Université de Mons
Faculté des Sciences

US-MC-INFO60-016-C — Modélisation Logicielle

US-MC-INFO60-016-C — Dévelopement dirigé par les modèles

Synthèse :
Modélisation Logicielle
Dévelopement dirigé par les modèles

Auteur:
Baptiste Grosjean
Professeurs:
Tom Mens
Stéphane Duppont
Assistants:
LalaLala
Lala Lala

Numéro d'étudiant:
232732

Résumé

Synthèse des cours:

- "Modélisation Logicielle"dispensé par Tom Mens.
- "Dévelopement dirigé par les modèles"dispensé par Stéphane Dupont.

Janvier 2023

Sommaire

Chapter 1

UML

1.1 Modélisation

Définition 1 (Modèle)**.** *Un modèle est une représentation simplifiée d'une partie de la réalité dans un but spécifique.*

Un modèle est toujours une abstraction: pour réduire la complexité, on ne tient pas compte de certaines caractéristiques ou propriétés de l'objet complexe qu'on modélise, afin de se concentrer sur d'autres.

- Exactitude: dévelopez des modèles syntaxiquement corrects
- Précision: évitez l'ambiguité dans les modèles
- Concision: évitez les détails inutiles
- Complétude: modélisez tous les aspects essentiels
- Cohérence: évitez les incohérences dans les modèles
- Compréhensibilité: créez des modèles lisibles et compréhensibles
- Uniformité: utilisez un style uniforme partout

Définition 2 (UML — Unified Modeling Language)**.** *UML (Unified Modeling Language) est un ensemble de notations pour décrire un système logiciel de manière visuelle en termes de modèles (Langage de modélisation).*

Table 1.1: Langage UML

Points Forts	Langage "semi-formel"et standardisé	Gain de précision
		Gage de stabilité
		Encourage l'automatisation via l'utilisation d'outils dédiés
	Support de communication efficace	Cadre l'analyse des besoins
		Facilite la compréhension de systèmes abstrait et complexes
		Langage universel car polyvalent et souple
Points Faibles	Nécessité d'apprentissage et de période d'adaptation	UML contient beaucoup (trop?) de diagrammes ...
		On peut commencer avec un sous-ensemble
	Absence du processus, clé de la réussite d'un projet	Intégration d'UML dans un processus $\Longrightarrow Difficile$
		Améliorer un processus est une tâche longue et complexe

Table 1.2: Catégories et sous-catégories de diagrammes UML

Modélisation		Type de diagramme	Description
Structure Statique	Objet	Diagramme de Classe	Décrit les classes, leurs interrelations, et leurs instances.
		Diagramme d'objets	Décrit les objets et leurs relations avec les classes.
		Diagramme de paquetages	Montre la répartition des éléments dans des groupes logiques.
		Composite structure diagram	Représente la structure interne de composants de logiciel.
	Architecture	Diagramme de composants (logiciel)	Montre comment les différents composants du logiciel sont organisés logiquement.
		Diagramme de déploiement (matériel)	Montre comment les différents composants du matériel (hardware) sont organisés physiquement.
Comportement Dynamique	Utilisation	Diagramme de Cas d'Utilisation (Use Case Diagram)	Montre les utilisations possibles d'un logiciel.
	Activités et États	Diagramme d'États Comportementaux (Statecharts)	Montre comment le système se comporte de façon interne, modélise le comportement discret piloté par les évènements.
		Diagramme d'Activités	Modélise le comportement d'un système ou processus basé sur les flux de contrôle.
	Interaction	Diagramme de Séquence	Montre le comportement du système par l'interaction des objets qui le compose.
		Diagramme de communication	Montre comment les objets communiquent entre eux.
		Diagramme d'Interaction (Interaction Overview Diagram)	Montre comment les processus et interactions sont organisés dans le système.
		Timing diagram	Montre comment les interactions et les changements d'états sont liés dans le temps.

1.2 Diagramme de Cas d'Utilisation (Use Case Diagram)

Définition 3 (Diagramme de Cas d'Utilisation)**.** *Un diagramme de cas d'utilisation est un type de diagramme UML qui représente les interactions entre un système et ses utilisateurs dans le but de réaliser une fonctionnalité spécifique.*

Un diagramme de cas d'utilisation:

- *partitionne les besoins du système en différent cas d'utilisation*
- *permet d'identifier des besoins contradictoires et des imprécisions*
- *permet d'identifier les rôles joués par les utilisateurs du système; des utilisateurs différents ont des besoins différents*

Un cas d'utilisation est une description des interactions entre un système et ses utilisateurs dans le but de réaliser une tâche spécifique. Il permet de décrire les besoins du système de manière structurée, de déterminer les rôles des utilisateurs et de comprendre les différents cas d'utilisation du système.

Un diagramme de cas d'utilisation est un outil de modélisation qui représente les interactions entre un système et ses utilisateurs pour atteindre un objectif spécifique. Il est utilisé pour identifier et spécifier les besoins du système en les partitionnant en différents cas d'utilisation, en identifiant les rôles joués par les utilisateurs et en repérant les besoins contradictoires et les imprécisions.

Un diagramme de cas d'utilisation est un type de diagramme UML qui représente les interactions entre les utilisateurs (acteurs) et un système. Ces diagrammes permettent de décrire les scénarios d'utilisation d'un système, c'est-à-dire les actions et les réactions du système face aux demandes des utilisateurs. Ils permettent également de définir les limites du système et de comprendre les interactions entre les différents acteurs et le système. Les diagrammes de cas d'utilisation sont souvent utilisés pour la modélisation de l'utilisation d'un système et peuvent être utilisés pour développer des spécifications fonctionnelles.

