Les transactions

De l'utilisation des méthodes `getAutoCommit()`, `setAutoCommit()` et `commit()`.

La classe TablesBD

```
139     public static void traiterTransaction(Connection c,
140                                           String[] requete) {
141         try {
142             boolean commit = c.getAutoCommit();
143             c.setAutoCommit(false);
144             Statement s = c.createStatement();
145             for (int i=0; i< requete.length; i++)
146                 s.execute(requete[i]);
147             c.commit();
148             c.setAutoCommit(commit);
149         }
150         catch(SQLException e){
151             System.out.println("Impossible de traiter la transaction");
152             try {
153                 c.rollback();
154             }
155             catch(SQLException e2){
156                 System.out.println("Impossible de traiter le ROLLBACK !");
157             }
158         }
159     }
```

8.3 Optimiser l'accès aux BD

De l'utilisation des méthodes `prepareStatement()` et `prepareCall()`.

Deux types d'optimisation : les instructions prédéfinies (le plan d'accès aux tables n'est fait qu'un fois), et les procédures stockées (compilées dans le SGBD).

8.3.1 Les instructions prédéfinies

Dans le cas d'un ensemble de requêtes similaires, il est préférable, au lieu de répéter N fois `s.execute(requete)`, de signaler au SGBD qu'il va devoir faire N fois la même chose sur des données différentes. Ainsi le SGBD ne recalcule pas son plan d'exécution entre deux requêtes successives.

```
PreparedStatement ps = connection.preparedStatement(requeteAvecParametres);
for (int i=0; i < N; i++) {
    ps.setFloat(1, 3.14159));
    ps.setInt(2, 5678);
    ps.execute();
}
```

```
connection.commit();
s.close();
```

8.3.2 Les procédures stockées

Il est possible de faire appel directement à une procédure stockée, si l'on connaît son nom, ses entrées et ses sorties.

```
CallableStatement cs = connection.preparedCall("{call procStockee[(?,?)]}");
cs.registerOutParameter(1,java.sql.Types.FLOAT);
for (int i=0; i < N; i++) {
    cs.setInt(2,parametre2());
    cs.execute();
}
connection.commit();
s.close();
```

8.4 Les accès dynamiques

De l'utilisation des interfaces `ResultSetMetaData` et `DatabaseMetaData`.

Les metaClasses ResultSetMetaData et DatabaseMetaData permettent l'accès aux informations colonnes, tables, propriétaires, droits....

La classe TablesBD

```
 99      private static void sortieResultSet(ResultSet res){
100          try {
101              ResultSetMetaData metaRes = res.getMetaData();
102              int cols = metaRes.getColumnCount();
103              for (int i=1; i <= cols; i++)
104                  TablesBD.sortieFormatee(metaRes.getColumnLabel(i),
105                                          metaRes.getColumnDisplaySize(i));
106              System.out.println();
107              while(res.next()){
108                  for (int i=1; i <= cols; i++)
109                      TablesBD.sortieFormatee(res.getObject(i).toString(),
110                                              metaRes.getColumnDisplaySize(i));
111                  System.out.println();
112              }
113          }
114          catch(SQLException e){
115              System.out.println("Impossible d'afficher le resultat");
116              e.printStackTrace();
117              return;
118          }
119      }
```

Table des matières

1 Introduction **3**
- 1.1 Bibliographie . . . 3
- 1.2 Les origines . . . 3
- 1.3 Limite de la programmation structurée . . . 3
- 1.4 Pourquoi les objets ? . . . 3
- 1.5 Perspectives . . . 4

2 Les concepts objets **5**
- 2.1 Les objets et les relations inter-objets . . . 5
- 2.2 Les classes et les relations interclasses . . . 6
  - 2.2.1 Les classes . . . 6
  - 2.2.2 Le graphe de généralisation . . . 6
  - 2.2.3 Le graphe d'instanciation . . . 7
  - 2.2.4 Le graphe d'association et d'agrégation . . . 7
  - 2.2.5 Les collections . . . 8
- 2.3 L'héritage . . . 8

