

# TI615M

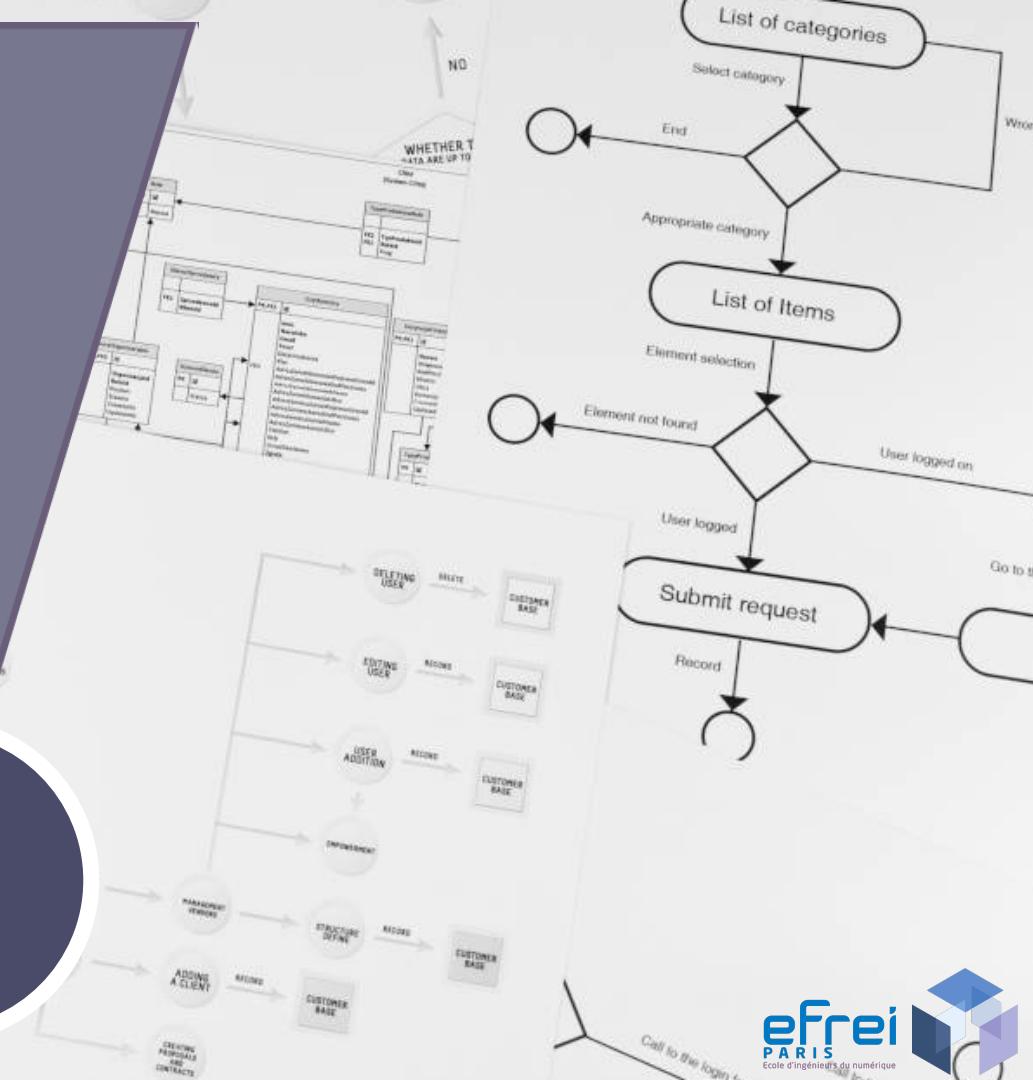
## Analyse et Conception Orientée Objet avec

# UML

Dr. Lilia SFAXI

## Introduction à la Conception Logicielle

1



# Plan

1

Introduction aux Systèmes d'Information

2

Étapes de conception

3

Langage de modélisation unifié  
(UML)

4

Le Paradigme Orienté Objet



# Plan

1

Introduction aux Systèmes d'Information

2

Étapes de conception

3

Langage de modélisation unifié  
(UML)

4

Le Paradigme Orienté Objet



# L'information

## Introduction aux Systèmes d'Information

- L'information est la base de la connaissance et de la communication humaines
- Dans une entreprise, l'information est, en même temps :
  - Un outil de communication interne
    - Permet la coordination entre les différents départements et acteurs de l'entreprise
  - Un outil de communication externe
    - Elle est diffusée à l'environnement extérieur
  - Un outil de cohésion sociale
    - La transparence et la diffusion de l'information renforcent le sentiment d'appartenance et la motivation collective
- L'information est différente des "données".
  - Les données deviennent des informations lorsqu'elles sont reçues par un être humain et interprétées
  - L'importance des données brutes varie d'une personne à l'autre



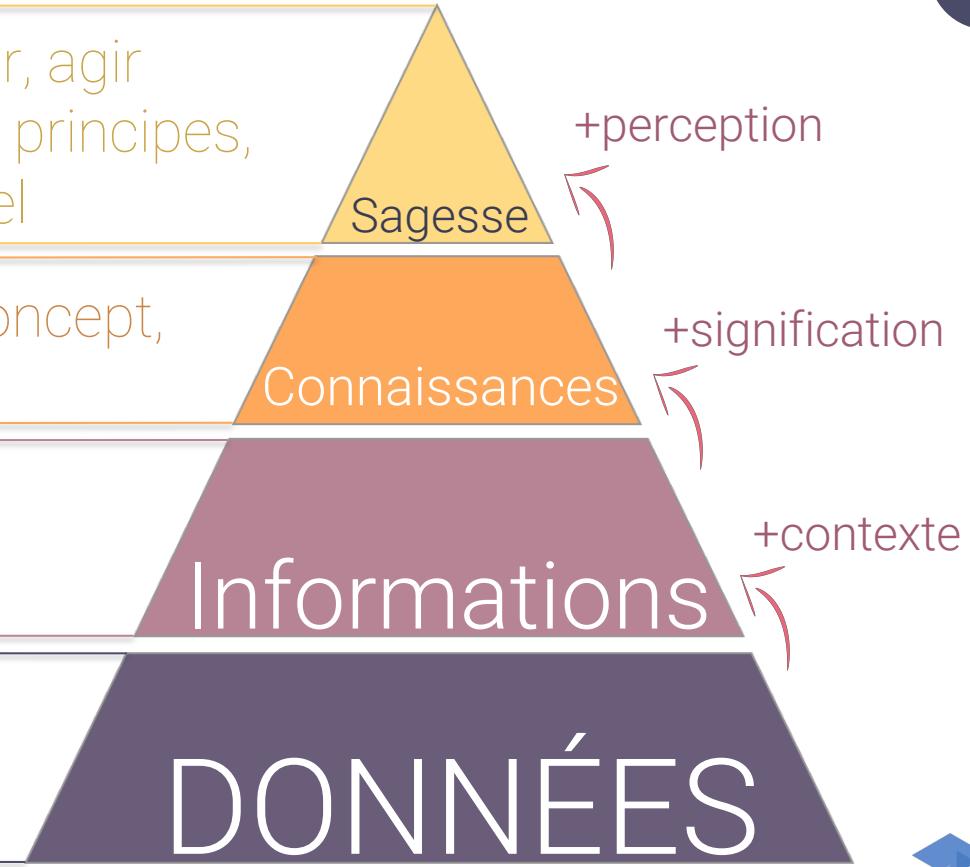
# Pyramide DIKW (\*)

Comprendre, intégrer, appliquer, agir  
Principes réfléchis, accumulés, principes,  
Modèles, processus décisionnel

Idée, Apprentissage, Notion, Concept,  
Synthétisé, comparé, discuté

Organisé, structuré, catégorisé  
Utile, Condensé, Calculé

Faits individuels, chiffres,  
Signaux, mesures



# Caractéristiques de l'information

## Introduction aux Systèmes d'Information

6

- La forme
  - Il peut être écrit, oral, visuel, olfactif, tactile ou goûté.
  - Peut être structuré, semi-structuré ou non-structuré
- Le contenu
  - Peut être synthétique, sélectif, précis
- Le coût et la valeur
  - Coût de son utilisation (recherche, collecte, traitement, stockage, destruction)
  - Valeur concernant :
    - Son impact sur la prise de décision
    - Son utilité pour l'utilisateur
  - Le coût est justifié s'il est inférieur à la valeur de l'information



# Qualité de l'information

## Introduction aux Systèmes d'Information

7

- Trois critères principaux pour déterminer la qualité d'une information
  - Fiabilité
    - Est-elle exacte et à jour ?
  - Disponibilité
    - Est-elle accessible au bon moment, aux bons destinataires et sous une forme directement et rapidement utilisable ?
  - Pertinence
    - Est-elle fidèle à la réalité, autorisée par la législation, non redondante, non calculée à partir d'autres informations ?



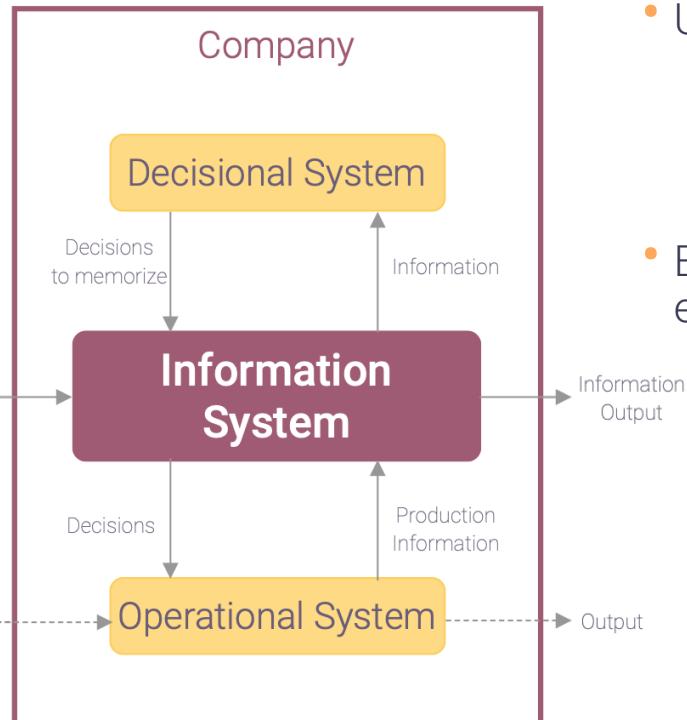
### Exemple

*Si vous enregistrez la date de naissance d'une personne dans une base de données, son âge n'est plus pertinent, car il peut être calculé à partir de sa date de naissance.*



# Le système d'entreprise

External Environment



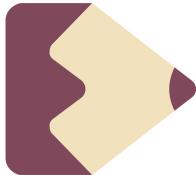
- Une entreprise est une organisation économique qui :
  - est intégrée dans un environnement
  - dispose d'une autonomie juridique
  - produit des biens et des services destinés à être vendus sur un marché
- Elle est donc composée de plusieurs sous-systèmes en interaction les uns avec les autres :
  - Système de production
    - Responsable de la production physique des biens et/ou services vendus
  - Système décisionnel
    - Analyse l'environnement et le comportement interne de l'entreprise
  - Système d'information
    - Alimente l'entreprise en informations
    - Mémorise, traite et communique les informations à d'autres sous-systèmes



# Système d'information : Définition

## Introduction aux Systèmes d'Information

9



### Définition

#### Système d'information

Ensemble unique de matériels, logiciels, bases de données, télécommunications, personnes et procédures configurées pour collecter, manipuler, stocker et transformer des données en informations

Stair & Reynolds - Principes fondamentaux des systèmes d'information.



# Système d'information : Composants

10

## Introduction aux Systèmes d'Information



# Système d'information : Fonctions

11

## Introduction aux Systèmes d'Information

- Quatre fonctions principales sont attribuées à un système d'information

### Acquisition d'informations

- Comprend la collecte d'informations, la sélection de celles qui sont pertinentes et le chargement dans le stockage du système d'information.
- Les informations sont recueillies auprès de sources externes et internes.

### Stockage des informations

- Stocke durablement les informations dans un système de stockage stable (principalement des bases de données)

### Exploitation de l'information

- Est capable d'effectuer des opérations de traitement sur les informations collectées : recherche, affichage, mise à jour et production de nouvelles informations à partir des informations existantes.

### Diffusion de l'information

- Fournir l'information à ceux qui en ont besoin au moment où ils en ont besoin, sous une forme exploitable.



# Système d'information : Développement

12

## Introduction aux Systèmes d'Information

- Création ou modification de systèmes d'information existants
- Comprend les sous-activités d'analyse, de conception, de développement, de mise en œuvre et d'évaluation
- **SDLC** : Cycle de vie du développement logiciel
  - Processus utilisé par l'industrie du logiciel pour concevoir, développer et tester des logiciels de haute qualité.
  - Composé de sept phases



# Système d'information : SDLC (\*)

13

## Introduction aux Systèmes d'Information

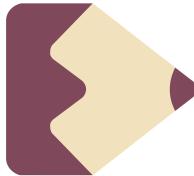


# Système d'information : SDLC

14

## Introduction aux Systèmes d'Information

- Attention, les phases du SDLC ne se déroulent pas toujours exactement dans cet ordre
  - Cela dépend de la méthodologie de conception utilisée



Définition

### Méthodologie de conception

Un plan d'action concret pour la conception de systèmes techniques (...). Elle comprend des plans d'action qui relient les étapes de travail et les phases de conception en fonction du contenu et de l'organisation.

Pahl & Beitz - *Conception technique : une approche systématique*

... mais c'est une histoire pour un autre cours



# Plan



- 1 Introduction aux Systèmes d'Information
- 2 Étapes de conception
- 3 Langage de modélisation unifié (UML)
- 4 Le Paradigme Orienté Objet



# Conception de logiciels

## Étapes de conception

16

- La conception de logiciels est un processus itératif par lequel les exigences sont traduites en un *plan de* construction du logiciel
  - Un plan est une reproduction d'un dessin technique, documentant une architecture ou un dessin technique
- Dans un premier temps, le plan décrit une vision globale du logiciel



# Processus d'ingénierie de conception

17

## Étapes de conception

- Au cours du processus de conception, les spécifications du logiciel sont transformées en modèles de conception
- Les **modèles** décrivent les détails des structures de données, l'architecture du système, l'interface et les composants
- La qualité de chaque **produit de conception** est examinée avant de passer à la phase suivante de développement du logiciel.
- À la fin du processus de conception, **un document de conception du système (SDD)** est produit.
- Ce document est composé des modèles de conception qui décrivent les données, l'architecture, les interfaces et les composants.



# Concepts fondamentaux de la conception

18

## Étapes de conception

- Abstraction
  - Permet aux concepteurs de se concentrer sur la résolution d'un problème sans se soucier des détails non pertinents de niveau inférieur
    - Abstraction procédurale : séquence d'événements nommés
    - Abstraction de données : collection nominative d'objets de données
- Raffinement
  - Processus d'élaboration où le concepteur fournit successivement plus de détails pour chaque élément du design
- Modularité
  - Mesure dans laquelle le logiciel peut être compris en examinant ses composants indépendamment les uns des autres



# Concepts fondamentaux de la conception : Patrons

## Étapes de conception

19

- Une solution commune à un problème commun dans un contexte donné
- Patrons de haut niveau pour l'organisation des logiciels : styles architecturaux
- Patrons de bas niveau pour décrire les détails
- Patrons de conception (*Design Patterns*)
  - Permettre à un concepteur de déterminer si le patron :
    - Est applicable au travail en cours
    - Peut être réutilisé
    - Peut servir de guide pour développer un patron similaire mais fonctionnellement ou structurellement différent



## Étapes de conception

- Une bonne répartition des modules est obtenue lorsqu'ils communiquent entre eux avec uniquement les informations nécessaires à la réalisation de la fonction du logiciel
- Appliquer les contraintes d'accès à
  - Détails de la procédure avec un module
  - Structure de données locale utilisée par ce module
- Avantages
  - Réduit la probabilité d'apparition d'"effets secondaires".
  - Limite l'impact global des décisions locales en matière de conception
  - Met l'accent sur la communication par le biais d'interfaces contrôlées
  - Décourage l'utilisation de données globales
  - Conduit à l'encapsulation - un attribut de la conception de haute qualité
  - Génère des logiciels de meilleure qualité

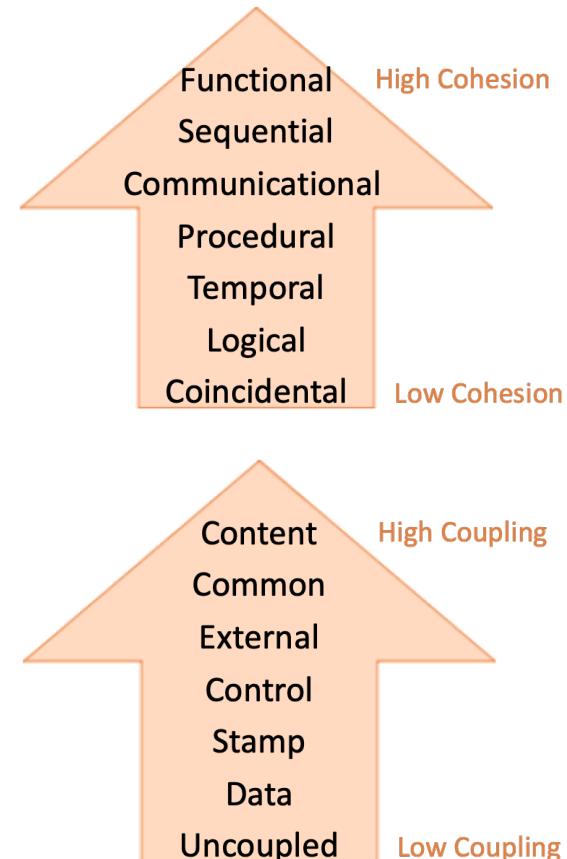


# Concepts fondamentaux de la conception : Indépendance fonctionnelle

21

## Étapes de conception

- Cohésion
  - Degré auquel un module remplit une seule et unique fonction
  - Tous les éléments d'une composante sont orientés vers la réalisation d'une même tâche
- Couplage
  - Degré de connexion d'un module à d'autres modules
  - Deux composantes peuvent être dépendantes de plusieurs façons :
    - Références faites les unes des autres
    - Quantité de données transmises par les uns et les autres
    - Degré de contrôle de l'un sur l'autre
    - ...