Définition 4 (Cas d'utilisation)**.** *Un cas d'utilisation est une description des interactions entre un système et ses utilisateurs dans le but de réaliser une tâche spécifique. Il permet de décrire les besoins du système de manière structurée, de déterminer les rôles des utilisateurs et de comprendre les différents cas d'utilisation du système.*

Définition 5 (Cas d'utilisation d'extension)**.** *Un cas d'utilisation d'extension est une spécification qui décrit un cas d'utilisation qui peut étendre ou modifier le comportement d'un cas d'utilisation principal en cas de conditions exceptionnelles ou conditionnelles. Ces cas d'utilisation d'extension permettent de maintenir la simplicité de la description du cas d'utilisation principal en séparant les exceptions spécifiques.*

Un diagramme de cas d'utilisation est un type de diagramme UML qui représente les interactions entre les acteurs et le système en vue de réaliser une tâche spécifique. Il est composé de symboles tels que :

- Les acteurs : représentés par des personnes ou des entités extérieures au système
- Les cas d'utilisation : représentés par des ellipses et décrivant une fonctionnalité du système
- Les liens : représentés par des flèches reliant les acteurs aux cas d'utilisation et indiquant la relation de déclenchement ou d'interaction
- Les conditions d'extension : représentées par des traits en pointillé reliant un cas d'utilisation principal à un cas d'utilisation d'extension et indiquant une fonctionnalité supplémentaire qui peut être réalisée dans certains cas spéciaux.

1.3 Diagramme d'Activités

Définition 6 (Diagramme d'Activités)**.** *Un diagramme d'activité est un type de diagramme UML qui représente le flux de contrôle d'un processus ou d'un système en utilisant des symboles tels que des activités, des décisions et des transitions. Il permet de visualiser et de modéliser les différentes étapes d'un processus, les rôles des différents acteurs et les interactions entre ces différents éléments.*

Un diagramme d'activité est un type de diagramme UML qui représente le comportement dynamique d'un système en décrivant la succession des activités nécessaires pour réaliser une certaine tâche, en mettant l'accent sur le flux de contrôle. Les diagrammes d'activités sont particulièrement utiles pour modéliser la fonctionnalité d'un système et pour présenter une vue dynamique de celui-ci.

Un diagramme d'activité est un type de diagramme UML qui représente le comportement d'un système ou processus en mettant en évidence les flux de contrôle et les actions qui sont réalisées. Il utilise des symboles tels que des boîtes pour représenter les actions et des flèches pour représenter le flux de contrôle entre ces actions. Les diagrammes d'activité sont utilisés pour modéliser les processus métier et les algorithmes, ainsi que pour analyser et optimiser les performances des systèmes. Ils peuvent également être utilisés pour décrire les interactions entre les différents éléments d'un système et pour déterminer comment ces éléments coopèrent pour atteindre un objectif commun. Enfin, les diagrammes d'activité sont souvent utilisés pour documenter les processus et les protocoles de travail, afin de faciliter la communication et la collaboration au sein d'une équipe.

Voici la syntaxe générale d'un diagramme d'activités :

- L'acteur est représenté par un personnage avec une tête en forme de boîte.
- Les activités sont représentées par des rectangles avec des bords arrondis. Elles peuvent être numérotées pour indiquer leur ordre d'exécution.
- Les transitions entre les activités sont représentées par des flèches. Elles peuvent être annotées avec des conditions ou des événements qui déclenchent la transition.
- Les décisions sont représentées par des losanges. Elles sont annotées avec une condition qui détermine la branche à suivre.
- Les jonctions sont représentées par des cercles. Elles permettent de regrouper plusieurs transitions en une seule.
- Les objets du système sont représentés par des boîtes avec un titre. Ils peuvent être utilisés pour indiquer les données manipulées par les activités.

Il existe également d'autres éléments qui peuvent être utilisés dans un diagramme d'activités, tels que les sous-activités, les points de synchronisation, les parallélismes, etc.

1.4 Diagramme d'Interaction (Interaction Overview Diagram)

Définition 7 (Diagramme d'interaction)**.** *Un diagramme d'interaction est un type de diagramme UML qui représente les interactions entre les objets d'un système, en mettant en évidence leur comportement et les messages échangés.*

Un diagramme d'interaction est un type de diagramme UML utilisé pour représenter et modéliser les interactions et les échanges de messages entre les différents éléments d'un système. Ces éléments peuvent être des objets, des classes, des composants, etc. Le diagramme d'interaction permet de visualiser la séquence des évènements qui se déroulent au cours d'une interaction entre ces éléments, ainsi que les conditions qui déclenchent ces évènements. Il permet également de décrire les rôles joués par chaque élément dans l'interaction et les obligations qu'il doit remplir. Le diagramme d'interaction est particulièrement utile pour comprendre et documenter le comportement dynamique d'un système.

La syntaxe générale d'un diagramme d'interaction comprend les éléments suivants:

- Acteurs: représentés par des boîtes avec des têtes en forme de personnages, ils représentent les différents rôles joués par les utilisateurs du système.
- Objets: représentés par des boîtes avec des titres, ils représentent les différents objets du système qui interagissent entre eux.
- Messages: représentés par des flèches, ils indiquent l'envoi et la réception de messages entre les objets et les acteurs.
- Répliques: représentées par des flèches en pointillés, elles indiquent les réponses des objets aux messages reçus.

1.5 Diagramme de Classe

Définition 8 (Diagramme de Classe)**.** *Un diagramme de classe est un type de diagramme UML qui représente les classes d'un système, leurs attributs et leurs méthodes, ainsi que leurs relations avec d'autres classes.*

Un diagramme de classe est un type de diagramme UML qui représente les classes d'un système ainsi que leurs relations et leurs attributs. Il permet de visualiser la structure statique d'un système en modélisant ses différents objets et leurs interactions.

Un diagramme de classe est un type de diagramme UML qui représente les classes d'un système et leurs relations. Il permet de modéliser la structure statique du système en décrivant les classes, leurs attributs et leurs méthodes. Un diagramme de classe peut également illustrer les relations entre les classes, comme l'héritage, l'association ou l'agrégation. Un diagramme de classe est utile pour comprendre comment le système est organisé et comment les différentes parties interagissent entre elles. Il peut également être utilisé pour développer et maintenir le code source d'un projet logiciel.

