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

Alain Griffault

5 janvier 2000

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

Un distributeur de boissons

2.1 Introduction

Cet exercice est inspiré de la vie de tous les jours, et servira de fil conducteur pour le cours et les travaux dirigés.

2.2 Un distributeur de boissons chaudes

Un constructeur désire construire un distributeur de café et de chocolat chauds. Voici une liste de contraintes auxquelles il a pensé.

1. le client pourra choisir parmi les boissons suivantes :
   - café non sucré,
   - café sucré,
   - chocolat non sucré,
   - chocolat sucré.
2. le prix des consommables sera (dans une unité quelconque) :
   - café : 2,
   - chocolat : 3,
   - sucre : 1,
   - eau et gobelet : gratuits.
3. la machine acceptera les pièces de 1,2 et 5 unités.
4. la machine rendra la monnaie.
5. le client pourra au choix : choisir une boisson puis mettre l'argent en quantité suffisante, ou bien mettre l'argent puis choisir.
6. le client pourra à tout instant annuler ses choix, la machine reviendra alors dans son état initial, et rendra notamment les pièces déjà introduites.
7. un opérateur pourra "de temps en temps" effectuer les actions suivantes :
   - vider le monayeur,
   - ajouter du café,
   - ajouter du chocolat,

   - ajouter du sucre,
   - ajouter des gobelets.

8. la machine sera raccordée directement au service de distribution en eau, que l'on supposera toujours opérationnel.

2.3 Les spécifications

Le constructeur est un homme honnête, mais pas philanthrope. Il désire donc que sa machine vérifie les propriétés suivantes :

1. La machine ne délivre pas de boissons si l'argent correspondant n'a pas été introduit.
2. La machine ne garde pas plus d'argent que le prix de la boisson servie.
3. La machine ne remplace pas une boisson par une autre sans l'accord du client.
4. La machine est opérationnelle tant qu'elle peut rendre au moins un des services.

Exercice 2.1 *Donnez un graphe entité-association reliant les consommables, les boissons, les clients et l'opérateur.*

Chapitre 3

Les concepts objets

3.1 Les objets et les relations inter-objets

Définition 3.1 *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.

$(o1 == o2) \Leftrightarrow (o1.identit\acute{e} = o2.identit\acute{e})$

$(o1 = o2) \Leftrightarrow (o1.\acute{e}tat = o2.\acute{e}tat)$

Les liens entre les objets peuvent être représentés par des références aux identités des objets. Un lien monovalué est représenté par un attribut dans l'objet source, dont la valeur est la réference 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 3.1 *Donnez au format (idendite, type d'objet, état) des exemples* *d'objets, et dessinez le graphe associé.*

3.2 Les classes et les relations interclasses

3.2.1 Les classes

Définition 3.2 *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 3.2 *Donnez des exemples de classes.*

3.2.2 Le graphe de généralisation

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

Définition 3.3 *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 3.3 *Ajouter une classe personne, dont client et opérateur deviennent des sous-classes.*

3.2.3 Le graphe d'instanciation

Définition 3.4 *Instanciation : relation entre un objet et sa classe d'appartenance qui a permis de la créer.*

Définition 3.5 *Métaclasse : 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 métaclasse est sa propre instance, et la classe object est la racine de la hiérarchie

Exercice 3.4 *Dessinez le graphe des classes avec metaclasse et object*

3.2.4 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 3.6 *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 3.7 *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 3.5 *Dessinez un graphe d'association et d'agrégation.*

3.2.5 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 3.8 *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.*

3.3 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 3.9 *Héritage simple : mécanisme de transmission des propriétés d'une classe vers une sous-classe.*

Définition 3.10 *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.*

Définition 3.11 *Classe abstraite = attributs + signature des opérations.*

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

Définition 3.12 *Surcharge : redéfinition d'une opération avec un code différent.*

Définition 3.13 *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 3.14 *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 3.6 *Le propriétaire est un client qui ne paye pas, et qui cherchera à contrôler l'honneteté des clients et de ses opérateurs. Donnez un graphe d'héritage et de la surcharge d'opération.*

Chapitre 4

Les langages objets

4.1 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.

4.2 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'interactivité 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.

4.3 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, génériсite, classe abstraite, métaclasse ...

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 ?

4.4 Lequel choisir ?

Bien évidemment, le tableau 4.1 ne répond pas à la question, tout juste permet-il de comprendre certains choix.