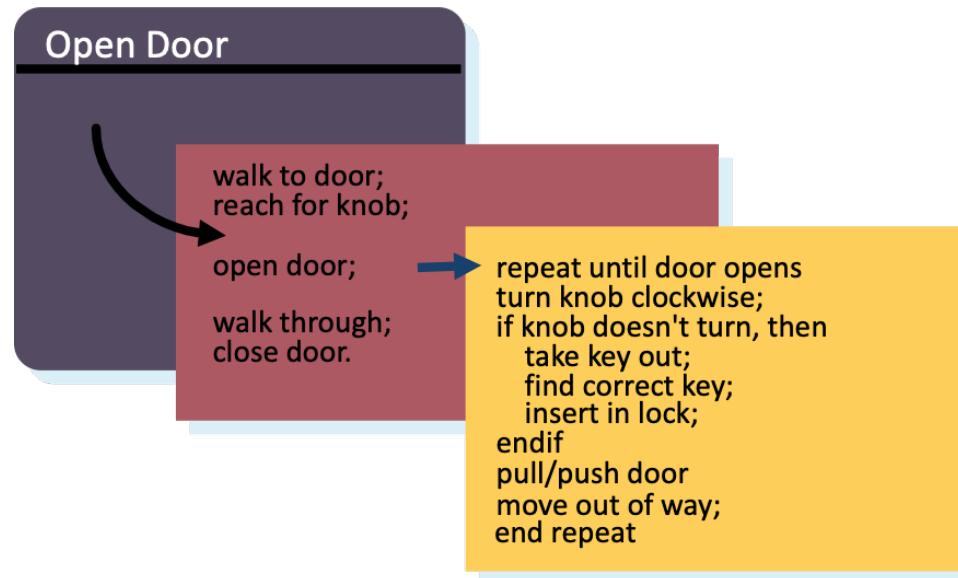


# Concepts fondamentaux de la conception : Raffinement

22

## Étapes de conception

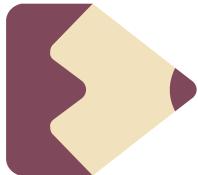
- Le raffinement est un processus d'élaboration
- Il s'agit d'une stratégie de conception descendante
- Un programme est élaboré en affinant les niveaux de détails des procédures



# Concepts fondamentaux de la conception : Refactoring

23

## Étapes de conception



### Définition

Le Refactoring est le processus qui consiste à modifier un système logiciel de manière à ne pas modifier le comportement externe du code, mais à améliorer sa structure interne.

*Martin Fowler - Refactoring : Améliorer la conception du code existant*

- Faire le refactoring d'un logiciel signifie examiner la conception existante pour :
  - Redondance
  - Éléments de conception non utilisés
  - Algorithmes inefficaces ou inutiles
  - Des structures de données mal construites ou inappropriées
  - Tout autre défaut de conception qui peut être corrigé pour obtenir une meilleure conception



# Plan



- 1 Introduction aux Systèmes d'Information
- 2 Étapes de conception
- 3 Langage de modélisation unifié (UML)
- 4 Le Paradigme Orienté Objet



# Modélisation visuelle

Langage de modélisation unifié (UML)

25

- La modélisation permet de saisir les éléments essentiels du système
- La modélisation visuelle est une modélisation utilisant des notations graphiques standard
- Modélisation visuelle
  - Saisit les processus commerciaux du point de vue de l'utilisateur
  - Est un outil de communication entre le domaine des entreprises et le domaine informatique
  - Gérer la complexité en utilisant des techniques de raffinement
  - Définit l'architecture logicielle
  - Favorise la réutilisation



## Langage de modélisation unifié (UML)

- Le *Unified Modelling Language* est un langage standard pour visualiser, spécifier, construire et documenter les artefacts d'un système logiciel.
- Combine des notions de :
  - Concepts de modélisation des données (diagrammes entité-relation)
  - Modélisation des entreprises (flux de travail)
  - Modélisation d'objets
  - Modélisation des composants
- Peut être utilisé avec tous les processus, tout au long du cycle de développement, et avec différentes technologies.



# Utilisation d'UML

Langage de modélisation unifié (UML)

27

- L'UML peut être utilisé pour
  - Afficher les limites d'un système et ses principales fonctions à l'aide de **cas d'utilisation** et d'**acteurs**
  - Illustrer les réalisations de cas d'utilisation à l'aide de **diagrammes d'interaction**
  - Représenter une structure statique d'un système à l'aide de **diagrammes de classe**
  - Modéliser le comportement des objets à l'aide de **diagrammes d'état-transition**
  - Révéler l'architecture physique de mise en œuvre avec des **diagrammes de composants et de déploiement**
  - Étendre les fonctionnalités de base à l'aide de **stéréotypes**
- UML n'est PAS :
  - Un langage de programmation
  - Un langage formel pour la démonstration de théorèmes



# Pourquoi "Unifié" ?

Langage de modélisation unifié (UML)

28

- A travers les méthodes et les notations historiques
  - Combine les concepts communément acceptés de nombreuses méthodes orientées objet
- Tout au long du cycle de vie du développement
  - Des exigences au déploiement en toute transparence
- Dans tous les domaines d'application
  - Modélise la plupart des domaines d'application, y compris les applications volumineuses, complexes, en temps réel, distribuées, à forte intensité de données ou de calculs, etc.
- Dans tous les langages et plates-formes de mise en œuvre
  - Utilisable pour des systèmes mis en œuvre dans différents langages et sur différentes plateformes
- Dans tous les processus de développement
  - Utilisable comme langage de modélisation sous-jacent à la plupart des processus de développement existants ou nouveaux
  - Supporte les modèles itératifs, incrémentaux, agiles,...
- Au-delà des concepts internes
  - Représente les relations internes d'une manière générale applicable à de nombreuses situations



Langage de modélisation unifié (UML)

- **Vue** : un sous-ensemble de constructions de modélisation UML qui représente un aspect d'un système
- Les vues sont divisées en zones :
  - Classification structurelle
    - Les choses dans le système et leurs relations avec d'autres choses
    - Les choses sont modélisées à l'aide de "classificateurs" (classe, cas d'utilisation, acteur, nœud, etc.)
  - Comportement dynamique
    - Comportement d'un système ou autre classificateur à travers le temps
    - Peut être décrit comme une série de modifications des instantanés du système tirés de la vue statique
  - Disposition physique
    - Ressources de calcul dans le système et déploiement d'artefacts sur celles-ci
  - Gestion des modèles
    - Organisation des modèles en unités hiérarchiques



# Vues UML (tel que définies par les trois amigos\*)

30

Langage de modélisation unifié (UML)

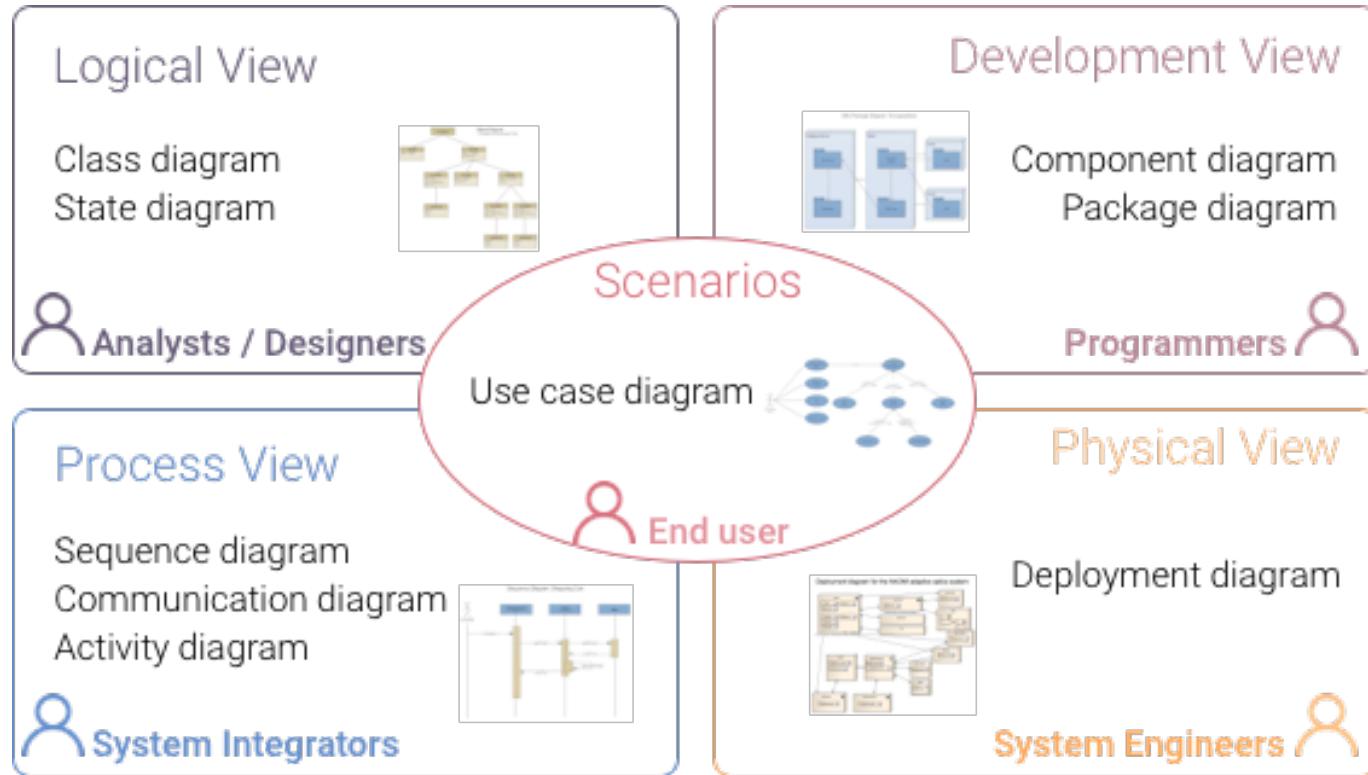
Domaine	Voir	Diagrammes
<i>Structural</i>	Vue statique	Diagramme de classe Structure interne
	Vue de la conception	Diagramme de collaboration Diagramme des composants
	Vue des cas d'utilisation	Diagramme de cas d'utilisation
<i>Dynamique</i>	Vue de la machine d'État	Diagramme d'état-transition
	Vue d'activité	Diagramme d'activité
	Vue d'interaction	Diagramme de séquence Diagramme de communication
<i>Physique</i>	Vue du déploiement	Diagramme de déploiement
<i>Gestion des modèles</i>	Vue de la gestion du modèle	Diagramme de paquet
	Profil	



# 4+1 Vues (\*)

31

Langage de modélisation unifié (UML)



# Plan



- 1 Introduction aux systèmes d'information
- 2 Étapes de conception
- 3 Langage de modélisation unifié (UML)
- 4 Le paradigme orienté objet (Rappel)



# Paradigme Orienté Objet... Pourquoi ?

Le Paradigme Orienté Objet

33

- Évolution très rapide du matériel contre évolution très lente des logiciels
- Concept clé : Réutilisation
- Nécessité de réutiliser les éléments logiciels au lieu de tout redévelopper à partir de zéro dans chaque projet.
- Il faut une approche ascendante, partant d'éléments simples pour construire des éléments plus grands et plus complexes
  - Le paradigme orienté objet



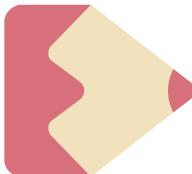
## Le Paradigme Orienté Objet

- Entité cohérente qui regroupe les données et les codes travaillant sur ces données
- Caractérisé par :
  - Son comportement : *que peut faire cet objet ?*
    - Méthodes
  - Son état : *Comment cet objet réagit-il lorsque le comportement change ?*
    - Attributs
  - Son identité : *Comment distinguer les objets qui ont le même état et le même comportement ?*
    - Identifiant
- Un objet a les mêmes réactions et la même modularité que dans le monde réel
  - Un objet logiciel est la projection de l'objet du monde réel



## Le Paradigme Orienté Objet

- Composante de base
- Contient la description d'un objet
  - Modèle d'objet efficace
- Correspond à l'*idée d'un objet*
  - Analogie avec la philosophie idéaliste platonique :



### Citation

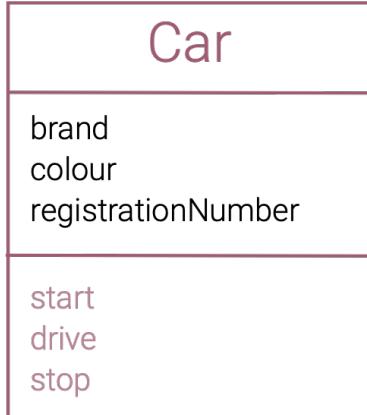
*Vous vous promenez dans la campagne, vous pensez avoir rencontré des troupeaux de chevaux. Quelle erreur !  
(...) Parce que le Cheval-Modèle, le Cheval-Idée, n'est ni noir ni blanc, il n'est d'aucune race de cheval. C'est un cheval pur et vos sens ne vous le montreront jamais ...*

A. Bonnard- Civilisation Grecque



# Exemple

## Le Paradigme Orienté Objet



```

class Car {
    // attributes
    String brand;
    String colour;
    String registrationNumber;

    Car(String brand, String colour, String reg){
        this.brand = brand;
        this.colour = colour;
        this.registrationNumber = reg;
    }

    // methods
    void start(){}
    void drive(){}
    void stop(){}
}

Car twingo = new Car("Renault","grey","111 111");
  
```

The code shows the implementation of the 'Car' class in Java. It includes attribute declarations for 'brand', 'colour', and 'registrationNumber', a constructor that initializes these attributes, and three methods: 'start()', 'drive()', and 'stop()'. An example of creating a 'Car' object named 'twingo' is also provided.



# Approche OO : Concepts fondamentaux

Le Paradigme Orienté Objet

37



INHERITANCE

Information sharing



ENCAPSULATION

Information grouping



ABSTRACTION

Information hiding



POLYMORPHISM

Information redefining



# Encapsulation

## Le Paradigme Orienté Objet

38

- Mécanisme qui regroupe, dans une même structure, les données et les traitements
  - Définition des attributs et des méthodes dans la classe
- L'implémentation de la classe est cachée à l'utilisateur
  - Le concept d'**interface** : vue extérieure d'un objet
- La possibilité de modifier l'implémentation sans modifier l'interface
  - Plus facile de faire évoluer l'objet
- Préserver l'intégrité des données
  - L'accès direct aux attributs est interdit
  - L'interaction entre les objets se fait uniquement par des méthodes



- Un objet spécial bénéficie ou hérite des caractéristiques de l'objet plus général, auquel il ajoute ses propres éléments
  - Création de nouvelles classes, sur la base des classes existantes
  - Transmission des propriétés (attributs et méthodes) de la classe mère à la classe enfant
- Représente la relation "*est un ...*"
- Deux orientations possibles
  - Spécialisation : Ajout / adaptation de caractéristiques
  - Généralisation : Regroupe les caractéristiques communes
- Possibilité d'héritage multiple (pas possible dans certaines langues)
- Avantages
  - Évite la duplication du code
  - Encourage la réutilisation du code



# Polymorphisme

Le Paradigme Orienté Objet

40

- Définition :
  - *Poly* : beaucoup
  - *Morphisme* : formes
- Capacité d'une méthode à être appliquée à des objets de plusieurs classes
- Capacité d'une classe à redéfinir une méthode héritée d'une classe mère
  - Surcharge
- Avantages
  - Lisibilité du code
  - Caractère générique du code



# Abstraction

## Le Paradigme Orienté Objet

41

- L'abstraction est une extension de l'encapsulation
- Permet de masquer certaines propriétés et méthodes du code extérieur pour simplifier l'interface des objets
- Avantages
  - Minimiser de l'impact du changement
  - Cacher à l'utilisateur des informations inutiles
  - Mise en œuvre possible d'une nouvelle logique sans penser à la complexité cachée



# Pour le prochain cours



- Téléchargez et installez Visual Paradigm (édition communautaire) ici :  
<https://www.visual-paradigm.com/download/community.jsp>
- Regardez la vidéo suivante, magnifiquement créée par Lucidchart, présentant les diagrammes de cas d'utilisation :  
<https://www.youtube.com/watch?v=J8NtoLxhoRc>



# TI615M

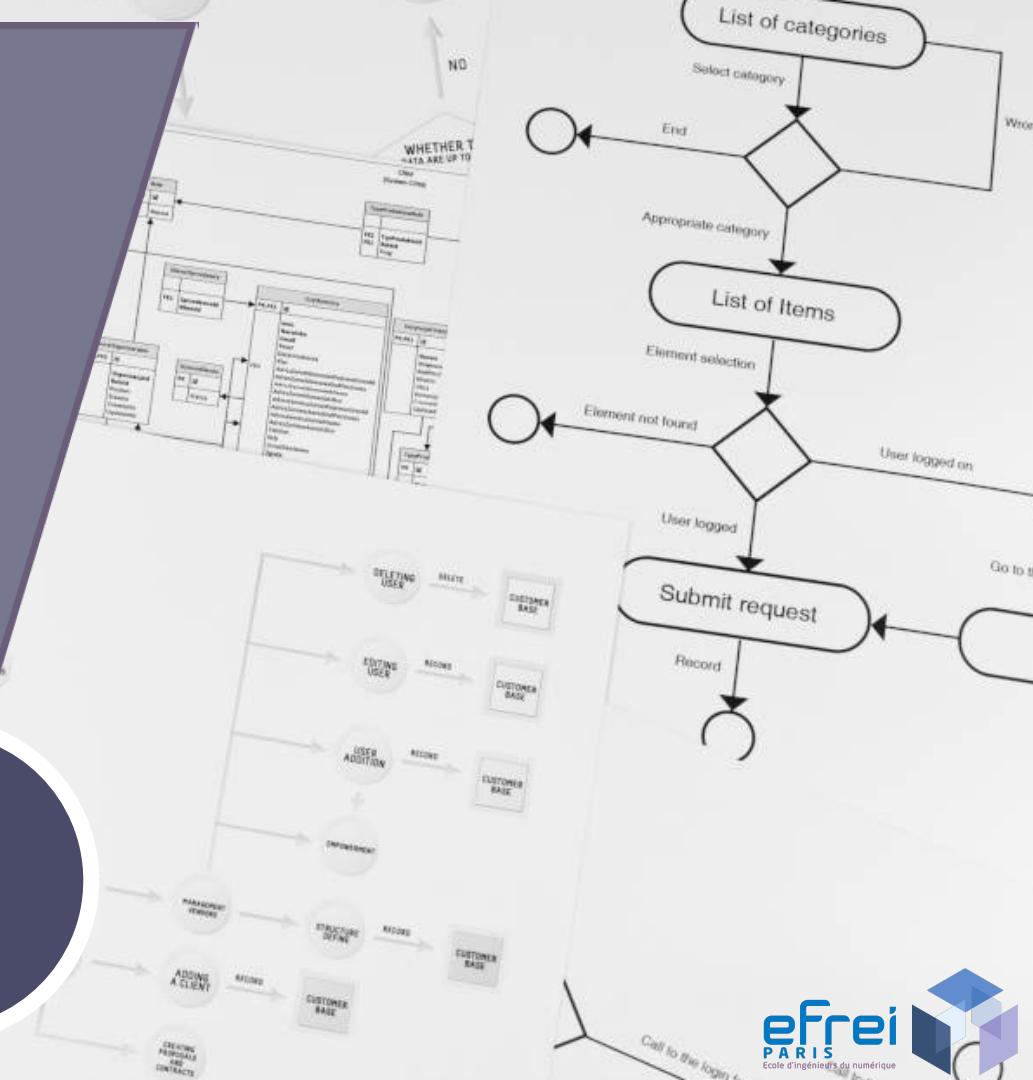
## Analyse et Conception Orientée Objet avec