La syntaxe générale d'un diagramme de classes comprend les éléments suivants :

- Classe: représentée par une boîte avec un titre, elle décrit les propriétés et les comportements d'un type d'objet.
- Attribut: représenté par un rectangle à l'intérieur de la classe, il décrit une propriété de l'objet.
- Méthode: représentée par un rectangle avec un triangle à l'intérieur, elle décrit un comportement de l'objet.
- Association: représentée par une flèche entre deux classes, elle indique que les objets de ces classes peuvent interagir entre eux.
- Aggrégation: représentée par une flèche avec un diamant à la pointe, elle indique que l'objet d'une classe contient des objets d'une autre classe.
- Composition: représentée par une flèche avec un cercle à la pointe, elle indique que l'objet d'une classe est dépendant des objets d'une autre classe.
- Héritage: représenté par une flèche avec une croix à la pointe, il indique que la classe enfant hérite des propriétés et comportements de la classe parente.
- Interfaces: représentées par des boîtes avec un titre souligné, elles décrivent les comportements attendus des objets qui implémentent cette interface.

1.5.1 Composants

Table 1.3: Composants d'une classe en UML 2.5

Composant	**Description**
Nom de la classe	Le nom de la classe, qui doit être unique et décrire de manière concise la fonction de la classe.
Visibilité	La visibilité de la classe, qui peut être publique, protégée ou privée.
Modificateurs	Les modificateurs de la classe, tels que « abstract » ou « final ».
Attributs	Les attributs de la classe, qui sont des variables membres qui décrivent les caractéristiques de la classe.
Opérations	Les opérations de la classe, qui sont des méthodes membres qui décrivent les actions que la classe peut effectuer.
Associations	Les associations de la classe, qui décrivent les relations entre la classe et d'autres classes ou objets.
Dépendances	Les dépendances de la classe, qui décrivent les relations de dépendance entre la classe et d'autres classes ou objets.
Notes	Les notes de la classe, qui sont des informations supplémentaires sur la classe.

Attributs

Table 1.4: Types d'attributs et de leur syntaxe en UML 2.5 :

Type d'attribut	**Symbole**	**Syntaxe**	**Description**
Attribut public	$+$	$+nom : type$	L'attribut est accessible à l'extérieur de la classe.
Attribut protégé	$\#$	$\#nom : type$	L'attribut est accessible seulement par les classes héritant de la classe contenant l'attribut.
Attribut privé	$-$	$-nom : type$	L'attribut est accessible seulement à l'intérieur de la classe.
Attribut de package	$\tilde{}$	$\tilde{}nom : type$	L'attribut est accessible par toutes les classes dans le même package.

Opérations

Nom	Type de retour	Arguments
returnPos	returnPos() : Position	
draw		
scaleFigure		scaleFigure(percent : int)

Associations

Cardinalité	Alternative	Signification
1	1..1	Un et un seul
0..1		Zéro ou un
m..n		De m à n (entiers naturels)
*	0..*	De zéro à plusieurs
1..*		D'un à plusieurs

1.5.2 Types de classes

Voici un tableau comparatif des différents types de classes et leurs syntaxes en UML 2.5 :

Type de classe	**Syntaxe**	**Description**
Classe normale	`class NomClasse`	Classe standard, représentée par une boîte pleine
Classe abstraite	`abstract class NomClasse`	Classe qui ne peut pas être instanciée, représentée par une boîte vide. Peut contenir des méthodes concrètes et des méthodes abstraites.Peut contenir des attributs et des constantes. Peut être utilisée comme classe mère pour hériter de ses méthodes concrètes et de ses méthodes abstraites.
Interface	`interface NomInterface`	Classe qui ne contient que des méthodes abstraites, représentée par une boîte avec des lignes brisées. Ne peut contenir que des méthodes abstraites. Ne peut contenir que des constantes.Ne peut pas être utilisée comme classe mère, mais peut être implémentée par une classe pour hériter de ses méthodes abstraites.
Classe enveloppe	`«envelope» NomClasse`	Classe qui encapsule un objet existant, représentée par une boîte avec une bordure en pointillés

1.6 Diagramme de Séquence

Définition 9 (Diagramme de séquence)**.** *Un diagramme de séquence est un type de diagramme UML qui représente l'interaction entre des objets ou des composants au cours du temps, en indiquant la séquence des messages échangés entre eux.*

Un diagramme de séquence est un type de diagramme UML qui représente les interactions entre différents objets au cours du temps. Il permet de visualiser les messages échangés entre ces objets, ainsi que leur ordre d'exécution. Un diagramme de séquence peut être utilisé pour modéliser une partie d'un système ou l'ensemble du système. Il est particulièrement utile pour comprendre comment les objets collaborent pour réaliser des tâches et pour identifier les éventuelles problèmes de synchronisation ou de dépendance. Un diagramme de séquence est souvent utilisé en conjonction avec d'autres types de diagrammes UML, comme les diagrammes de cas d'utilisation et les diagrammes de classes.

Un diagramme de séquence est un type de diagramme UML qui représente la séquence des messages échangés entre les objets dans le temps. Il est utilisé pour modéliser le comportement dynamique d'un système en décrivant les interactions entre objets à un niveau de détail plus fin que le diagramme de cas d'utilisation.

La syntaxe d'un diagramme de séquence comprend les éléments suivants :

Acteur : un acteur est un rôle joué par un utilisateur du système. Il est représenté par un personnage à la tête en losange. Objet : un objet est un élément du système qui possède des comportements et des données. Il est représenté par un rectangle. Message : un message est un échange de données ou de commandes entre deux objets. Il est représenté par une flèche reliant les deux objets. Lifeline : une lifeline est une ligne verticale qui représente la durée de vie d'un objet. Activation : une activation est un rectangle qui indique que l'objet est occupé à exécuter une action pendant un certain temps.

1.7 Diagramme d'États Comportementaux (Statecharts)

Définition 10 (Diagramme d'États Comportementaux)**.** *Un diagramme d'états comportementaux est un outil de modélisation qui représente les transitions d'un système entre différents états, ainsi que les actions et événements qui en découlent.*

Un diagramme d'états comportementaux (ou statechart en anglais) est un type de diagramme UML qui permet de modéliser le comportement discret d'un système ou d'un objet, en prenant en compte les évènements qui déclenchent des transitions entre différents états. Ces diagrammes permettent de visualiser comment un système ou un objet réagit à des évènements internes ou externes, ainsi que les actions qui sont effectuées à chaque transition. Ils peuvent également inclure des informations sur les conditions qui doivent être remplies pour que la transition ait lieu, ainsi que sur les actions qui sont exécutées pendant l'état courant. En résumé, les diagrammes d'états comportementaux sont utiles pour comprendre comment un système ou un objet fonctionne de manière interne, et pour identifier les scénarios d'utilisation possibles.