TAB. 4.1: 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	non
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
Moniteur	RdV	oui	non	non	non		non	oui
Gestion mémoire								
Chargeur dynamique		non	non	oui	non			oui
Ramasse-miettes		oui	non	oui	oui	oui	non	oui

Chapitre 5

La notation UML

5.1 Les cycles classiques du développement

Le modèle en cascade date des années 1970, 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 qui permet de distinguer la vérification des composants et la validation du système, mais celle-ci apparait 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 bein que décrit que le plus adapté aux objets, aucune méthodes objets n'y fait explicitement référence.

Le modèle tridimentionnel intègre 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.

5.2 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'intéractions non hiérarchique dans le cas des systèmes complexes.

Les approches systèmiques 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.

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

5.3 L'approche objets

5.3.1 Les méthodes objets

Les méthodes objets sont basées sur une triple perception du système : une dimension statique qui décrits 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.

5.3.2 Les modèles et techniques objets

Chaque méthode utilisent ses propres modèles formels, la pluspart 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 intéractions 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
- **Les règles actives** décrivent les réactions du système (comme les triggers dans les SGBD), dans un formalisme qui généralise le "switch".
- **Les statecharts** sont des diagrammes états/transitions hiérarchisés.
- **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 modèles fonctionnels utilisent fréquemment les diagrammes de flux de données. Il est également possible d'utiliser les réseaux de Petri pour décrire le modèle fonctionnel d'un système.

Exercice 5.1 *Donnez un modèle dynamique pour les consommables et pour les* *boissons.*

5.3.3 Limites des approches objets

Aucune des méthodes ne couvre entiérement le cycle de développement, chacune ayant ses qualités et ses défauts. Plus grâve, 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.

5.4 La notation UML

Un survol de la notation UML par Pierre-Alain Muller.

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 selection dans les menus où l'activité de la souris.

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.

FIG. 6.1: Le modèle MVC de smalltalk

FIG. 6.2: Les liens dans le modèle MVC de smalltalk

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 Cataloque et classification des modèles de conception

		Rôle		
		Créateur	Structurel	Comportement
Domaine	Classe	Fabrication	Adapdateur	Interprète Patron de méthode
	Objet	Fabrique abstraite Monteur Prototype Singleton	Adapdateur Pont Composite Décorateur Façade Poids mouche Procuration	Chaine de responsabilité Commande Itérateur Médiateur Memento Observateur Etat 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.