# UML

Dr. Lilia SFAXI

## Diagramme des cas d'utilisation

2



# Plan



- 1 La vue et le diagramme des cas d'utilisation
- 2 Concepts de base
- 3 Concepts de raffinement
- 4 Documentation des cas d'utilisation



# Plan

- 1 La vue et le diagramme des cas d'utilisation
- 2 Concepts de base
- 3 Concepts de raffinement
- 4 Documentation des cas d'utilisation



# La vue des cas d'utilisation

La vue et le diagramme des cas d'utilisation

- Modélise la fonctionnalité d'un **sujet** (tel qu'un système) telle qu'elle est perçue par des agents extérieurs, appelés **acteurs**
- Les acteurs interagissent avec le sujet d'un point de vue particulier
- Cas d'utilisation
  - Un cas d'utilisation est une unité de fonctionnalité exprimée comme une transaction entre les acteurs et le sujet
- Objectif :
  - Liste des acteurs et des cas d'utilisation
  - Montrer quels acteurs participent à chaque cas d'utilisation
- Une vue de cas d'utilisation **ne représente pas** le comportement dynamique (lié au temps) des cas d'utilisation
  - Ceci est illustré dans la vue d'interaction (dynamique)



# Diagramme de cas d'utilisation

5

La vue et le diagramme des cas d'utilisation

- Schéma fonctionnel en UML
- Représente l'utilisation et les fonctionnalités du système
- Représente les interactions entre les utilisateurs et le système
- Est une représentation graphique (diagramme) accompagnée d'une description textuelle



# Plan

- 1 La vue et le diagramme des cas d'utilisation
- 2 Concepts de base
- 3 Concepts de raffinement
- 4 Documentation des cas d'utilisation



# Concepts de base du diagramme de cas d'utilisation

7

## Concepts de base

- Acteur
  - Rôle donné à toute entité externe qui interagit avec le système
- Cas d'utilisation
  - Spécification d'un ensemble d'actions effectuées par le système
- Relations
  - Association
  - Inclusion
  - Extension
  - Généralisation



Actor



<< include >>



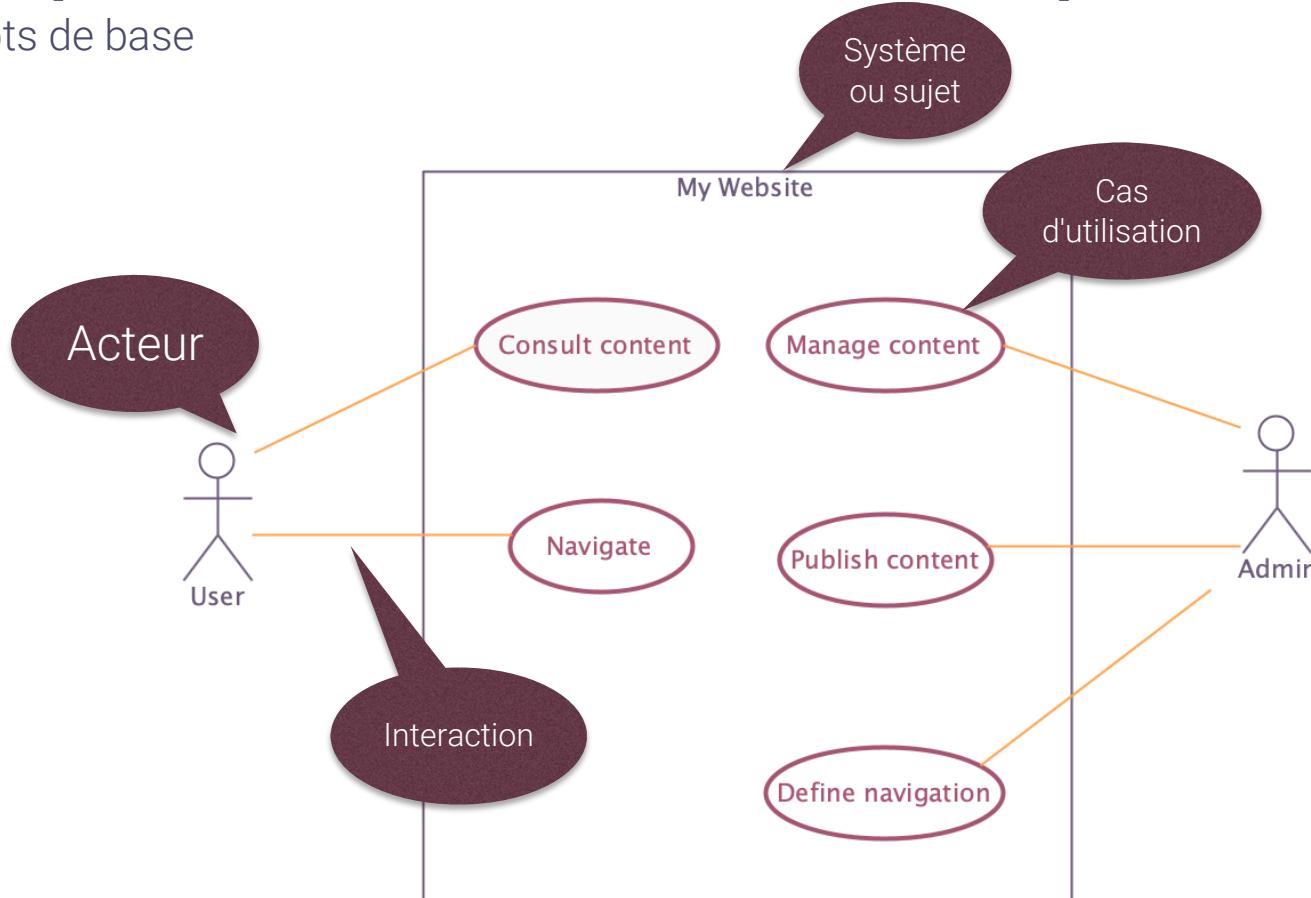
<< extend >>



# Exemple d'un cas d'utilisation simple

8

Concepts de base





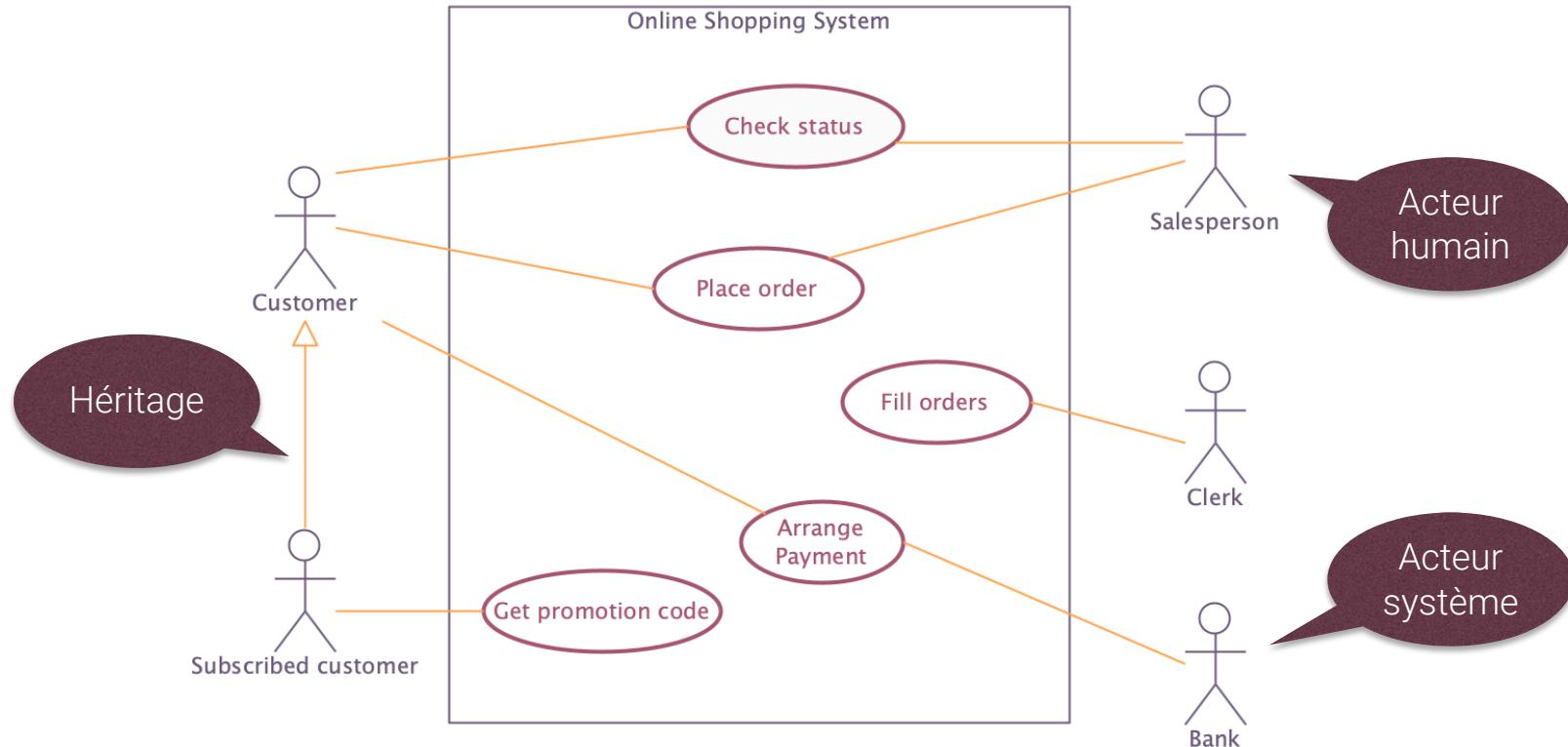
- Un acteur
  - Idéalisat d'un rôle joué par une personne, un processus ou une chose externe en interaction avec un système, un sous-système ou une classe
  - caractérise les interactions qu'une catégorie d'utilisateurs extérieurs peut avoir avec le système
- Un utilisateur physique peut être lié à plusieurs acteurs au sein du système
- Différents utilisateurs peuvent être liés au même acteur, représentant ainsi plusieurs instances de la même définition d'acteur
- Chaque acteur
  - participe à un ou plusieurs cas d'utilisation
  - Interagit avec le cas d'utilisation en échangeant des messages
- Les acteurs peuvent être définis dans des hiérarchies de généralisation
  - Une description abstraite d'acteur est partagée et complétée par une ou plusieurs descriptions d'acteur spécifiques



# Exemple d'acteurs

10

Concepts de base



# Cas d'utilisation

use the case

11

Concepts de base

- Unité cohérente de fonctionnalité visible de l'extérieur fournie par le système et exprimée par des séquences de messages échangés par le système et un ou plusieurs acteurs
- Définit un comportement cohérent sans révéler la structure interne du système
- Dans le modèle, l'exécution de chaque cas d'utilisation est indépendante des autres
  - Mais leur mise en œuvre peut créer des dépendances implicites dues à des objets partagés
- La dynamique d'un cas d'utilisation peut être spécifiée par des interactions UML, présentées sous forme de **diagrammes d'état**, de **diagrammes de séquence** ou de **communication**, ou de **descriptions textuelles informelles**

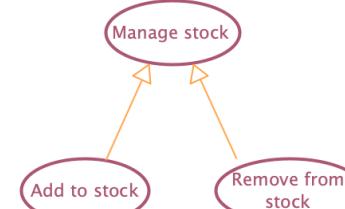


# Relations des cas d'utilisation (1)

## Concepts de base



- Association
  - Le chemin de communication entre un acteur et un cas d'utilisation auquel il participe
- Inclusion
  - montre que le comportement du cas d'utilisation inclus (l'ajout) est inséré dans le comportement du cas d'utilisation inclus (la base).
- Extension
  - spécifie comment et quand le comportement défini dans le cas d'utilisation d'extension (facultatif) peut être inséré dans le comportement défini dans le cas d'utilisation étendu
- Généralisation
  - Une relation entre un cas d'utilisation général et un cas d'utilisation plus spécifique qui en hérite et y ajoute des caractéristiques



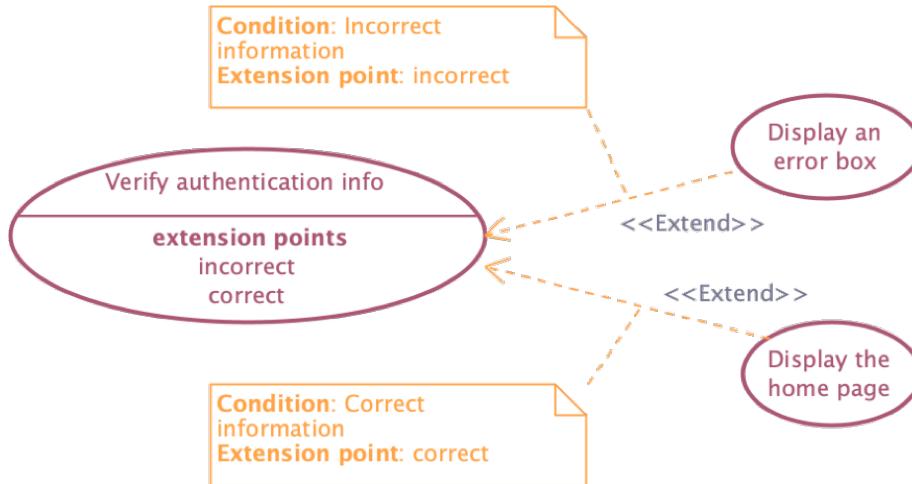
# Relations des cas d'utilisation (2)

## Concepts de base



- Points d'extension

- Un marqueur nommé qui identifie un endroit ou un ensemble d'endroits dans la séquence comportementale d'un cas d'utilisation, où un comportement supplémentaire peut être inséré
- Une déclaration de point d'extension ouvre le cas d'utilisation à la possibilité d'extension



# Aperçu



- 1 La vue et le diagramme des cas d'utilisation
- 2 Concepts de base
- 3 Concepts de raffinement
- 4 Documentation sur les cas d'utilisation

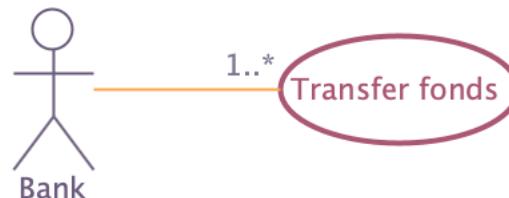


# Multiplicité (1)

15

Concepts de raffinement

- Une spécification du champ des valeurs de cardinalités autorisées (tailles) qu'une collection peut assumer
  - Un sous-ensemble éventuellement infini d'entiers non négatifs
  - Un acteur est lié à un cas d'utilisation avec une multiplicité  $> 1$ , représentée à la fin de l'association, à côté du cas d'utilisation
  - L'acteur est impliqué dans de nombreux cas d'utilisation de ce type



# Multiplicité (2)

Concepts de raffinement

16

- Un cas d'utilisation est lié à un acteur, avec une multiplicité > 1, représenté à la fin de l'association, à côté de l'acteur
  - De nombreuses instances de cet acteur sont impliquées dans l'exécution de ce cas d'utilisation



# Plan

- 
- 1 La vue et le diagramme des cas d'utilisation
  - 2 Concepts de base
  - 3 Concepts de raffinement
  - 4 Documentation des cas d'utilisation



# Documentation

## Documentation des cas d'utilisation

18

Élément de cas d'utilisation	Description
Numéro	Identifiant pour représenter votre cas d'utilisation
Application	Quel est le système ou l'application concerné
Nom	Le nom de votre cas d'utilisation, soyez bref
Description	Préciser davantage le nom, sous forme de paragraphe
Acteur principal	Qui est l'acteur principal que représente ce cas d'utilisation
Condition préalable	Quelles conditions préalables doivent être remplies avant que ce cas d'utilisation puisse commencer
Déclencheur	Quel est l'événement qui déclenche le cas d'utilisation
Flux de base	Ce devrait être les événements du cas d'utilisation où tout est parfait, sans erreur, sans exception.
Flux alternatif	Les alternatives et exceptions les plus significatives



# Exemple

## Documentation des cas d'utilisation

19

- Prenons le cas d'utilisation suivant, qui décrit une personne faisant la lessive.
- Dans l'exemple suivant :
  - Un employé de ménage fait la lessive le mercredi
  - Elle lave chaque chargement.
  - Elle sèche chaque chargement.
  - Elle plie certains articles.
  - Elle repasse certains articles.
  - Elle jette certains objets.



# Exemple : Faire la lessive - Partie 1(\*)

20

Documentation des cas d'utilisation

Élément de cas d'utilisation	Description
Numéro	1
Nom	Faire la lessive
Description du cas d'utilisation	Nous sommes mercredi et il y a du linge sale dans la buanderie. L'employé de ménage le trie, puis procède au lavage de chaque brassée. Elle plie le linge sec lorsqu'elle le retire du sèche-linge. Elle repasse les articles qui ont besoin d'être repassés.
Acteur principal	L'employé de ménage
Conditions préalables	1. Nous sommes mercredi 2. Il y a du linge sale dans la salle de lavage.
Déclencheur	Le mercredi, le linge sale est transporté à la buanderie.