Table 1.5: Actions/activités internes d'un état

Type	**d'action**	**Déclenchement**	**Mot clé**
Action de transition		Lorsqu'une transition est suivie	
Actions/activités internes d'un état	Action d'entrée	Se produisent lorsque le système se trouve dans un certain état Lorsque le système entre dans un certain état	entry
	Action de sortie	Lorsque le système sort d'un certain état	exit
	Action temporelle	Se déclenche basée sur un événement temporel	every x s
	Activité d'état	Se déroule tout le temps où l'on est dans l'état	do
Autres événements		Peuvent également déclencher des actions internes	

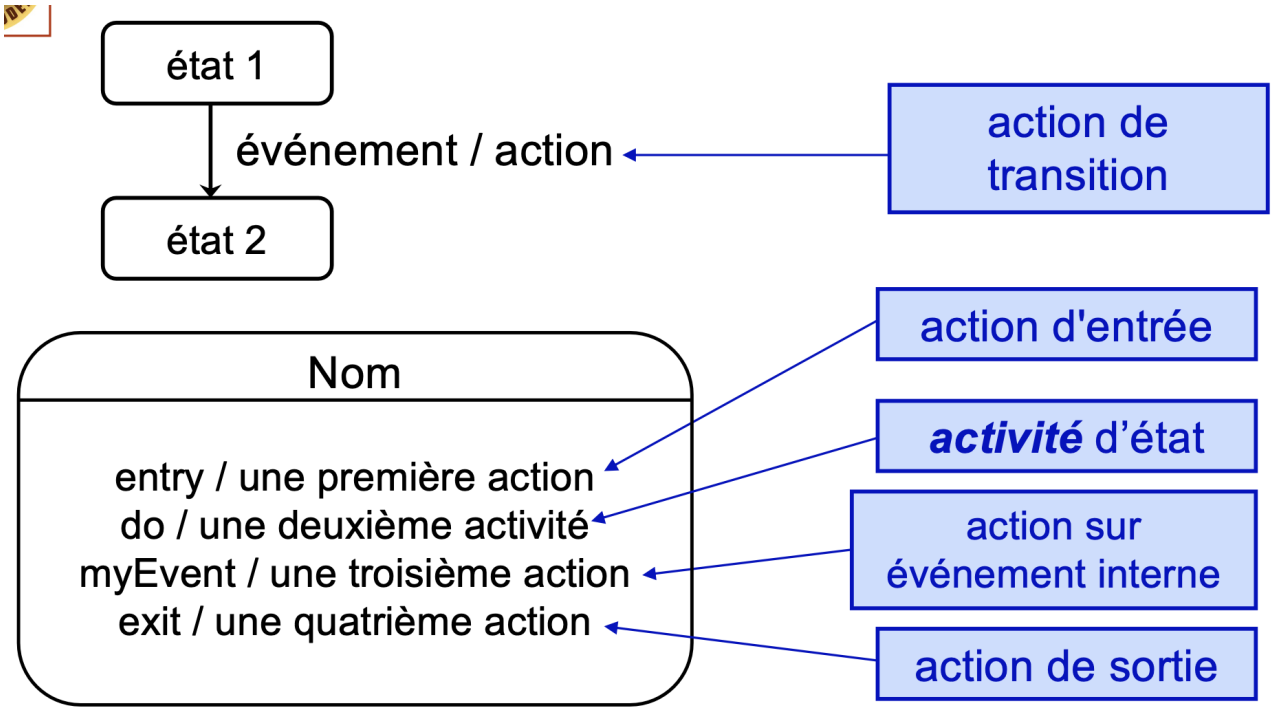


Figure 1.1: Les actions associées à entry et exit ne sont pas interruptibles. Les activités associées à do sont interruptibles.

Table 1.6: UML fait la distinction entre une activité et une action

Différences principales	Une activité est composée d'actions ordonnées	
	Une action est une unité d'exécution automatique et non interruptible	Elle doit se terminer avant qu'une autre action puisse être considérée Elle ne peut pas être interrompue par une transition.
Une action associée à entry ou exit n'a pas de durée	Elle n'est pas interruptible	
Une activité associée à do prend un temps non négligeable	Elle est exécutée pendant que l'objet est dans l'état donné	
	Elle peut être interrompue à tout moment, dès qu'une transition de sortie d'état est déclenchée	

Chapter 2

Développement dirigé par les modèles

2.1 Design Patterns

Définition 11 (Design Pattern)**.**

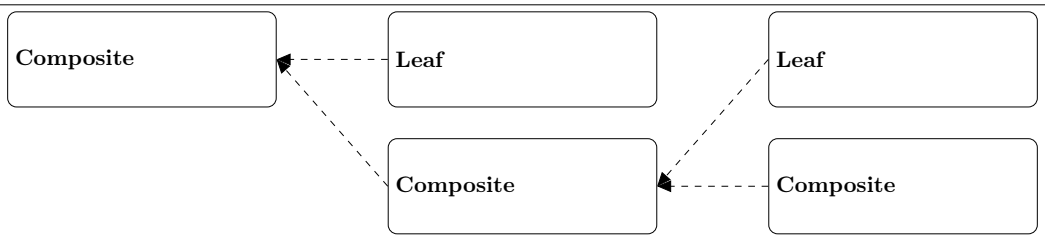
Table 2.1: Principaux design patterns d'objets par catégorie