3 Les langages objets **11**
- 3.1 Introduction . . . 11
- 3.2 Les langages comparés . . . 11
- 3.3 Les critères de comparaison . . . 11
- 3.4 Lequel choisir ? . . . 12

4 Architecture d'une application Java **13**
- 4.1 La modularité : les packages . . . 13
  - 4.1.1 La visibilité des classes . . . 13
- 4.2 La modélisation . . . 14
  - 4.2.1 Les attributs d'un objet . . . 14
  - 4.2.2 Les opérations d'un objet . . . 14
  - 4.2.3 La référence aux objets . . . 14
- 4.3 L'encapsulation des objets . . . 15
  - 4.3.1 L'encapsulation des attributs et des méthodes . . . 15
- 4.4 L'héritage . . . 15
- 4.5 Liaison dynamique et polymorphisme . . . 16
- 4.6 Les exceptions . . . 17
  - 4.6.1 Un premier exemple . . . 17
  - 4.6.2 Gérer ses propres exceptions . . . 18
- 4.7 Le parallélisme . . . 19
- 4.8 La gestion mémoire . . . 19

5 Les méthodes de développement logiciel **21**
5.1 Les phases classiques du développement . . . 21
5.2 Les cycles classiques du développement . . . 21
5.3 Les méthodes de conception . . . 22
5.4 L'approche objets . . . 22
5.4.1 Les méthodes objets . . . 22
5.4.2 Les modèles et techniques objets . . . 22
5.4.3 Limites des approches objets . . . 23

6 Les catalogues de conception **25**
6.1 Les interfaces Homme-machine . . . 25
6.1.1 Réalisation d'interfaces dans Smalltalk . . . 25
6.2 Les modèles de conception . . . 26
6.2.1 Qu'est ce qu'un modèle de conception . . . 26
6.2.2 Catalogue et classification des modèles de conception . . . 27
6.2.3 Description des modèles de conception . . . 27
6.2.4 Les modèles créateurs . . . 27
6.2.5 Les modèles structuraux . . . 33
6.2.6 Les modèles comportementaux . . . 35
6.2.7 Résumé . . . 44
6.3 Mode d'emploi . . . 45
6.3.1 Trouver les objets appropriés . . . 45
6.3.2 Spécification des interfaces d'objets . . . 45
6.3.3 Comparaison entre héritage de classe et héritage d'interface . . . 46
6.3.4 Concevoir évolutif . . . 46

7 Les objets distribués **47**
7.1 Développement Client/Serveur . . . 47
7.1.1 Le modèle deux parties . . . 47
7.1.2 Le modèle trois parties . . . 47
7.2 Les modèles d'objets . . . 47
7.2.1 La sérialisation des objets . . . 47
7.2.2 Le modèle d'observation . . . 47
7.2.3 Le modèle de persistance . . . 48
7.2.4 Le modèle peer . . . 48
7.2.5 Le modèle de fabrique . . . 48
7.3 l'API RMI . . . 48
7.3.1 Accéder aux objets distants . . . 48
7.3.2 Interfaces distantes . . . 48
7.3.3 Stubs et squelettes . . . 48
7.4 Un exemple . . . 49
7.4.1 Mise en œuvre . . . 52

8 Les accès aux bases de données **53**
8.1 Applications OO et SGBD . . . 53
8.2 Accéder aux bases de données grâce à JDBC . . . 53
8.2.1 les interfaces de JDBC . . . 53
8.2.2 Se connecter et se déconnecter à la base de données . . . 54
8.2.3 Les interactions simples . . . 54
8.3 Optimiser l'accès aux BD . . . 55
8.3.1 Les instructions prédéfinies . . . 55
8.3.2 Les procédures stockées . . . 55
8.4 Les accès dynamiques . . . 55