# Exemple : Faire la lessive - Partie 2

21

Documentation des cas d'utilisation

Élément de cas d'utilisation	Description
Flux de base	<p><b>Description :</b> Ce scénario décrit la situation dans laquelle seuls le tri, le lavage et le pliage sont nécessaires. C'est le scenario principal.</p> <ol style="list-style-type: none"><li>1. L'employé de ménage trie les articles sales.</li><li>2. L'employé de ménage lave chaque chargement.</li><li>3. L'employé de ménage sèche chaque chargement.</li><li>4. L'employé de ménage vérifie que le linge n'a pas besoin d'être repassé, qu'il est propre et non rétréci.</li><li>5. L'employé de ménage vérifie que l'article lavé est pliable.</li><li>6. L'employé de ménage plie l'article</li><li>7. L'employé de ménage fait cela jusqu'à ce qu'il n'y ait plus de linge à plier.</li></ol> <p><b>Résultat de l'exécution :</b> Le linge est propre et plié</p>



# Exemple : Faire la lessive - Partie 3

22

Documentation sur les cas d'utilisation

Élément de cas d'utilisation	Description
Flux alternatif 1	<p><b>Le linge doit être repassé.</b></p> <p><b>Description :</b> Ce scénario décrit la situation dans laquelle un ou plusieurs articles doivent être repassés avant ou au lieu d'être pliés</p> <ol style="list-style-type: none"><li>1. L'employé de ménage vérifie que le linge doit être repassé, qu'il est propre et non rétréci</li><li>2. L'employé de ménage repasse le linge</li><li>3. L'employé de ménage met un article de linge sur un cintre</li></ol> <p><b>Résultat de l'exécution :</b> Le linge qui doit être repassé est repassé et accroché.</p>



# Exemple : Faire la lessive - Partie 4

23

Documentation sur les cas d'utilisation

Élément de cas d'utilisation	Description
Flux alternatif 2	<p><b>Le linge est sale.</b></p> <p><b>Description :</b> Ce scénario décrit la situation où le linge n'a pas été nettoyé la première fois par le lavage.</p> <ol style="list-style-type: none"><li>1. L'employé de ménage vérifie que le linge n'est pas propre.</li><li>2. L'employé de ménage lave à nouveau le linge</li></ol> <p><b>Résultat de l'exécution :</b> Le linge sale est relavé.</p>



# Représentation des flux de base et alternatifs

Documentation sur les cas d'utilisation

24

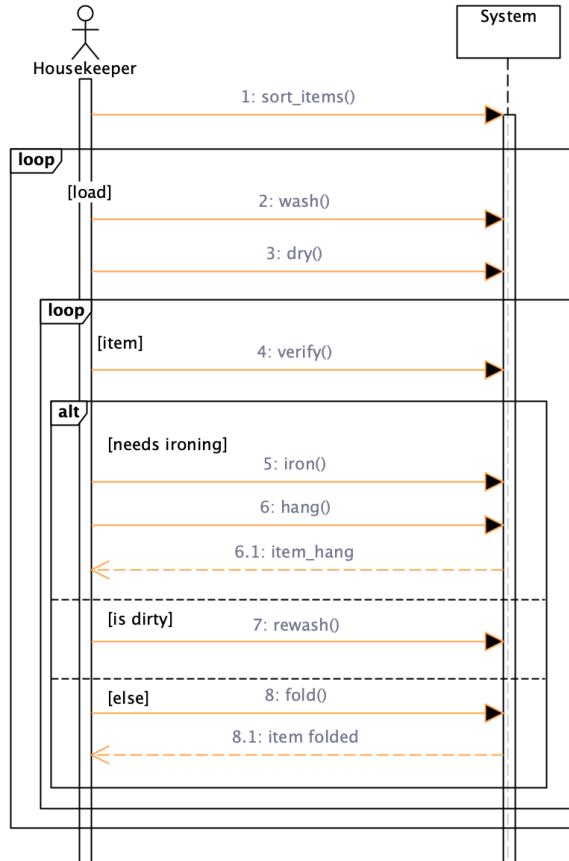
- Il est possible de représenter les flux à l'aide d'un **diagramme de séquence de système**
- Un **diagramme de séquence du système** décrit le comportement du système, où il est vu comme une boîte noire
- Le système est considéré de l'extérieur, par les acteurs, sans idée de la manière dont il sera réalisé
- Le cas d'utilisation est décrit sous la forme d'une séquence de messages échangés entre les acteurs et le système



# Diagramme de séquence système

25

Documentation sur les cas d'utilisation



TI615M  
Analyse et Conception Orientée Objet avec

# UML

Dr. Lilia SFAXI

Diagramme de classes et  
d'objets

3



# Plan

1

Diagramme de classe

2

Diagramme d'objet



# Plan

1

Diagramme de classe

2

Diagramme d'objet



# Diagramme de classe

4

## Diagramme de classe

- Un diagramme de classes divise le système en domaines de responsabilité (classes) et montre les "associations" (dépendances) entre eux.
- Fournit une image ou une vue de certaines ou de toutes les classes/interfaces du modèle
- Vue statique de la conception du système
- Les attributs (données), les opérations (méthodes), les contraintes, les relations de partie (navigabilité) et de type (héritage), l'accès et la cardinalité (1 à plusieurs) peuvent tous être notés.



# Perspectives des diagrammes de classe

5

## Diagramme de classe

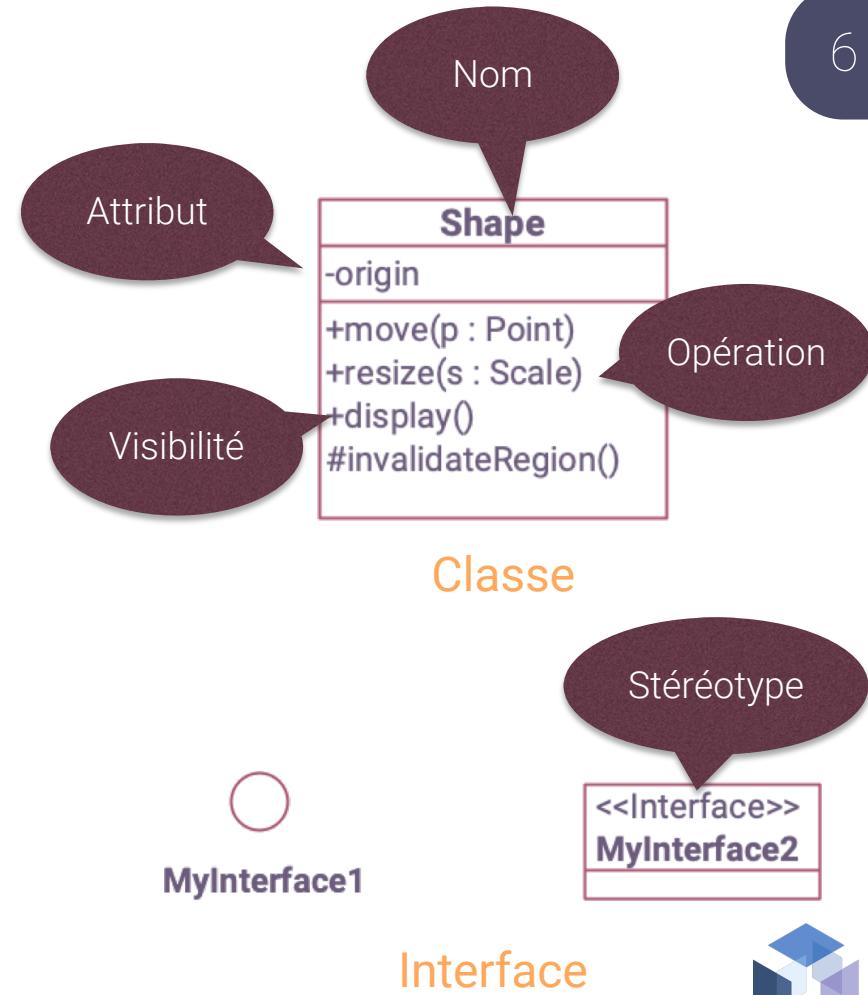
- Les diagrammes de classe peuvent avoir un sens à trois niveaux distincts, ou perspectives :
  - **Conceptuelle** :
    - Le diagramme représente les concepts dans le domaine du projet.
    - Il s'agit d'une répartition des rôles et des responsabilités dans ce domaine.
  - **Spécification** :
    - Affiche les interfaces entre les composants du logiciel.
    - Les interfaces sont indépendantes de la mise en œuvre.
  - **Mise en œuvre** :
    - Affiche les classes qui correspondent directement au code informatique (souvent des classes Java ou C++).
    - Sert de schéma directeur pour une réalisation effective du logiciel en code.



# Structure statique

## Diagramme de classe

- Notion de classe : une description d'un groupe d'objets avec :
  - des propriétés communes (attributs),
  - un comportement commun (opérations),
  - des relations communes avec d'autres objets, et une sémantique commune.
- En UML, les classes sont représentées par des rectangles compartimentés :
  - le compartiment supérieur contient le nom de la classe
  - le compartiment du milieu contient la structure de la classe (attributs)
  - le compartiment inférieur contient le comportement de la classe (opérations)

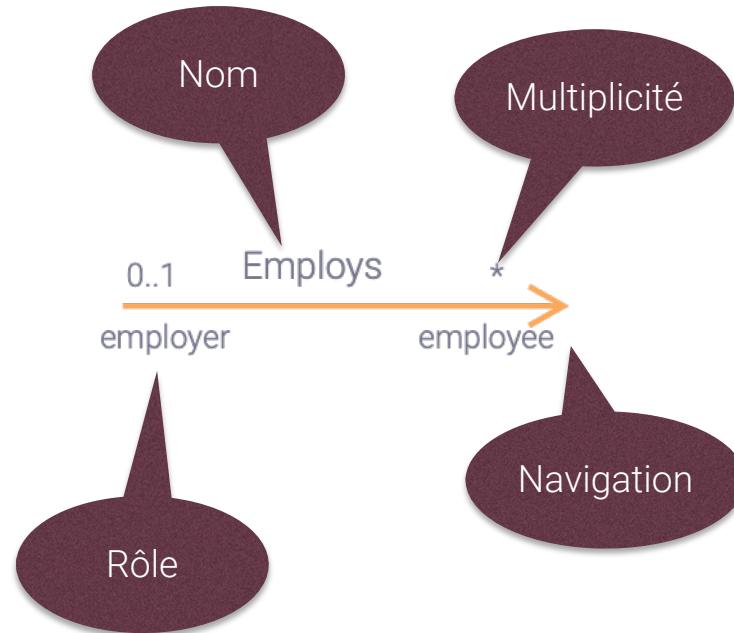
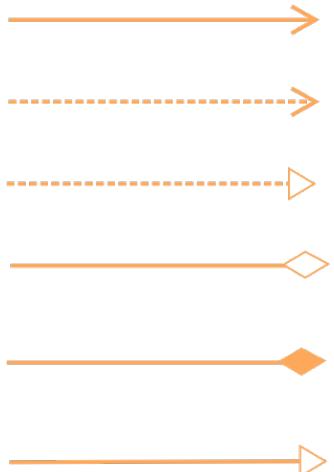


# Relations

7

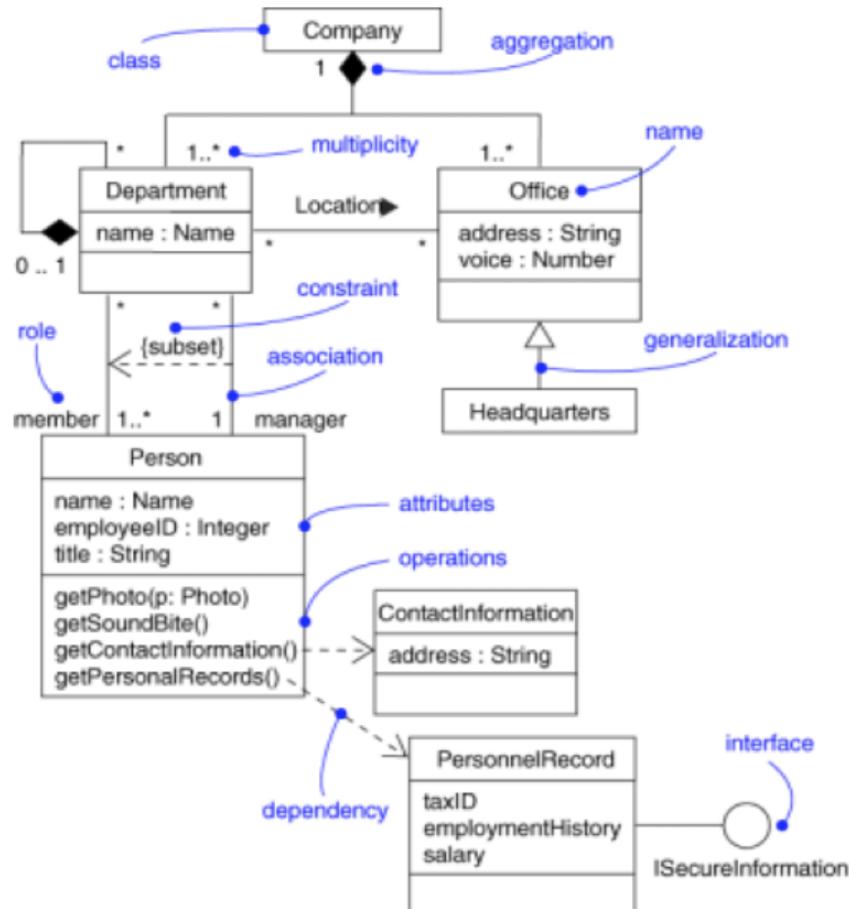
## Diagramme de classe

- Fournir le conduit pour l'interaction des objets
- Plusieurs types de relations
  - Association
  - Dépendance
  - Réalisation
  - Agrégation
  - Composition
  - Héritage



# Un exemple d'entreprise

Diagramme de classe



# Stéréotypes

9

## Diagramme de classe

- Fournir la capacité de créer un nouveau type d'élément de modélisation.
- Nous pouvons créer de nouveaux types de classes en définissant des stéréotypes pour les classes.
- Le stéréotype d'une classe est indiqué sous le nom de la classe enfermé dans des guillemets (<< >>).
  - Exemples de stéréotypes de classe : Exception, Utilité, etc.
- L'élément stéréotypé peut avoir sa propre icône distincte
  - Entité, Limite, Contrôle, Acteur sont des stéréotypes prédéfinis avec des icônes



# Attributs

10

## Diagramme de classe

- Données intégrées dans les objets de la classe
- Définies par un nom, un type et une visibilité
- Formulaire
  - <visibilité> [ / ] <nom\_de\_l'attribut> : <type> [ ' [ '<multiplicité>' ]  
[ {<contrainte>} ] ] [ = <valeur\_par\_défaut> ]
  - Exemple : + couleur : int [3] {liste}
- Attribut de classe
  - Un attribut spécifique à la classe, et non à l'objet (appelé **statique** en Java)
  - Conserve une valeur unique pour toutes les instances d'une même classe
  - Est souligné en UML



# Méthodes

## Diagramme de classe

- Décrit une fonction de la classe
- Doit contenir un nom, un type de retour et des paramètres
- Formulaire :
  - <visibilité> <nom\_meth> ([<param\_1>, ... , <param\_N>]) : [<type\_retour>] [{<propriétés>}]
- Un paramètre a la forme :
  - [<direction>]<nom\_param>:<type>['['<multiplicité>']] [=<valeur\_par défaut>]
- Exemple
  - + déplacement (in distance : int = 2) : void {abstract}



# Méthodes abstraites

12

## Diagramme de classe

- Méthodes sans implémentation
- Doivent être surchargées
- Si une méthode abstraite non surchargée est appelée, une erreur est déclenchée
- Utilisée si l'on veut définir un squelette pour un objet qui a de nombreux descendants qui doivent tous avoir un comportement analogue.
- Exemple :
  - Dans la classe mère :
    - `void print (message de chaîne) ; // pas d'implémentation`
  - Dans la classe fille :
    - `void print (message de chaîne){ ... //implémentation }`



# Interfaces

## Diagramme de classe

13

- Un type particulier de classe, où toutes les méthodes sont abstraites
  - Stéréotype <<interface>>
- Permet le regroupement d'un ensemble de biens et d'opérations réalisant un service cohérent
- Doit être implémentée par au moins une classe concrète
  - Relation : Réalisation
- Une classe peut dépendre d'une interface
  - Relation : Dépendance



# Relation d'association

14

## Diagramme de classe

- Une relation entre deux classes ou plus, décrivant les connexions structurelles entre leurs instances
- Relie les classes d'un même niveau hiérarchique
- Quatre décorations peuvent préciser les liens entre les objets :
  - Nom : nature des relations entre les objets
  - Direction : direction de l'application du nom
  - Rôle : Rôle spécifique de chaque classe de l'association
  - La multiplicité : Nombre d'éléments concernés par la relation

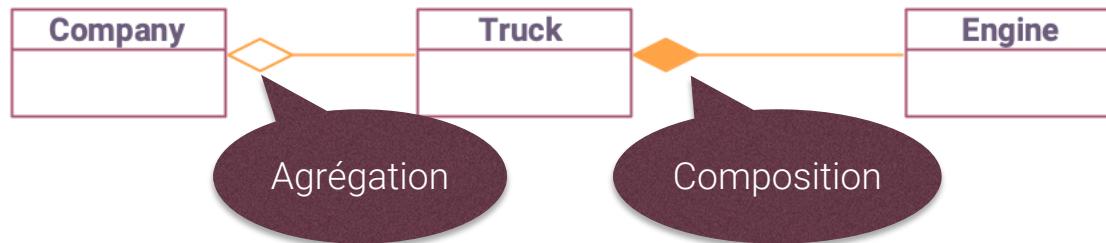


# Relations d'agrégation et de composition

15

Diagramme de classe

- Agrégation
  - Définit une relation hiérarchique entre les entités
  - Définit la relation : "*est composé de*" et modélise la notion de "*tout et partie*".
- Composition
  - Définit l'appartenance structurelle entre les instances
  - La création de l'objet composant est la responsabilité du composite
  - Une instance du composant appartient au maximum à une instance du composite

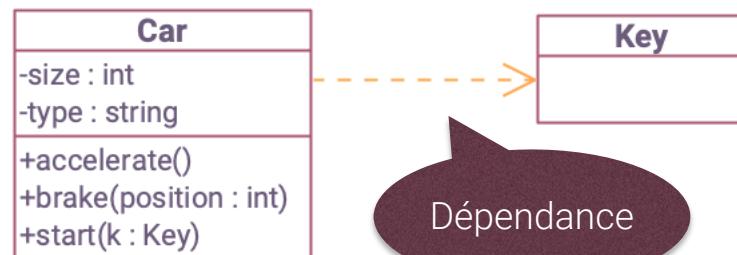


# Relations de dépendance

16

## Diagramme de classe

- Une dépendance établit une relation entre deux entités d'un même diagramme
- La plupart du temps, il s'agit d'une relation d'*utilisation* (*use*) :
  - Comme argument d'une méthode, par exemple
- Cela permet de définir les implications possibles de la modification d'une entité
  - Modification telle que le changement de comportement d'une classe

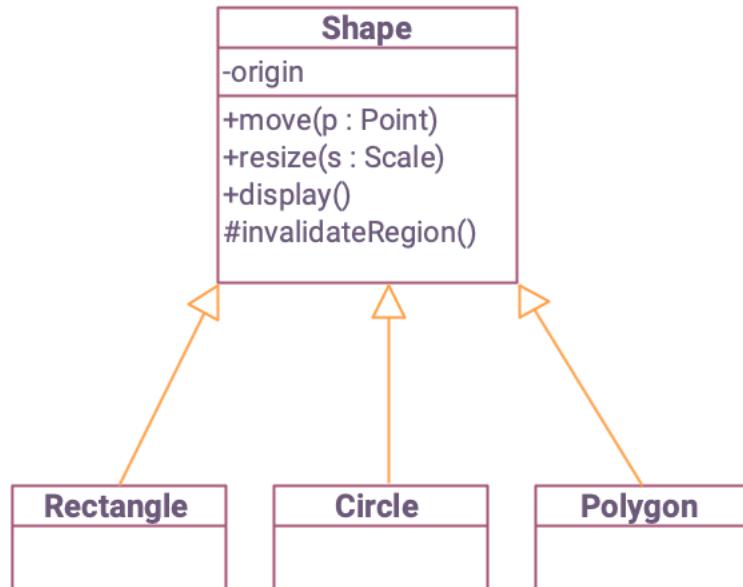


# Généralisation de la relation

17

## Diagramme de classe

- Modélisation de la relation d'héritage
  - Correspond à la notion de "est une sorte de".
  - Modélisation de la relation "parent / enfant".
- Les entités issues d'une généralisation sont utilisées partout où se trouve leur classe mère (mais pas l'inverse)
- Cette relation est modélisée par une flèche triangulaire vide pointant vers la classe mère.