Catégorie	**Design patterns**	**Description**
Création	Fabrique abstraite	Permet de déléguer la création d'objets à des sous-classes.
	Prototype	Permet de créer de nouveaux objets en copiant des objets existants.
	Singleton	Assure qu'une classe ne possède qu'une seule instance et fournit un accès global à celle-ci.
	Builder	Sépare la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction puisse créer différentes représentations.
	Facteur de construction	Abstrait la création d'objets en une étape de construction séparée.
Structure	Adaptateur	adapte l'interface d'une classe à une autre interface attendue par les clients.
	Bridge	Sépare l'implémentation d'une classe de son interface, de sorte que les deux puissent être modifiées indépendamment.
	Composite	Compose des objets en structures arborescentes pour représenter l'héritage partiel et laisse le client traiter de manière uniforme des objets simples et composites.
	Decorator	Ajoute de nouvelles responsabilités à un objet de manière transparente.
	Façade	Fournit une interface unique pour une sous-système complexe afin de le rendre plus facile à utiliser.
	Flyweight	Utilise des objets partagés pour soutenir de nombreuses instances avec un état faible ou non-existent.
	Poids mouche	Est un patron de conception qui vise à minimiser l'utilisation de mémoire en partageant des objets similaires au lieu de les créer indépendamment.
Comportement	Méthode de template	Définit un schéma de traitement pour une opération en déléguant certaines étapes à des sous-classes.
	Méthode de chaîne de responsabilité	Permet à un objet de passer une requête le long d'une chaîne de traitement jusqu'à ce qu'un objet la traite.
	Méthode de commande	Encapsule une requête sous la forme d'un objet, ce qui permet de paramétrer des objets avec différentes requêtes, de les mettre dans une file d'attente, et de supporter l'annulation et le retour en arrière.
	Méthode d'observer	Définit une relation un-à-plusieurs entre des objets de sorte que lorsqu'un objet change d'état, tous ses dépendants en sont notifiés et mis à jour automatiquement.
	Méthode de récepteur	Encapsule la logique de traitement d'une requête dans un objet, de sorte qu'une requête peut être passée à différents objets de manière interchangeable.
	Méthode de visiteur	Représente une opération à effectuer sur les éléments d'une structure de données hiérarchique.

2.1.1 Composite

Définition 12 (Design Pattern Composite)**.**

Table 2.2: Design pattern Composite

But	Permettre la manipulation d'une hiérarchie de objets de manière uniforme, que ces objets soient des composants simples ou des groupes de composants.
Quand	Lorsque vous avez une hiérarchie d'objets et que vous voulez traiter chaque objet de manière uniforme, qu'il s'agisse d'un composant simple ou d'un groupe de composants.
Comment	Le design pattern Composite définit une interface pour les composants de la hiérarchie, qui peut être implémentée par des classes de composants simples ou de groupes de composants. Cela permet de traiter chaque objet de la hiérarchie de manière uniforme, indépendamment de son type.
Avantages	Permet de traiter les composants simples et les groupes de composants de manière uniforme. Rend le code plus lisible et facilite l'ajout de nouvelles fonctionnalités.
Inconvénients	Nécessite la création d'une interface pour les composants de la hiérarchie. Peut rendre le code plus complexe et difficile à comprendre si utilisé de manière excessive.
Exemples	Modélisation d'une structure hiérarchique : Le design pattern Composite peut être utilisé pour modéliser une structure hiérarchique de données, comme par exemple l'arborescence d'un système de fichiers. Dans ce cas, chaque dossier peut être considéré comme un "composite", qui peut contenir à la fois des fichiers et d'autres dossiers (qui sont eux aussi des composites). Le composite "racine" de l'arborescence est la racine du système de fichiers.
Structure	

```java
public abstract class FileSystemNode {
  protected String name;

  public FileSystemNode(String name) {
    this.name = name;
  }

  public abstract int getSize();
}

public class File extends FileSystemNode {
  private int size;

  public File(String name, int size) {
    super(name);
    this.size = size;
  }

  @Override
  public int getSize() {
    return size;
  }
}

public class Directory extends FileSystemNode {
  private List<FileSystemNode> children;

  public Directory(String name) {
    super(name);
    children = new ArrayList<>();
  }

  public void addNode(FileSystemNode node) {
    children.add(node);
  }

  @Override
  public int getSize() {
    int size = 0;
    for (FileSystemNode child : children) {
      size += child.getSize();
    }
    return size;
  }
}

// Exemple d'utilisation :
Directory root = new Directory("root");
Directory home = new Directory("home");
Directory user = new Directory("user");

File file1 = new File("file1.txt", 100);
File file2 = new File("file2.txt", 200);
File file3 = new File("file3.txt", 300);

user.addNode(file1);
user.addNode(file2);
home.addNode(user);
home.addNode(file3);
root.addNode(home);

System.out.println("Taille totale du repertoire racine : "
    + root.getSize() + " octets");
```

Arbre de décision en intelligence artificielle : Le design pattern Composite peut être utilisé pour modéliser un arbre de décision en intelligence artificielle. Dans ce cas, chaque nœud de l'arbre peut être considéré comme un "composite", qui peut contenir à la fois des feuilles (qui sont des éléments "feuilles" de l'arbre) et d'autres nœuds (qui sont eux aussi des composites). Le composite "racine" de l'arbre est le nœud racine de l'arbre de décision.

```java
// Classe abstraite pour representer un noud de l'arbre de decision
public abstract class DecisionTreeNode {
  // Methode abstraite pour valuer le noud de l'arbre de decision
  public abstract boolean evaluate();
}

// Classe concrete pour representer un noud "composite" de l'arbre de decision
public class CompositeDecisionTreeNode extends DecisionTreeNode {
  private List<DecisionTreeNode> children; // Liste des enfants du noud

  // Constructeur pour initialiser la liste des enfants
  public CompositeDecisionTreeNode(List<DecisionTreeNode> children) {
    this.children = children;
  }

  // Methode d'valuation de l'arbre de decision qui parcourt tous les enfants et renvoie true si tous les enfants renvoient true
  @Override
  public boolean evaluate() {
    for (DecisionTreeNode child : children) {
      if (!child.evaluate()) {
        return false;
      }
    }
    return true;
  }
}

// Classe concrete pour representer une feuille de l'arbre de decision
public class LeafDecisionTreeNode extends DecisionTreeNode {
  private boolean value; // Valeur de la feuille

  // Constructeur pour initialiser la valeur de la feuille
  public LeafDecisionTreeNode(boolean value) {
    this.value = value;
  }

  // Methode d'valuation de l'arbre de decision qui renvoie la valeur de la feuille
  @Override
  public boolean evaluate() {
    return value;
  }
}

// Exemple d'utilisation de l'arbre de decision
DecisionTreeNode root = new CompositeDecisionTreeNode(Arrays.asList(
  new LeafDecisionTreeNode(true),
  new LeafDecisionTreeNode(true),
  new LeafDecisionTreeNode(false)
));

boolean result = root.evaluate(); // false
```