Un exemple : un labyrinthe

Les sources suivantes sont une première implémentation pour la création d'un labyrinthe.

```
import java.util.*;

public class Dedale {

    private Vector plan;

    public Dedale(){
        plan = new Vector(5,0);
    }

    public void ajouteSalle(Salle s){
plan.add(s);
    }

    public void acolleSalles(int d1, Salle s1, Salle s2){
Mur m1 = (Mur)s1.acqCote(d1);
Mur m2 = (Mur)s2.acqCote(Direction.opposee(d1));
m1.accolleMur(m2);
m2.accolleMur(m1);
    }

    public Salle salleNumero(int num){
for (int i = 0; i< plan.size(); i++)
    if (((Salle)plan.elementAt(i)).acqNumero() == num)
return (Salle)plan.elementAt(i);
return null;
    }
}



public class Direction {

    public static final int est = 0;
    public static final int nord = 1;
    public static final int ouest = 2;
    public static final int sud = 3;

    public Direction(){
```

```
    }

    public static int opposee(int d) {
switch (d) {
case est : return ouest;
case ouest : return est;
case nord : return sud;
case sud : return nord;
default : return 0;
}
    }

}
```

```
import java.util.*;

public class JeuDuLabyrinthe {

    private Random r = new Random();

    public JeuDuLabyrinthe(){
    }

    public Dedale CreeDedale() {
Dedale unDedale = new Dedale();
Salle s;
Mur me,mo,ms,mn;
for (int i=1; i<4; i++) {
    s = new Salle(i);
    unDedale.ajouteSalle(s);
    me = new Mur(Direction.est,s);
    mo = new Mur(Direction.ouest,s);
    ms = new Mur(Direction.sud,s);
    mn = new Mur(Direction.nord,s);
}
Porte laPorte = new Porte(Direction.est,
  unDedale.salleNumero(1),
  unDedale.salleNumero(2),
  r.nextBoolean());
unDedale.acolleSalles(Direction.nord,
      unDedale.salleNumero(1),
      unDedale.salleNumero(3));
System.out.println("dedale cree");
return unDedale;
    }

    public static void main(String args[]){
System.out.println("Debut");
```

```
JeuDuLabyrinthe jeu = new JeuDuLabyrinthe();
jeu.CreeDedale();
System.out.println("Fin");
    }

}



public abstract class Lieu {

    public abstract boolean entrer();

}



public class Mur extends Lieu {

    private int direction;
    private Salle salle;
    private Mur mitoyen = null;

    public Mur(){
direction = -1;
salle = null;
    }

    public Mur(int d, Salle s){
direction = d;
salle = s;
s.installeCote(d,this);
    }

    public void poserMur(int d, Salle s){
direction = d;
salle = s;
s.installeCote(d,this);
    }

    public boolean entrer(){
return false;
    };

    public Salle acqSalle(){
return salle;
    }

    public int acqDirection(){
return direction;
    }
```

```
    public void accolleMur(Mur m){
mitoyen = m;
    }

    public Mur acqMitoyen(){
return mitoyen;
    }

}

public class Porte extends Lieu {

    private int direction;
    private Salle s1,s2;
    private boolean ouverte;;

    public Porte(){
direction = -1;
s1 = null;
s2 = null;
ouverte = false;
    }

    public Porte(int d1, Salle s1, Salle s2, boolean b){
direction = d1;
this.s1 = s1;
this.s2 = s2;
ouverte = b;
s1.installeCote(d1,this);
s2.installeCote(Direction.opposee(d1),this);
    }

    public void poserPorte(int d1, Salle s1, Salle s2, boolean b){
direction = d1;
this.s1 = s1;
this.s2 = s2;
ouverte = b;
s1.installeCote(d1,this);
s2.installeCote(Direction.opposee(d1),this);
    }

    public boolean entrer(){
return ouverte;
    };

    public int acqDirection(Salle s){
if (s.equals(s1))
    return direction;
else if (s.equals(s2))
```

```
    return Direction.opposee(direction);
else
    return -1;
    }

    public Salle acqAutreCote(Salle s){
if (s.equals(s1))
    return s2;
else if (s.equals(s2))
    return s1;
else
    return null;
    }

}



public class Salle extends Lieu {

    private int numero;
    private Lieu[] cotes = new Lieu[4];

    public Salle(int num){
numero = num;
    }

    public boolean entrer(){
return true;
    };

    public void installeCote(int i, Lieu l){
cotes[i] = l;
    }

    public int acqNumero(){
return numero;
    }

    public Lieu acqCote(int i){
return cotes[i];
    }

    public int acqDirection(Lieu l){
for (int i= 0; i<4; i++)
    if (l.equals(cotes[i]))
return i;
return -1;
    }
```

```
}
```

La fabrique abstraite

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

Exemples de code

Le 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

La fabrication

Intention La 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.

Exemples de code

Le prototype

Intention Le 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.

Exemples de code

Le singleton

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

Exemples de code

6.2.5 Les modèles structuraux

6.2.6 Les modèles comportementaux

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.

Exemples de code

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.

Adapdateur convertit l'interface d'une classe en une autre conforme à l'attente du client. L'adapdateur 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 indépendammenet 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-système. La façade fournit une interface de plus haut niveau, qui rend le sous-système 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.

Interprèteur 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'intéraction 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.

Memento 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 défintit 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 at automatiquement 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.

Etat 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 algorithem). 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 réprésentation.

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-système 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 createurs.

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.
- 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 déplacés sans que les clients soient avertis.

7.2 Les modèles d'objets

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

7.2.1 Le modèle d'observation

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

Remarque : ce modèle existe en Java à travers l'interface *ImageObserver*.

7.2.2 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.

7.2.3 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

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).

figure des classes page 41.

8.2.1 Se connecter à la base de données

exemple page 47

8.2.2 Les intéractions simples

Les classes Statement et ResultSet.

exemple page 50

8.3 Optimiser l'accès aux BD

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

exemple page 57

8.3.2 Les procédures stockées

exemple page 59

8.4 Les accès dynamiques

Les metaClasses ResultSetMetaData et DatabaseMetaData permettent l'accès aux informations colonnes, tables, propriétaires, droits. . . .
exemple page 63