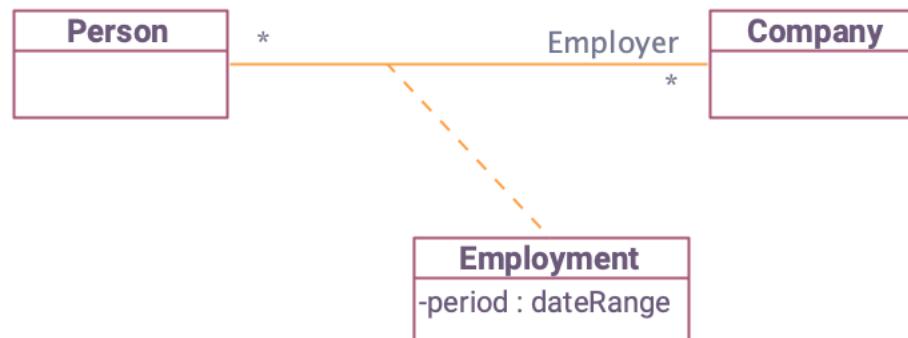


# Classe d'association

18

## Diagramme de classe

- Une association peut avoir des propriétés qui ne sont pas disponibles dans les classes qu'elle relie
  - Nous définissons ces propriétés dans une classe d'association.
- Les classes d'association vous permettent d'ajouter des attributs, des opérations et d'autres caractéristiques aux associations
- Attention, l'exemple suivant ne permettrait pas à une personne d'avoir plus d'un emploi au sein de la même entreprise !
  - Si vous voulez l'autoriser, *Employment* doit être une classe à part entière

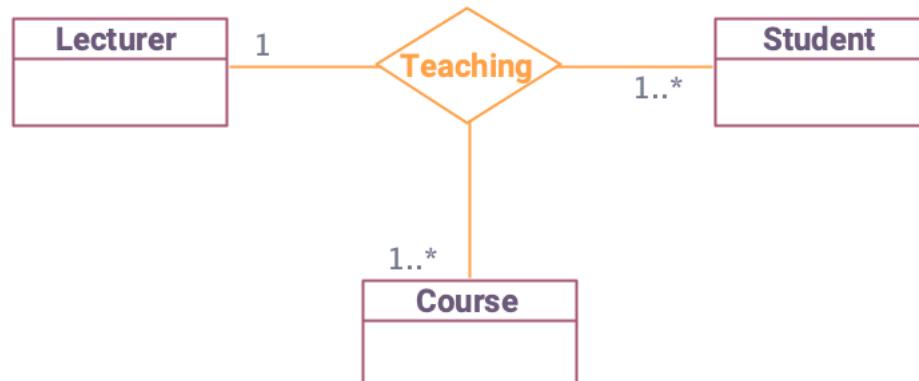


# Association N-ary

19

Diagramme de classe

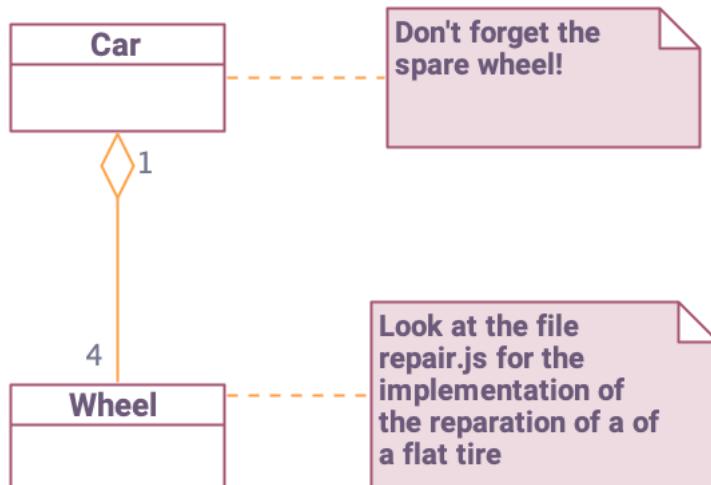
- Une association qui relie plus que deux classes
- Peu utilisée, car il est difficile de gérer les multiplicités !
- Dans l'exemple ci-dessous, la multiplicité en relation avec le conferencier (Lecturer) se lit comme suit :
  - Toute paire fixe d'objets Étudiant et Cours correspond à un seul d'objet Enseignant



# Notes (ou commentaires)

## Diagramme de classe

- Lors de la modélisation d'une spécification, il est nécessaire de documenter les modèles
  - Contraintes matérielles
  - Contraintes de performance
  - Les choix techniques
  - Références à d'autres documents
  - Explications techniques
  - etc.
- En UML, nous utilisons des *notes*



# Plan

1

Diagramme de classe

2

Diagramme d'objet



# Diagramme d'objet

22

## Diagramme d'objet

- Un diagramme d'objets représente les objets (instances de classes) et leurs liens (instances de relations) pour donner une vue statique de l'état du système à un moment donné
- Peut être utilisé pour:
  - Illustrer le modèle de classe en montrant un exemple qui explique le modèle
  - Préciser certains aspects du système en mettant en évidence les détails imperceptibles dans le diagramme de classe
  - Montrer une exception en modélisant les cas spécifiques ou les connaissances non généralisables qui ne sont pas modélisés dans un diagramme de classe
- **Le diagramme de classes modélise les règles et le diagramme d'objets modélise les faits**



# Graphiquement

23

## Diagramme d'objet

- Un objet est représenté comme une classe, sans le compartiment des méthodes
- Le nom de l'objet est composé du nom de l'instance, deux points, puis le nom de la classe, et est souligné
- Les attributs ont des valeurs
  - Si certaines valeurs ne sont pas définies, l'objet est dit **partiellement défini**
  - La relation de généralisation n'est jamais représentée
  - Les multiplicités ne sont jamais représentées
  - La relation d'instanciation peut être représentée
    - Utiliser le stéréotype <<instance de>>



# Exemple 1

24

Diagramme d'objet

Diagramme de classe

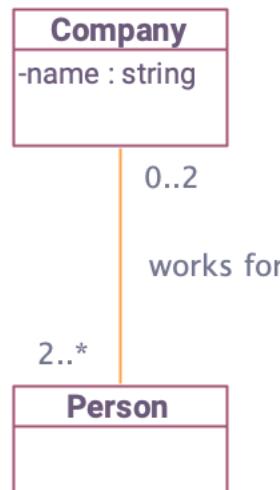
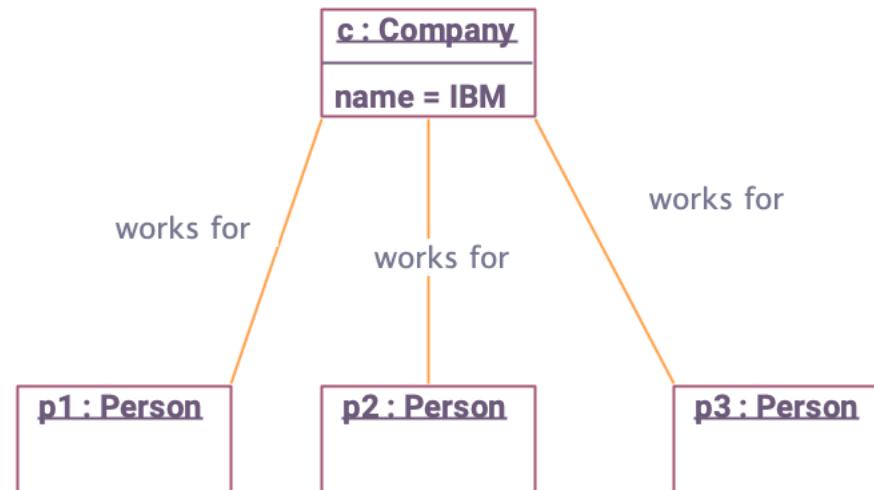


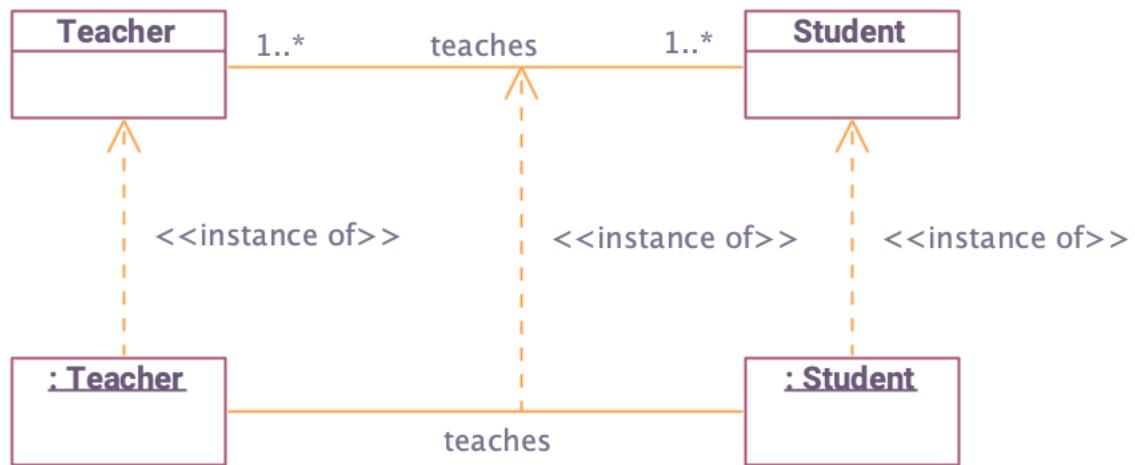
Diagramme d'objet



# Exemple 2

25

Diagramme d'objet



TI615M  
Analyse et Conception Orientée Objet avec

# UML

Dr. Lilia SFAXI

## Diagrammes d'interaction

4



# Diagrammes d'interaction

## Présentation

2

- Diagrammes dynamiques
- Aspect commun :
  - **Messages** : permettent la communication entre deux entités (objets, acteurs, sous-systèmes...)
- Plusieurs schémas :
  - Diagramme de séquence
  - Diagramme de communication
  - Diagramme global d'interaction
  - Diagramme de temps



# Plan

1 Diagramme de séquence

2 Diagramme de communication



# Plan

1 Diagramme de séquence

2 Diagramme de communication



# Diagrammes de séquences

## Diagrammes de séquences

- Le diagramme d'interaction le plus utilisé
- Représentation temporelle des messages échangés entre les objets
  - Séquencement des messages
- Représentation d'un scénario unique
  - Possibilité de combiner plusieurs scénarios
- Le temps s'écoule de haut en bas



# Contenu

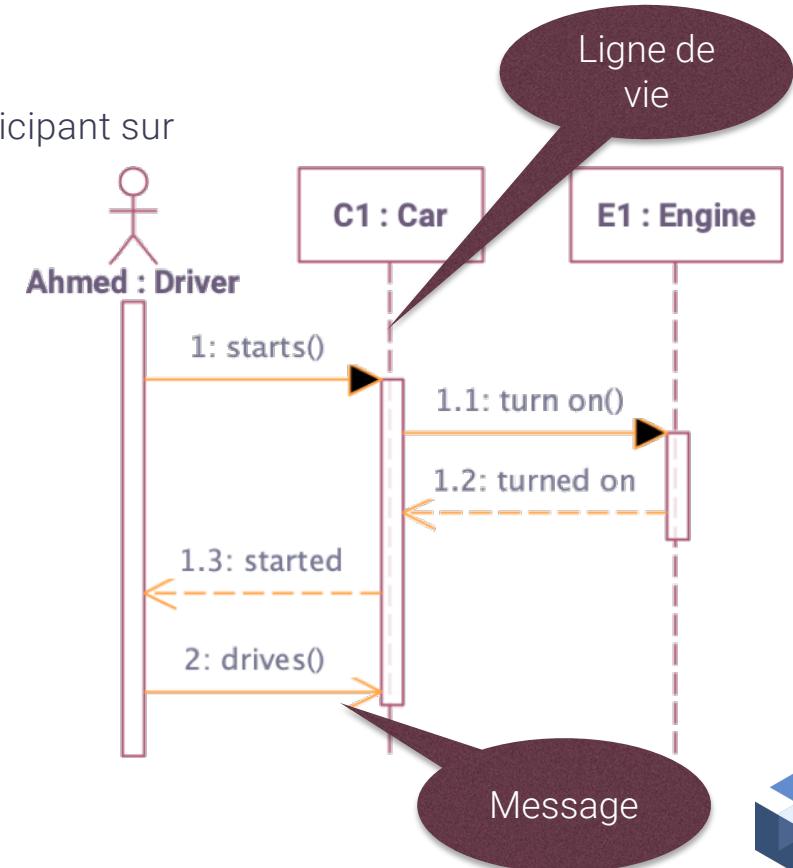
## Diagrammes de séquences

- Ligne de vie

- Un rôle dans une interaction qui représente un participant sur une période de temps
- Représenté par un rectangle et une ligne pointillée
- Étiquette :
  - {Objet} : {Classe}

- Messages

- Communication entre les lignes de vie
- Peut être :
  - Un signal
  - Une opération
  - La création ou la destruction d'une instance

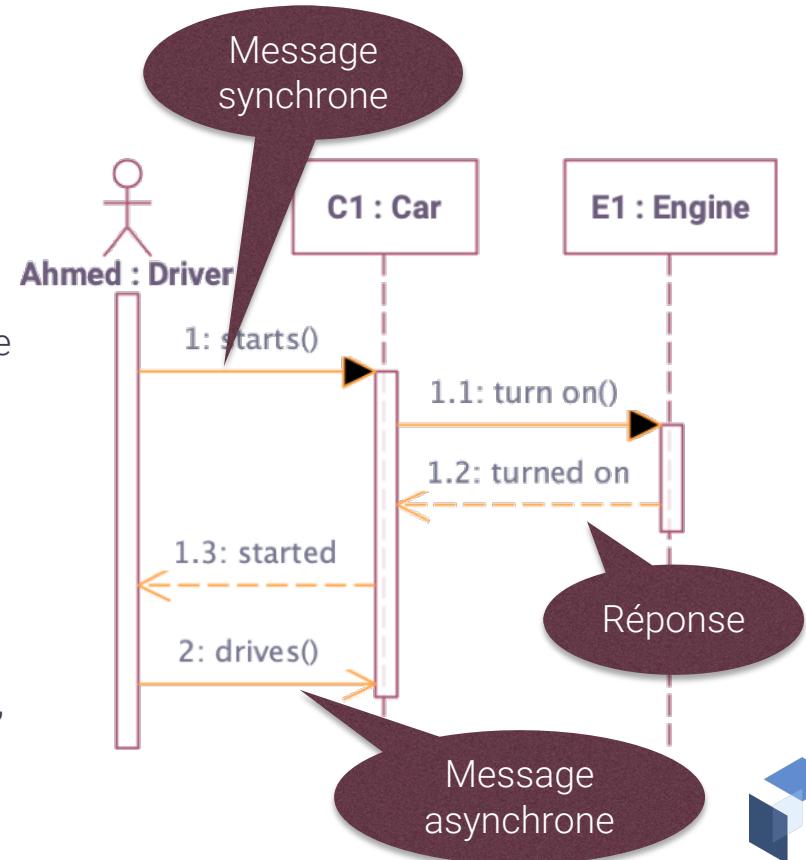


# Messages (1)

7

## Diagrammes de séquences

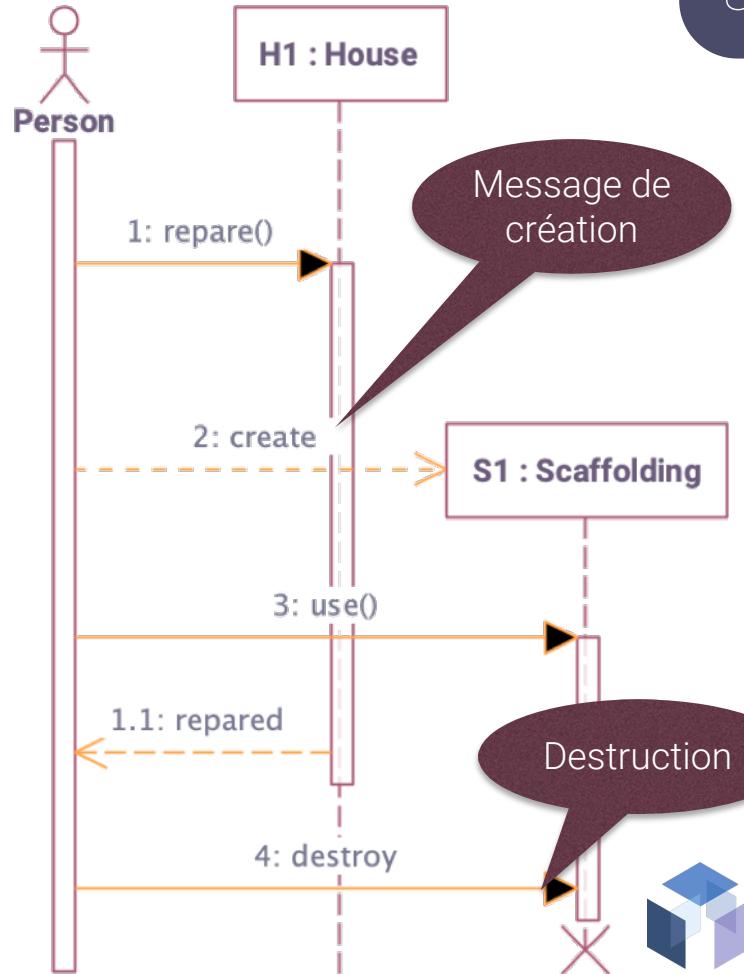
- Message asynchrone
  - N'attend pas de réponse
  - Ne bloque pas l'émetteur
  - **Exemple** : signal (interruption, événement)
  - **Représentation** : flèche pleine avec extrémité ouverte
- Message synchrone
  - Émetteur bloqué jusqu'à ce que le récepteur envoie une réponse
  - **Exemple** : appel d'une opération
  - **Représentation** : Flèche pleine avec une tête remplie, suivie d'une flèche pointillée