2.1.2 Visitor

Définition 13 (Design Pattern Visitor)**.**

Table 2.3: Design pattern Visitor

But	Permettre l'ajout de fonctionnalités à une hiérarchie de classes sans en changer la structure.
Quand	Lorsque vous avez une hiérarchie de classes et que vous voulez ajouter des fonctionnalités à chaque classe de cette hiérarchie sans en changer la structure.
Comment	Le design pattern Visitor définit une nouvelle opération à chaque classe de la hiérarchie, qui accepte un visiteur comme argument. Le visiteur contient les fonctionnalités à ajouter. Lorsqu'une classe de la hiérarchie accepte un visiteur, elle appelle la méthode correspondante de ce visiteur.
Avantages	Permet d'ajouter de nouvelles fonctionnalités sans changer la structure de la hiérarchie de classes. Sépare les fonctionnalités ajoutées de la hiérarchie de classes, ce qui peut rendre le code plus lisible et faciliter l'extension.
Inconvénients	Nécessite la création d'une nouvelle classe pour chaque fonctionnalité à ajouter. Peut rendre le code plus complexe et difficile à comprendre si utilisé de manière excessive.
Exemples	Imaginons que vous avez une hiérarchie de classes représentant des formes géométriques (cercle, carré, triangle, etc.). Vous souhaitez ajouter la fonctionnalité de calcul de l'aire de chaque forme sans changer la structure de la hiérarchie de classes. Vous pouvez utiliser le design pattern Visitor pour créer une classe Visitor qui contient une méthode pour chaque type de forme. Lorsque la forme accepte le visiteur, elle appelle la méthode correspondante de ce visiteur pour calculer son aire.

```java
public interface Shape {
  void accept(Visitor visitor);
}

public class Circle implements Shape {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public double getRadius() {
    return radius;
  }

  @Override
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}

public class Square implements Shape {
  private double side;

  public Square(double side) {
    this.side = side;
  }

  public double getSide() {
    return side;
  }

  @Override
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}

public class Triangle implements Shape {
  private double base;
  private double height;

  public Triangle(double base, double height) {
    this.base = base;
    this.height = height;
  }

  public double getBase() {
    return base;
  }

  public double getHeight() {
    return height;
  }

  @Override
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}

public interface Visitor {
  void visit(Circle circle);
  void visit(Square square);
  void visit(Triangle triangle);
}

public class AreaVisitor implements Visitor {
  @Override
  public void visit(Circle circle) {
    double radius = circle.getRadius();
    double area = Math.PI * radius * radius;
    System.out.println("Area of circle: " + area);
  }

  @Override
  public void visit(Square square) {
    double side = square.getSide();
    double area = side * side;
    System.out.println("Area of square: " + area);
  }

  @Override
  public void visit(Triangle triangle) {
    double base = triangle.getBase();
    double height = triangle.getHeight();
    double area = 0.5 * base * height;
    System.out.println("Area of triangle: " + area);
  }
}

Circle circle = new Circle(5);
Square square = new Square(10);
Triangle triangle = new Triangle(5, 10);

Visitor visitor = new AreaVisitor();
circle.accept(visitor);
```

Imaginons que vous avez une hiérarchie de classes représentant des employés dans une entreprise (directeur, manager, employé). Vous souhaitez ajouter la fonctionnalité de calcul de la rémunération de chaque employé sans changer la structure de la hiérarchie de classes. Vous pouvez utiliser le design pattern Visitor pour créer une classe Visitor qui contient une méthode pour chaque type d'employé. Lorsque l'employé accepte le visiteur, il appelle la méthode correspondante de ce visiteur pour calculer sa rémunération.

```java
// Classe abstraite repr sentant un employ
abstract class Employee {
  // Attributs de l'employ  (nom, salaire, etc.)
  protected String name;
  protected double salary;

  // Constructeur prenant en param tre le nom et le
      salaire de l'employ
  public Employee(String name, double salary) {
    this.name = name;
    this.salary = salary;
  }

  // M thode abstraite acceptant un visiteur
  public abstract void accept(Visitor visitor);
}

// Classe repr sentant un directeur
class Director extends Employee {
  public Director(String name, double salary) {
    super(name, salary);
  }

  // Acceptation du visiteur
  @Override
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}

// Classe repr sentant un manager
class Manager extends Employee {
  public Manager(String name, double salary) {
    super(name, salary);
  }

  // Acceptation du visiteur
  @Override
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}

// Classe repr sentant un employ
class Employee extends Employee {
  public Employee(String name, double salary) {
    super(name, salary);
  }

  // Acceptation du visiteur
  @Override
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}

// Interface repr sentant un visiteur
interface Visitor {
  // M thode de visite pour chaque type d'employ
  void visit(Director director);
  void visit(Manager manager);
  void visit(Employee employee);
}

// Classe repr sentant un visiteur calculant la
    r mun ration des employ s
class SalaryCalculatorVisitor implements Visitor {
  // Attributs stockant la r mun ration totale pour
      chaque type d'employ
  private double totalDirectorSalary;
  private double totalManagerSalary;
  private double totalEmployeeSalary;

  // M thodes de visite pour chaque type d'employ
  @Override
  public void visit(Director director) {
    totalDirectorSalary += director.salary;
  }

  @Override
  public void visit(Manager manager) {
    totalManagerSalary += manager.salary;
  }

  @Override
  public void visit(Employee employee) {
    totalEmployeeSalary += employee.salary;
  }
}
```

2.1.3 Observer

Table 2.4: Design pattern Observer

But	Permettre à un objet de suivre l'état d'un autre objet et de recevoir une notification en cas de changement d'état.
Quand	Lorsque vous voulez que plusieurs objets restent synchronisés et reçoivent une notification en cas de changement d'état de l'un d'entre eux.
Comment	Le design pattern Observer définit une relation de type "un-vers-plusieurs" entre un objet observé (Observable) et plusieurs objets observateurs (Observer). L'Observable envoie une notification aux Observer en cas de changement d'état. Les Observer peuvent alors mettre à jour leur état en fonction de celui de l'Observable.
Avantages	Permet de maintenir la synchronisation entre plusieurs objets sans avoir à connaître leurs implémentations. Facilite l'ajout ou la suppression d'Observer sans avoir à modifier l'Observable.
Inconvénients	Nécessite la création de liens de dépendance entre les objets observés et les objets observateurs. Peut rendre le code plus complexe si utilisé de manière excessive.
Exemples	Mise à jour en temps réel d'un tableau de bord d'un système de gestion de projets. Lorsque des données sont mises à jour dans le système, un observateur est déclenché et met à jour le tableau de bord en conséquence. Notification d'un utilisateur lorsqu'un nouveau message est reçu dans une application de messagerie. L'application est configurée pour observer les nouveaux messages et en informer l'utilisateur en envoyant une notification push.
Structure	

```java
import java.util.ArrayList;
import java.util.List;

// Classe Observateur
interface Observateur {
  public void update(int nouvelleDonnee);
}

// Classe Sujet
class Sujet {
  private List<Observateur> observateurs = new ArrayList<
      Observateur>();
  private int donnee;

  public void ajouterObservateur(Observateur observateur) {
    observateurs.add(observateur);
  }

  public void supprimerObservateur(Observateur observateur)
      {
    observateurs.remove(observateur);
  }

  public void notifierObservateurs() {
    for (Observateur observateur : observateurs) {
      observateur.update(donnee);
    }
  }

  public void mettreAJourDonnee(int nouvelleDonnee) {
    donnee = nouvelleDonnee;
    notifierObservateurs();
  }
}

// Classe Observateur concr te
class TableauDeBord implements Observateur {
  private Sujet sujet;

  public TableauDeBord(Sujet sujet) {
    this.sujet = sujet;
    sujet.ajouterObservateur(this);
  }

  @Override
  public void update(int nouvelleDonnee) {
    // Mise   jour du tableau de bord avec la nouvelle
        donn e
  }
}

// Classe de test
class Test {
  public static void main(String[] args) {
    Sujet sujet = new Sujet();
    TableauDeBord tableauDeBord = new TableauDeBord(sujet);

    sujet.mettreAJourDonnee(10); // Le tableau de bord sera
        mis    jour avec la nouvelle donn e
  }
}
```

```java
import java.util.ArrayList;
import java.util.List;

// Classe repr sentant un utilisateur de l'application de
    messagerie
class User {
  private String name;
  private List<Message> messages;

  public User(String name) {
    this.name = name;
    this.messages = new ArrayList<>();
  }

  // M thode appel e lorsqu'un nouveau message est re u
  public void receiveMessage(Message message) {
    messages.add(message);
    // Envoi de la notification push    l'utilisateur
    sendPushNotification();
  }

  // M thode permettant d'envoyer une notification push
      l'utilisateur
  public void sendPushNotification() {
    System.out.println("Notification envoy e   l'
        utilisateur " + name + " : nouveau message re u !
        ");
  }
}

// Classe repr sentant un message dans l'application de
    messagerie
class Message {
  private String content;

  public Message(String content) {
    this.content = content;
  }

  public String getContent() {
    return content;
  }
}

// Classe repr sentant l'application de messagerie
class MessagingApp {
  private List<User> users;
  private List<Message> messages;

  public MessagingApp() {
    this.users = new ArrayList<>();
    this.messages = new ArrayList<>();
  }

  // M thode permettant d'ajouter un utilisateur    l'
      application
  public void addUser(User user) {
    users.add(user);
  }

  // M thode permettant d'envoyer un message    tous les
      utilisateurs de l'application
  public void sendMessage(Message message) {
    messages.add(message);
    // Notification de tous les utilisateurs de l'
        application
    for (User user : users) {
      user.receiveMessage(message);
    }
  }
}

public class Main {
  public static void main(String[] args) {
    // Cr ation de l'application de messagerie
    MessagingApp messagingApp = new MessagingApp();

    // Cr ation de deux utilisateurs
    User user1 = new User("Alice");
    User user2 = new User("Bob");

    // Ajout des utilisateurs    l'application de
        messagerie
    messagingApp.addUser(user1);
    messagingApp.addUser(user2);

    // Envoi d'un message    tous les utilisateurs de l'
        application
    messagingApp.sendMessage(new Message("Bonjour, comment
        vas-tu ?"));
  }
}
```

2.1.4 Singleton

Table 2.5: Design pattern Singleton

But	Garantir qu'une classe ne peut être instanciée qu'une seule fois, en maintenant un point d'accès global à cette instance.
Quand	Lorsque vous voulez que votre application n'ait qu'une seule instance d'une classe particulière, par exemple pour gérer l'accès à une base de données ou à une ressource partagée.
Comment	Le design pattern Singleton définit une méthode d'accès global à une instance unique de la classe. Cette instance est créée au moment de la première utilisation de la méthode d'accès. Les autres appels à cette méthode retourneront simplement la référence à l'instance existante.
Avantages	Permet de garantir qu'il n'existe qu'une seule instance d'une classe donnée dans l'application. Facilite le partage de ressources et la gestion de l'accès à celles-ci.
Inconvénients	Peut rendre le code difficile à tester, car il n'est pas possible de créer plusieurs instances de la classe pour les tests. Peut également rendre le code plus difficile à maintenir si utilisé de manière excessive.
Exemples	Gestionnaire de connexion à une base de données : vous souhaitez que votre application n'ait qu'une seule connexion à la base de données afin d'éviter les conflits et d'optimiser les performances. Vous pouvez utiliser le design pattern singleton pour créer un objet unique qui gère la connexion à la base de données et qui est accessible depuis n'importe quelle partie de votre application.
Structure	

```java
public class GestionnaireConnexionBDD {
  private static GestionnaireConnexionBDD instance = null;
  private Connection connexion = null;

  private GestionnaireConnexionBDD() {
    // Initialisation de la connexion    la base de
        donn es
  }

  public static GestionnaireConnexionBDD getInstance() {
    if (instance == null) {
      instance = new GestionnaireConnexionBDD();
    }
    return instance;
  }

  public Connection getConnexion() {
    return connexion;
  }
}


GestionnaireConnexionBDD gestionnaire =
    GestionnaireConnexionBDD.getInstance();
Connection connexion = gestionnaire.getConnexion();
```