# Messages (2)

## Diagrammes de séquences

- Création d'une instance
  - Instantiation d'un objet qui n'existe pas
  - **Représentation** : flèche qui pointe vers la tête d'une ligne de vie
- Destruction de l'instance
  - Destruction d'un objet qui n'existera plus
  - N'est pas nécessairement déclenchée par un message
  - **Représentation** : une croix au bout d'une ligne de vie

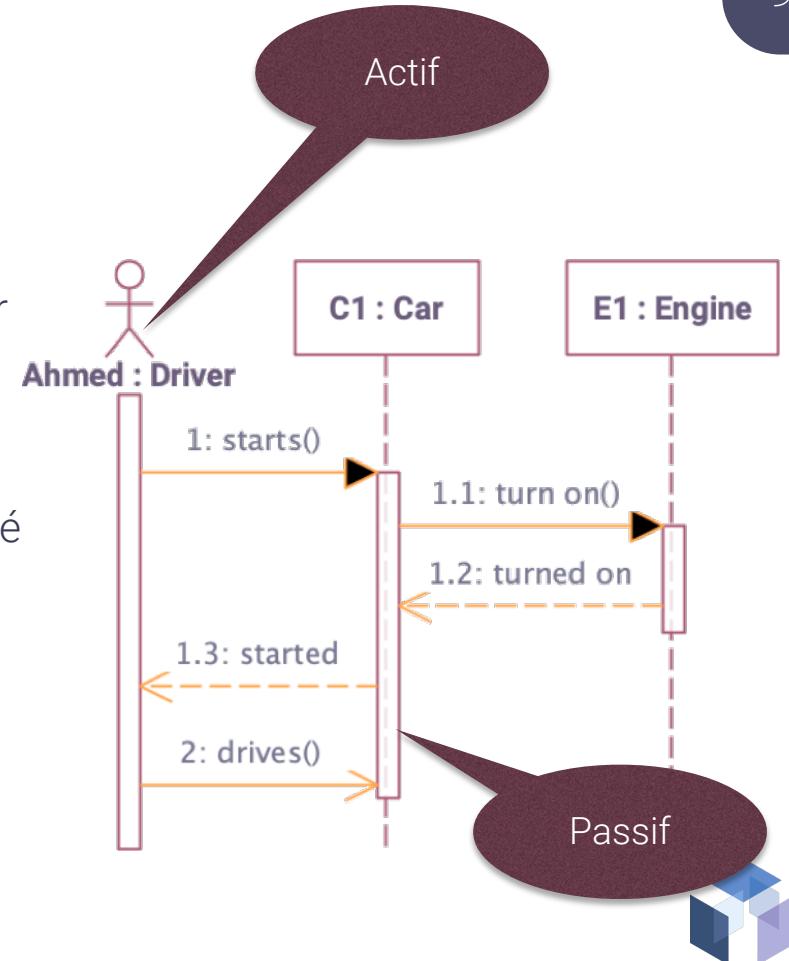


# Objets actifs et passifs

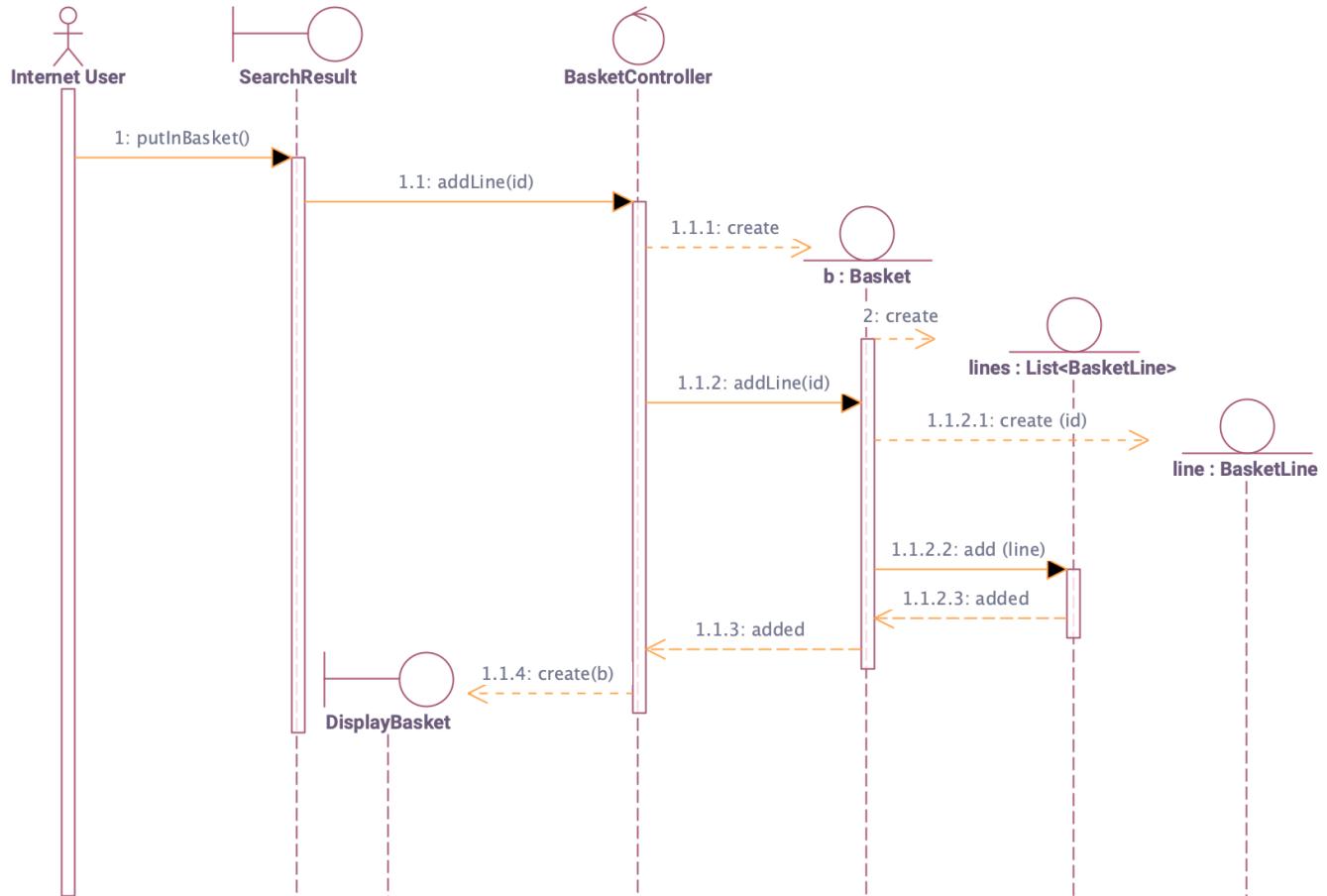
9

## Diagrammes de séquences

- Objet actif
  - Détient la racine d'une pile d'exécution
  - Possède son propre fil de contrôle déclenché par un événement qui s'exécute en parallèle avec d'autres objets actifs
  - **Représentation** : une double ligne de chaque côté de son symbole de tête
- Objet passif
  - Appelé par un objet actif
  - Reçoit le contrôle uniquement lorsqu'on l'appelle et le rend à son retour



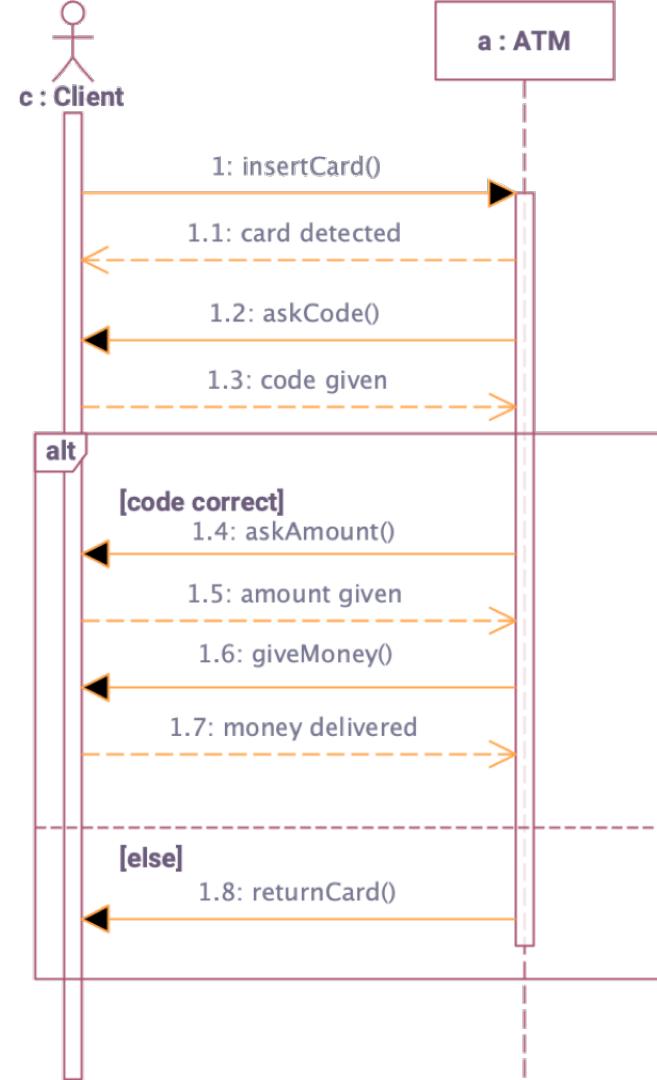
# Exemple de diagramme de séquence



# Structures de contrôle (1)

Diagrammes de séquences

- Opérateur alternatif (*alt*)
  - Opérateur conditionnel
    - Equivalent à une exécution à choix multiple (*switch*)
  - Peut contenir plusieurs opérandes, chacune ayant une garde
  - En cas d'absence de la condition de garde, celle-ci est considérée comme une condition *vraie*
  - La condition *else* est vraie si aucune autre condition dans l'alternative n'est vraie

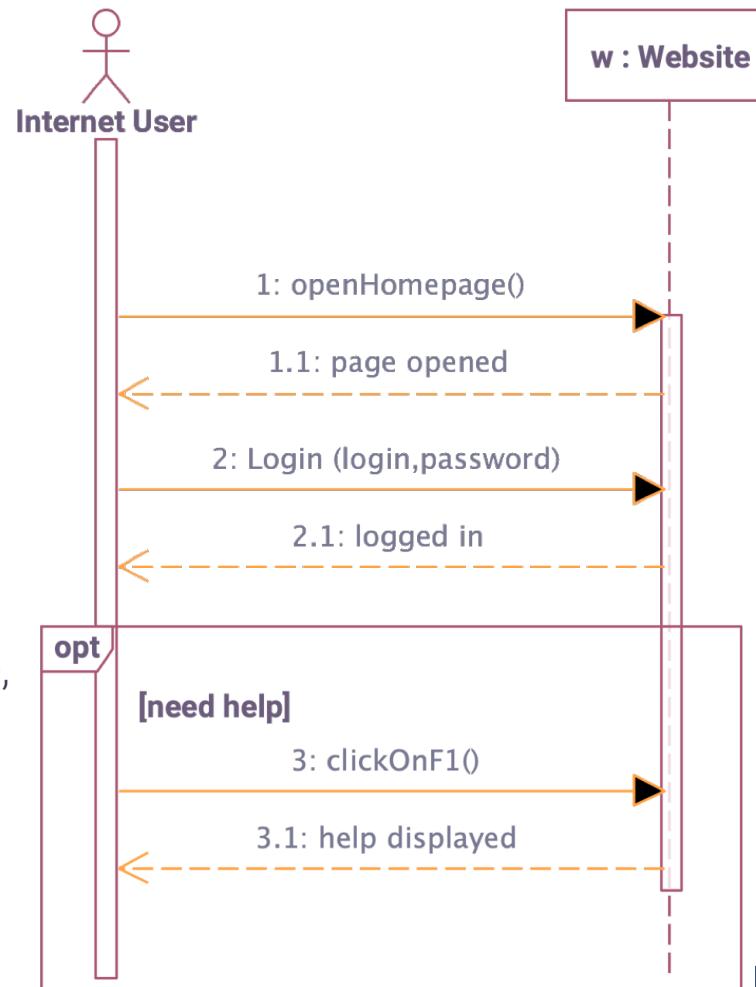


# Structures de contrôle (2)

Diagrammes de séquences

12

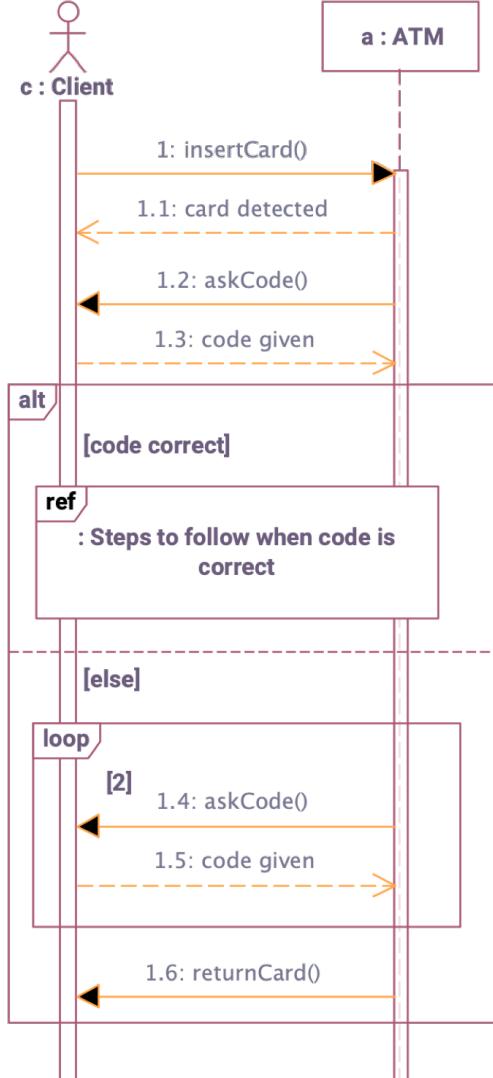
- Opérateur optionnel (*opt*)
  - Représente un comportement qui peut avoir lieu (ou non).
  - Equivalent d'un *alt* avec une seule branche, et rien d'autre.



# Opérateur de boucle

Diagrammes de séquences

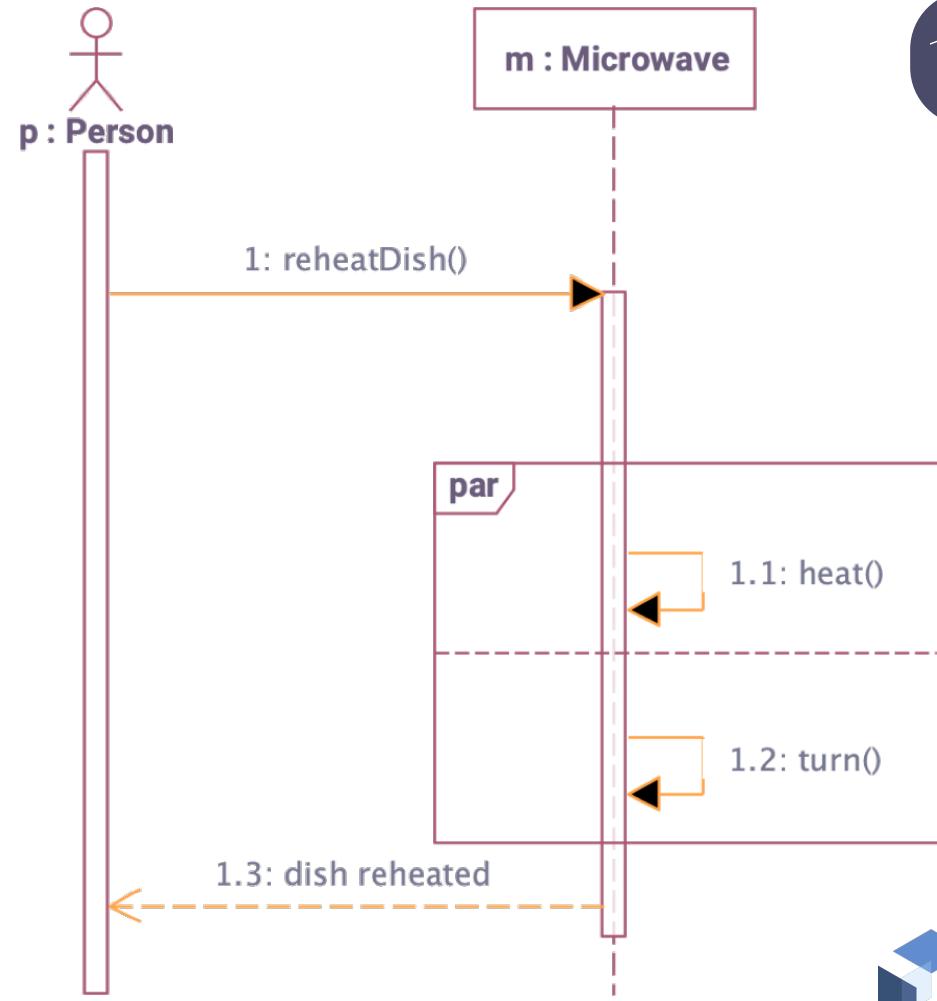
- Opérateur de boucle (*loop*)
  - Équivalent d'un *pour*
  - Décrit les interactions qui s'exécutent dans une boucle de code
  - La condition (garde) représente le nombre de répétitions (min et max) ou une condition booléenne à respecter



# Opérateur parallèle

Diagrammes de séquences

- Opérateur parallèle (**par**)
  - A au moins deux sous-fragments exécutés simultanément
  - Simule une exécution parallèle