Gestionnaire de fichiers de configuration : vous avez un fichier de configuration qui est utilisé par plusieurs parties de votre application et vous souhaitez qu'il soit accessible de manière simple et rapide. Vous pouvez utiliser le design pattern singleton pour créer un objet unique qui gère l'accès au fichier de configuration et qui est accessible depuis n'importe quelle partie de votre application.

```java
import java.util.Properties;
import java.io.FileInputStream;
import java.io.IOException;

public class ConfigurationManager {
  private static ConfigurationManager instance;
  private Properties config;

  private ConfigurationManager() {
    config = new Properties();
    try {
      config.load(new FileInputStream("config.properties"))
          ;
    } catch (IOException e) {
      // G rer l'exception
    }
  }

  public static ConfigurationManager getInstance() {
    if (instance == null) {
      instance = new ConfigurationManager();
    }
    return instance;
  }

  public String getProperty(String key) {
    return config.getProperty(key);
  }
}

String property = ConfigurationManager.getInstance().
    getProperty("key");
```

2.1.5 Decorator

Table 2.6: Design pattern Decorator

But	Ajouter dynamiquement de nouvelles fonctionnalités à un objet sans altérer sa structure.
Quand	Lorsque vous voulez ajouter de nouvelles fonctionnalités à un objet de manière flexible, sans avoir à utiliser une héritage complexe ou à créer de nouvelles classes pour chaque combinaison de fonctionnalités.
Comment	Le design pattern Decorator définit une interface commune pour les objets à décorer et une classe de base pour les décorateurs qui enveloppent ces objets. Les décorateurs ajoutent des fonctionnalités supplémentaires en étendant la classe de base des décorateurs et en implémentant l'interface commune. Les objets peuvent alors être décorés en ajoutant successivement des décorateurs qui ajoutent les fonctionnalités souhaitées.
Avantages	Permet d'ajouter de nouvelles fonctionnalités à un objet de manière flexible et sans altérer sa structure. Facilite la modification du comportement d'un objet à runtime.
Inconvénients	Peut rendre le code plus complexe et difficile à comprendre, en particulier si de nombreux décorateurs sont utilisés.
Exemples	Un système de commande de restaurant en ligne qui permet aux clients de personnaliser leurs commandes en ajoutant des ingrédients supplémentaires. Le design pattern Decorator peut être utilisé pour ajouter des fonctionnalités à l'objet "commande" de manière dynamique, en créant des classes décoratrices pour chaque ingrédient supplémentaire (par exemple, "oeuf", "bacon", "oignon"). Ces classes décoratrices hériteront de la classe de base "commande" et ajouteront leur propre comportement (par exemple, augmenter le prix de la commande) lorsque la commande est créée.
Structure	

```java
// Classe de base "Commande"
public abstract class Commande {
protected double prix;

public abstract double getPrix();
}

// Classe d coratrice "Ingredient"
public abstract class Ingredient extends Commande {
protected Commande commande;

public Ingredient(Commande commande) {
this.commande = commande;
}
}

// Classe d coratrice concr te "Oeuf"
public class Oeuf extends Ingredient {
public Oeuf(Commande commande) {
super(commande);
}

@Override
public double getPrix() {
return commande.getPrix() + 1.50;
}
}

// Classe d coratrice concr te "Bacon"
public class Bacon extends Ingredient {
public Bacon(Commande commande) {
super(commande);
}

@Override
public double getPrix() {
return commande.getPrix() + 2.00;
}
}

// Classe d coratrice concr te "Oignon"
public class Oignon extends Ingredient {
public Oignon(Commande commande) {
super(commande);
}

@Override
public double getPrix() {
return commande.getPrix() + 0.75;
}
}

// Exemple d'utilisation
Commande commandeDeBase = new Commande();
Commande commandePersonnalisee = new Oeuf(new Bacon(new
    Oignon(commandeDeBase)));
System.out.println("Prix de la commande : " +
    commandePersonnalisee.getPrix() + " ");
```

Un logiciel de retouche photo qui permet aux utilisateurs de ajouter des filtres et des effets à leurs photos. Le design pattern Decorator peut être utilisé pour ajouter des fonctionnalités à l'objet "photo" de manière dynamique, en créant des classes décoratrices pour chaque effet (par exemple, "filtre sépia", "flou", "couleur inversée"). Ces classes décoratrices hériteront de la classe de base "photo" et ajouteront leur propre comportement (par exemple, appliquer le filtre ou l'effet à l'image) lorsque la photo est modifiée.

```java
// Classe de base "Photo"
public abstract class Photo {
protected String description;

public String getDescription() {
return description;
}

public abstract void modify();
}

// Classe d coratrice "FiltreSepia"
public class FiltreSepia extends Photo {
public FiltreSepia(Photo photo) {
description = "Filtre s pia";
}

public void modify() {
// Appliquer le filtre s pia    l'image
}
}

// Classe d coratrice "Flou"
public class Flou extends Photo {
public Flou(Photo photo) {
description = "Flou";
}

public void modify() {
// Appliquer le flou    l'image
}
}

// Classe d coratrice "CouleurInversee"
public class CouleurInversee extends Photo {
public CouleurInversee(Photo photo) {
description = "Couleur invers e";
}

public void modify() {
// Appliquer la couleur invers e    l'image
}
}

// Utilisation du design pattern Decorator
Photo photo = new FiltreSepia(new Flou(new CouleurInversee(
    new Photo())));
System.out.println(photo.getDescription()); // Affiche "
    Couleur invers e, Flou, Filtre s pia"
photo.modify(); // Applique tous les effets    l'image
```

2.2 Java

2.2.1 Variables

2.2.2 Méthodes

2.2.3 Classes

Chapter 3

Examens Corrigés

3.1 Examen 2022

Question 1

A

B

C

Question 2

Question 3

Question 4

Question 5

Question 6

3.2 Examen 2021

Question 1

Question 2

Question 3

Question 4

Question 5

Question 6

3.3 Examen 2020

Question 1

Question 2

Question 3

Question 4

Question 5

Question 6

3.4 Examen 2019

Question 1

Question 2

Question 3

Question 4

Question 5

Question 6

3.5 Examen 2018

Question 1

Question 2

Question 3

Question 4

Question 5

Question 6

3.6 Examen 2017

Question 1

Question 2

Question 3

Question 4

Question 5

Question 6

Figures

Tableaux