# Plan

1 Diagramme de séquence

2 Diagramme de communication



# Diagrammes de communication

16

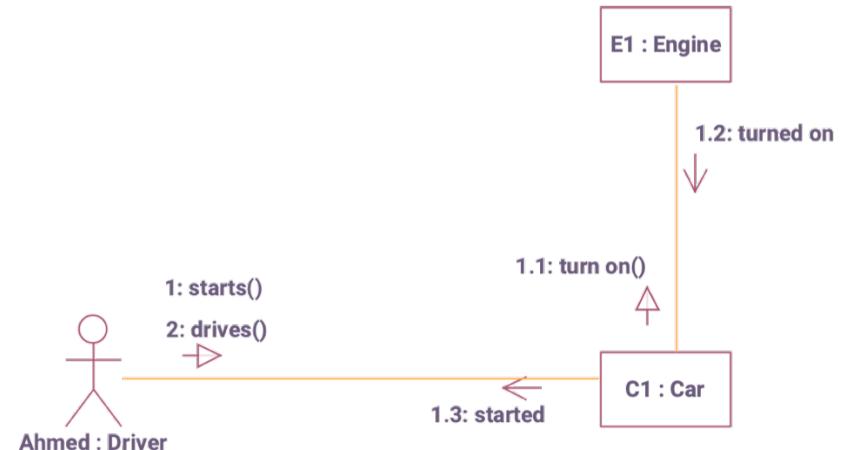
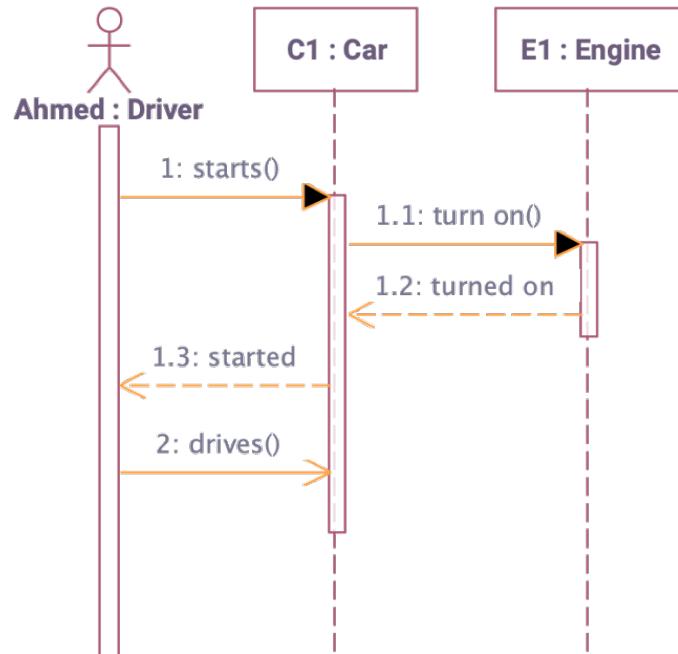
## Diagrammes de communication

- Aussi appelé diagramme de *collaboration*, avant UML2
- Montre les interactions entre les objets
- Insiste sur la structure spatiale pour se concentrer sur la collaboration d'un groupe d'objets
  - Messages : Liens entre les objets
  - L'heure : implicitement représentée par la numérotation du message



# Exemple de diagramme de communication

17



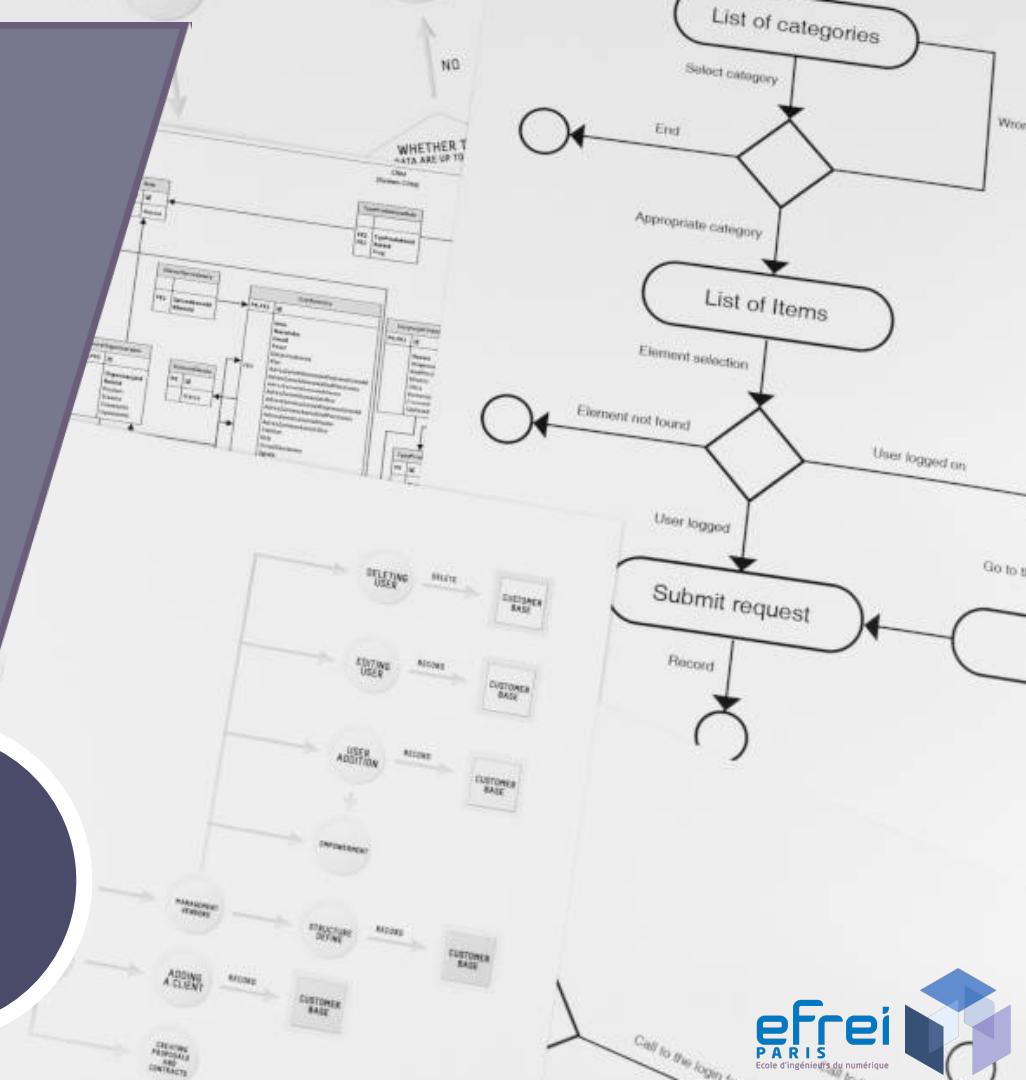
TI615M  
Analyse et Conception Orientée Objet avec

# UML

Dr. Lilia SFAXI

## Patrons de Conception

5



# Plan



- 1 Introduction aux patrons de conception
- 2 Patrons de création
- 3 Patrons structurels
- 4 Patrons de comportement



# Plan

1

Introduction aux patrons de conception

2

Patrons de création

3

Patrons structurels

4

Patrons de comportement



# Définition et utilité

Introduction aux patrons de conception

4

- En génie logiciel, un patron de conception est une solution générale et réproductible à un problème courant de conception de logiciels
- Il ne s'agit pas d'une conception finie qui peut être transformée directement en code, mais d'une description ou d'un modèle pour résoudre un problème qui peut être utilisé dans de nombreuses situations différentes



# Utilisation

## Introduction aux patrons de conception

- Les patrons de conception :
  - Fournit des solutions générales, documentées dans un format qui ne nécessite pas de précisions liées à un problème particulier
  - Peut accélérer le processus de développement en fournissant des paradigmes de développement testés et éprouvés
  - Aide à bénéficier de l'expérience de développeurs chevronnés
  - Prévenir les questions subtiles qui peuvent causer des problèmes majeurs
  - Améliorer la lisibilité du code pour les codeurs et les architectes



# Éléments essentiels

Introduction aux patrons de conception

6

- Un patron comporte quatre éléments essentiels :
  - Le nom du patron que nous utilisons pour décrire un problème de conception
  - Le problème qui décrit quand appliquer le modèle
  - La solution qui décrit les éléments qui composent le dessin
  - Les conséquences qui sont les résultats et les compromis de l'application du patron



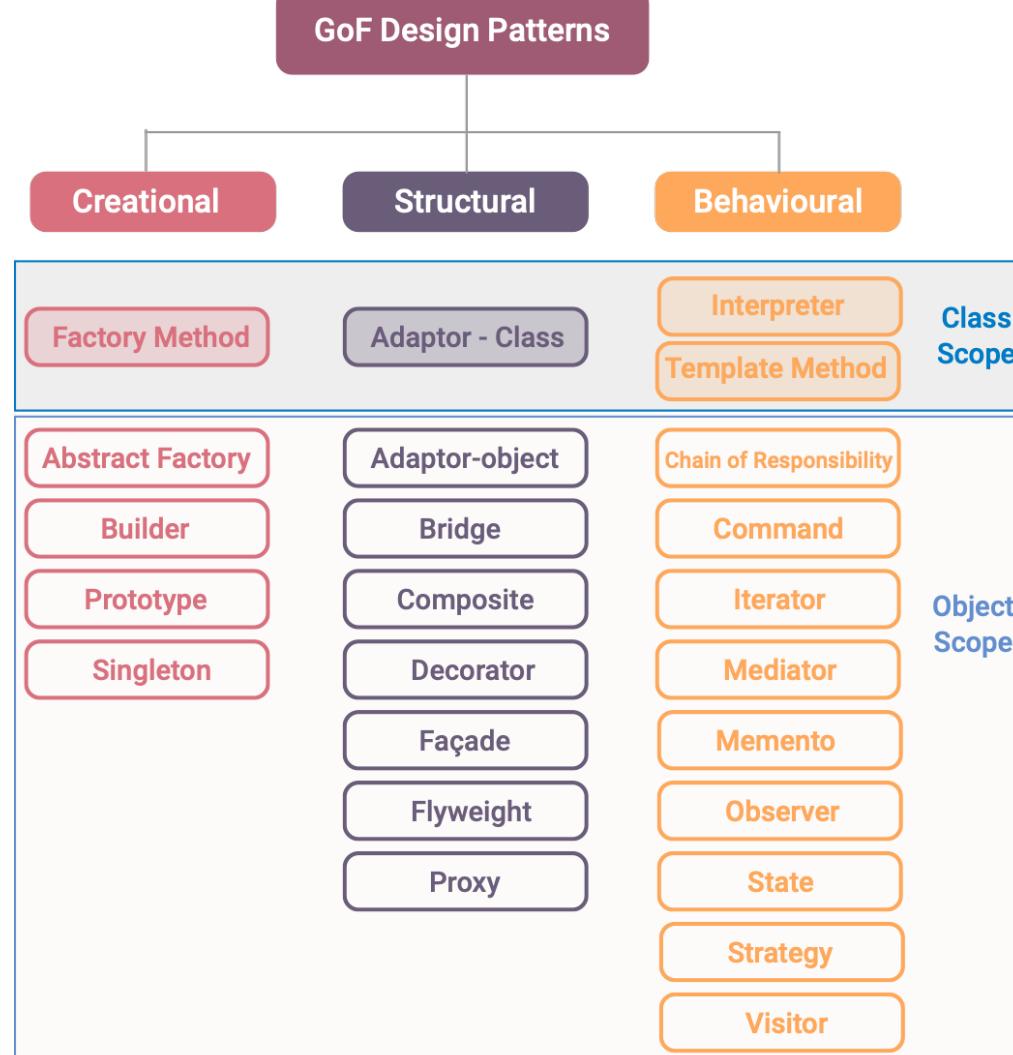
# Patrons de conception GoF

## Introduction aux patrons de conception

7

- La Bande des quatre (Gang of Four) sont les quatre auteurs du livre "*Design Patterns : Elements of Reusable Object-Oriented Software*".
- Définition de 23 patrons de conception pour les problèmes de conception récurrents, appelés patrons de conception GoF
- Classement par objectif :
  - **Structural** : concerne la composition des classes et des objets
  - **Comportemental** : caractérise l'interaction et la responsabilité des objets et des classes
  - **Créationnel** : concerne le processus de création d'objets et de classes
- ... et par portée :
  - **Portée de la classe** : relation entre les classes et les sous-classes, définie statiquement
  - **Portée de l'objet** : relations entre les objets, dynamique





# Plan



- 1 Introduction aux patrons de conception
- 2 Patrons de création
- 3 Patrons structurels
- 4 Patrons de comportement



# Définition

10

## Patrons de création

- Les patrons de création permettent d'abstraire le processus d'instanciation.
- Ils contribuent à rendre un système indépendant de la manière dont ses objets sont créés, composés et représentés
  - Les patrons de création de **classes** utilisent l'héritage pour faire varier la classe qui est instanciée.
  - Les patrons de création d'**objets** délèguent l'instanciation à un autre objet.
- Exemples :
  - Factory
  - Singleton
  - Builder
  - Prototype

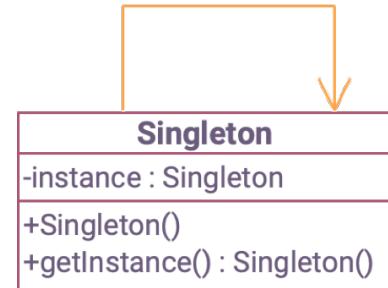


# Singleton

## Patrons de création

- Garantit qu'une seule instance d'une classe est créée
- Fournit un point d'accès global à l'objet

```
class Singleton{  
    private static Singleton instance;  
  
    private Singleton(){ ... }  
  
    public static synchronized Singleton getInstance(){  
        if (instance == null){  
            instance = new Singleton();  
        }  
        return instance;  
    }  
    public void doSomething(){ ... }  
}
```



# Singleton : Utilisation

12

## Patrons de création

- Quand l'utiliser :
  - Quand il faut s'assurer qu'une seule instance d'une classe est créée
  - Lorsque l'instance doit être disponible à travers tout le code
  - Dans les environnements multi-thread, lorsque plusieurs threads doivent accéder aux mêmes ressources par le biais d'un même objet singleton.
- Usage commun
  - Classes Logger
  - Classes de configuration
  - Accès aux ressources en mode partagé
  - D'autres patrons de conception mis en œuvre en tant que Singletons :
    - Factories et Abstract Factories, Builder, Prototype



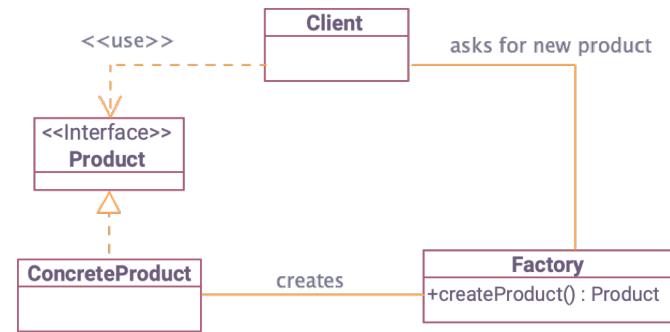
# Factory

13

Patrons de création

- Crée des objets sans exposer la logique d'instanciation au client
- Fait référence à l'objet nouvellement créé par le biais d'une interface commune.

```
public class ProductFactory {  
    public Product createProduct (String productID){  
        if (productID == "ID1"){  
            return new OneProduct();  
        }  
        if (productID == "ID2"){  
            return new AnotherProduct();  
        }  
        ...  
        return null;  
    }  
    ...  
}
```



# Factory : Utilisation

14

## Patrons de création

- Quand l'utiliser :
  - Lorsqu'un framework délègue à l'usine la création d'objets issus d'une superclasse commune
  - Lorsque nous avons besoin de flexibilité pour ajouter de nouveaux types d'objets qui doivent être créés par la classe
- Usage commun
  - Factories fournissant un parseur xml :
    - javax.xml.parsers.DocumentBuilderFactory
    - javax.xml.parsers.SAXParserFactory
  - java.net.URLConnexion



# Plan



- 1 Introduction aux patrons de conception
- 2 Patrons de création
- 3 Patrons structurels
- 4 Patrons de comportement



# Définition

## Patrons structurels

16

- Les patrons structurels s'intéressent à la façon dont les classes et les objets sont composés pour former des structures plus grandes.
  - Les modèles de **classes** structurelles utilisent l'héritage pour composer des interfaces ou des implémentations.
  - Les modèles d'**objets** structurels décrivent des façons de composer des objets pour réaliser de nouvelles fonctionnalités. La souplesse supplémentaire de la composition des objets provient de la possibilité de modifier la composition à l'exécution, ce qui est impossible avec la composition de classes statiques.
- Exemples :
  - Adapter
  - Proxy
  - Bridge
  - Composite

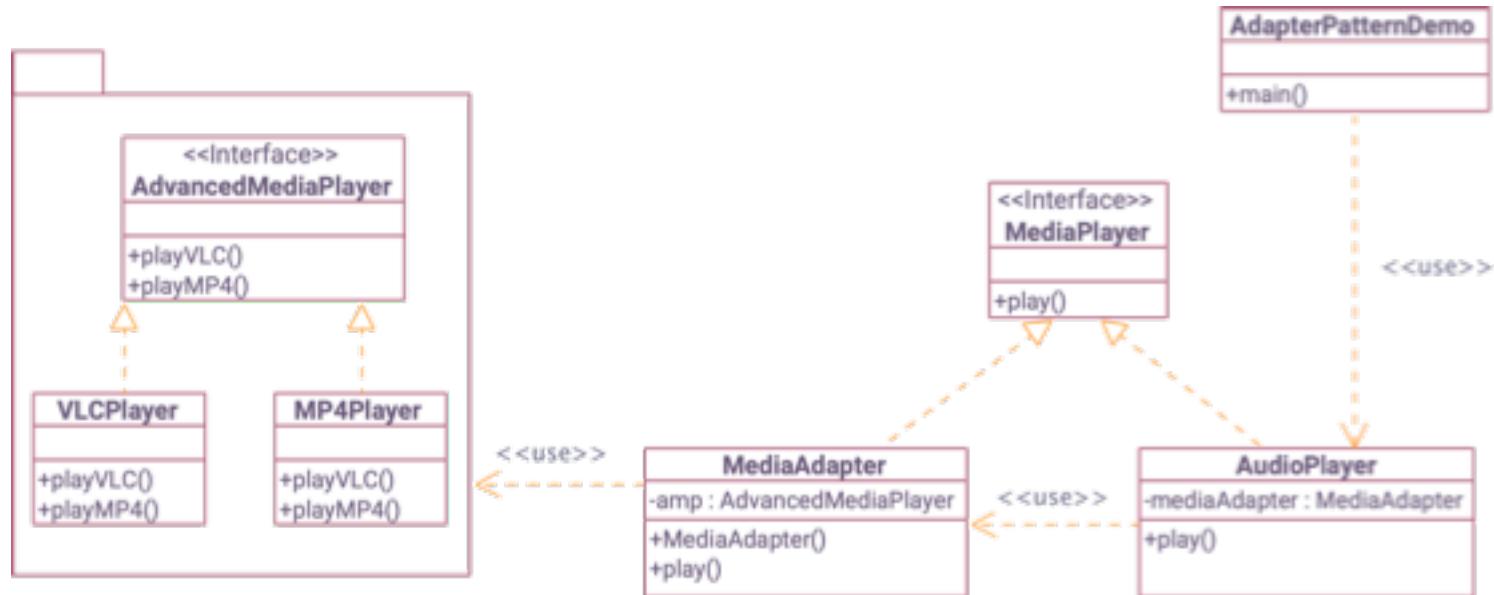


# Adaptateur

17

Patrons structurels

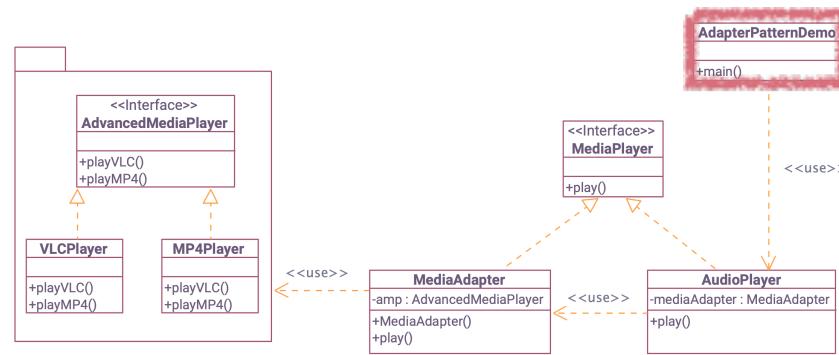
- Convertit l'interface d'une classe en une autre interface que les clients attendent
- Permet aux classes de travailler ensemble, ce qui n'est normalement pas le cas, en raison d'interfaces incompatibles



# Adaptateur

## Patrons structurels

18

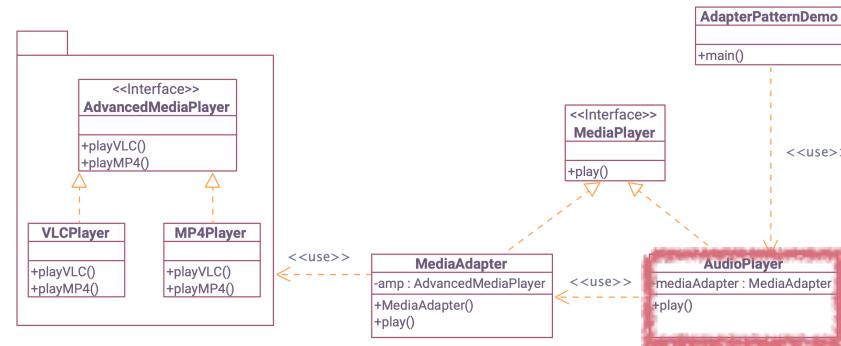


```
public class AdapterPatternDemo {  
    public static void main(String[] args){  
        AudioPlayer audioPlayer = new AudioPlayer();  
  
        audioPlayer.play("mp3", "beyond the horizon.mp3");  
        audioPlayer.play("mp4", "alone.mp4");  
        audioPlayer.play("vlc", "far far away.mp3");  
        audioPlayer.play("avi", "mind me.mp3");  
    }  
}
```



# Adaptateur

## Patrons structurels



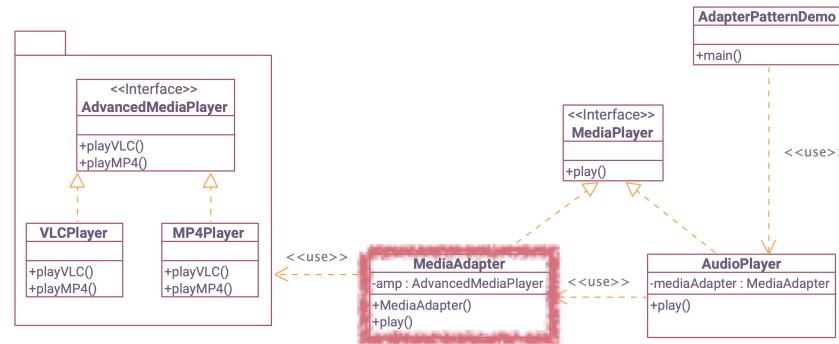
```
public class AudioPlayer implements MediaPlayer {  
    MediaAdapter mediaAdapter;  
  
    @Override  
    public void play(String audioType, String filename){  
        if (audioType.equalsIgnoreCase("mp3")){  
            System.out.println("Playing MP3 file: "+filename);  
        }  
        else if (audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){  
            mediaAdapter = new MediaAdapter(audioType);  
            mediaAdapter.play(audioType, filename);  
        }  
        else{  
            System.out.println("Invalid Media format: "+audioType);  
        }  
    }  
}
```



# Adaptateur

## Patrons structurels

20



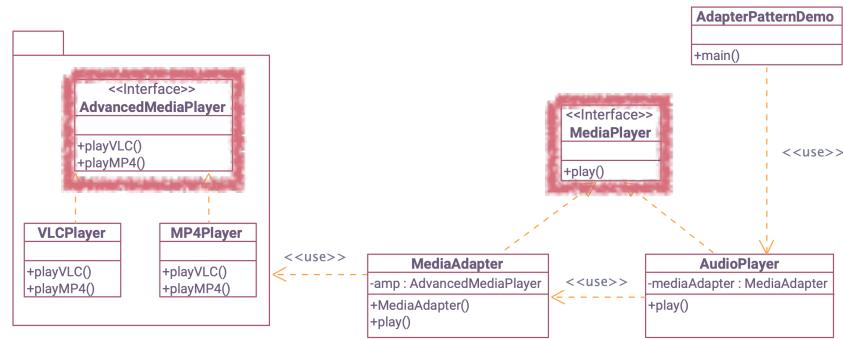
```
public class MediaAdapter implements MediaPlayer {  
    AdvancedMediaPlayer amp;  
    public MediaAdapter(String audioType){  
        if (audioType.equalsIgnoreCase("vlc")){  
            amp = new VLCPlayer();  
        } else if (audioType.equalsIgnoreCase("mp4")){  
            amp = new MP4Player();  
        }  
    }  
  
    @Override  
    public void play(String audioType, String filename){  
        if (audioType.equalsIgnoreCase("vlc")){  
            amp.playVLC(filename);  
        } else if (audioType.equalsIgnoreCase("mp4")){  
            amp.playMP4(filename);  
        }  
    }  
}
```



# Adaptateur

Patrons structurels

21



```
public interface MediaPlayer {  
    public void play(String audioType, String filename);  
}
```

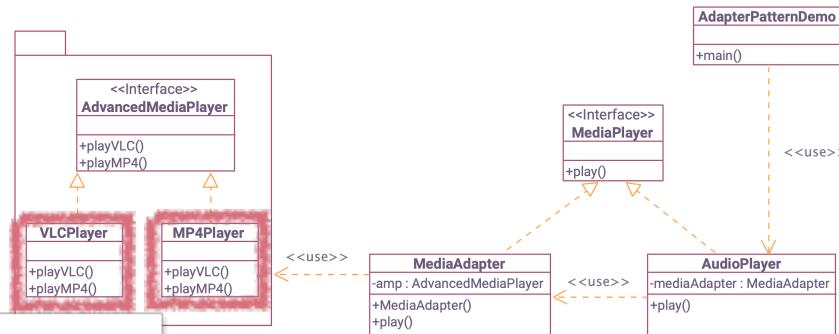
```
public interface AdvancedMediaPlayer {  
    public void playVLC(String filename);  
    public void playMP4(String filename);  
}
```



# Adaptateur

## Patrons structurels

22



```
public class VLCPlayer implements AdvancedMediaPlayer{
    @Override
    public void playVLC(String filename){
        System.out.println("Playing VLC file: "+filename);
    }
    @Override
    public void playMP4(String filename){
        // Do Nothing
    }
}
```

```
public class MP4Player implements AdvancedMediaPlayer{
    @Override
    public void playVLC(String filename){
        // Do Nothing
    }
    @Override
    public void playMP4(String filename){
        System.out.println("Playing MP4 file: "+filename);
    }
}
```

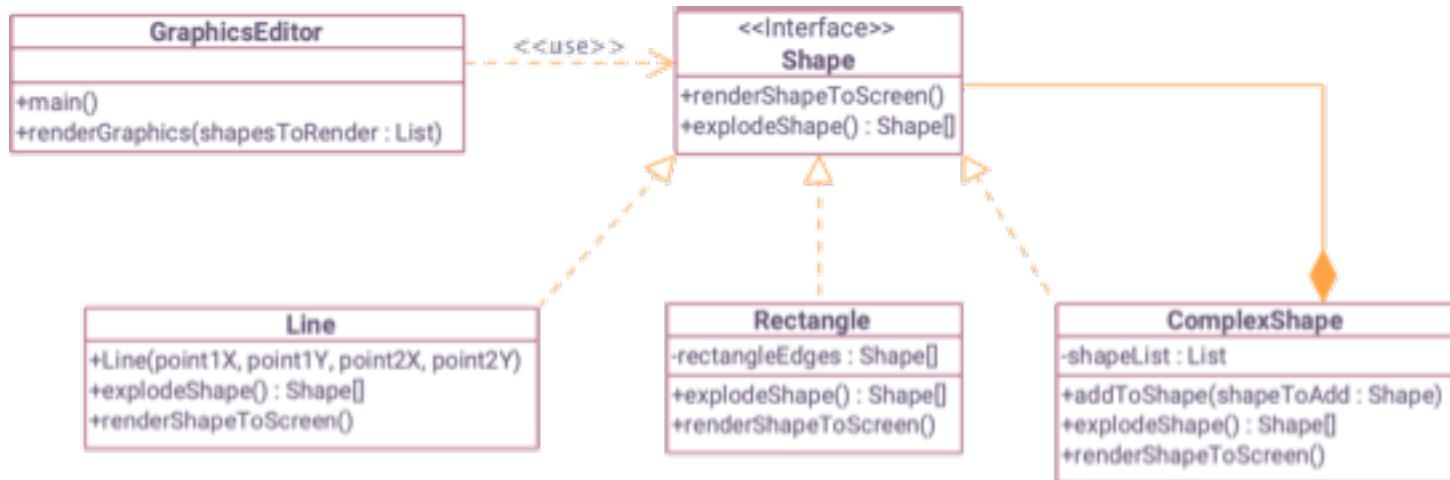


# Composite

23

## Patrons structurels

- Compose des objets en arborescence pour représenter des hiérarchies partielles ou complètes.
- Permet aux clients de traiter des objets individuels et des compositions d'objets de manière uniforme.

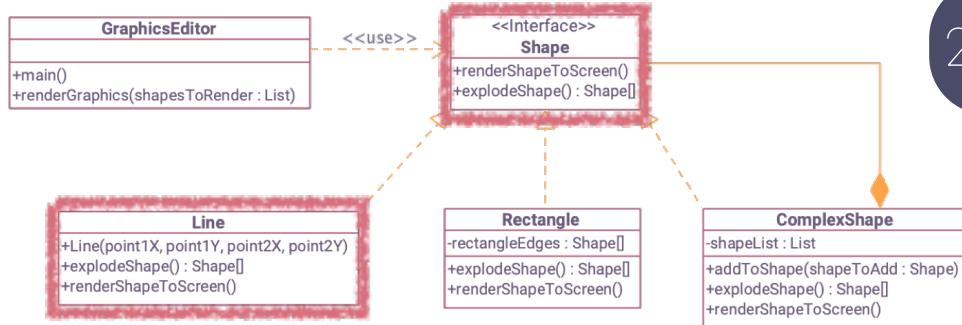


# Composite

Patrons structurels

24

```
public class Line implements Shape {  
  
    public Line(int pointX, int pointY,  
               int point2X, int point2Y){  
    }  
  
    @Override  
    public Shape[] explodeShape(){  
        Shape[] shapeParts = this;  
        return shapeParts;  
    }  
  
    public void renderShapeToScreen(){  
        //render shape to screen  
    }  
}
```



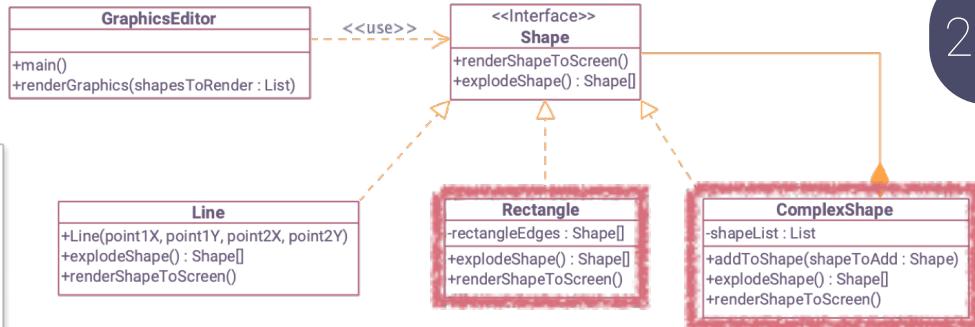
```
public interface Shape {  
  
    public void renderShapeToScreen();  
    public Shape[] explodeShape();  
}
```



# Composite

Patrons structurels

```
public class ComplexShape implements Shape {  
    List shapeList = new ArrayList();  
    public void addToShape(Shape shapeToAdd){  
        shapeList.add(shapeToAdd);  
    }  
    public Shape[] explodeShape(){  
        return (Shape[]) shapeList.toArray();  
    }  
    public void renderShapeToScreen(){  
        for (Shape s: shapeList){  
            s.renderShapeToScreen();  
        }  
    }  
}
```



```
public class Rectangle implements Shape {  
    Shape[] rectangleEdges = {...}  
    @Override  
    public Shape[] explodeShape(){  
        return rectangleEdges;  
    }  
    public void renderShapeToScreen(){  
        for (Shape s : rectangleEdges){  
            s.renderShapeToScreen();  
        }  
    }  
}
```

# Plan



- 1 Introduction aux patrons de conception
- 2 Patrons de création
- 3 Patrons structurels
- 4 Patrons de comportement



# Définition

27

## Patrons comportementaux

- Les patrons comportementaux concernent les algorithmes et l'attribution des responsabilités entre les objets.
  - Les modèles comportementaux de **classe** utilisent l'héritage pour répartir le comportement entre les classes.
  - Les modèles d'**objets** comportementaux utilisent la composition plutôt que l'héritage. Certains décrivent comment un groupe d'objets pairs coopèrent pour effectuer une tâche qu'aucun objet ne peut réaliser seul.
- Exemples :
  - Commande
  - Iterator
  - Observer
  - Strategy

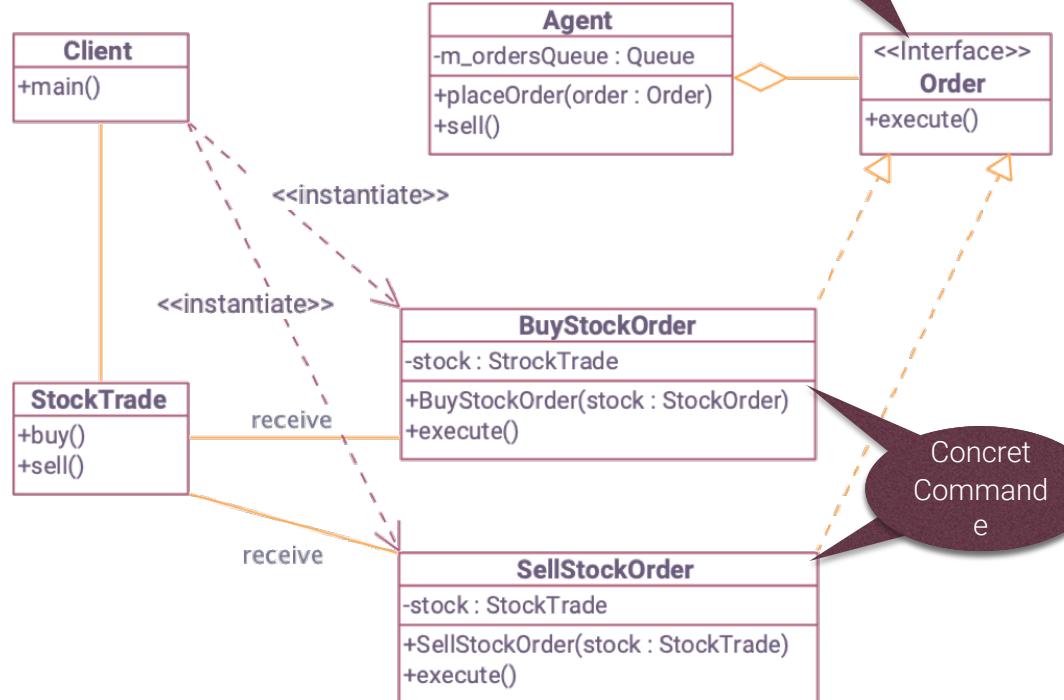


# Commande

## Patrons comportementaux

28

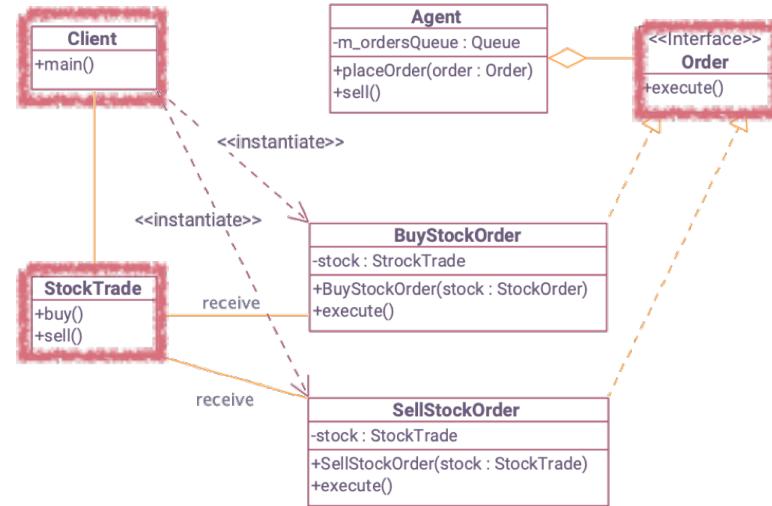
- Encapsule une demande dans un objet
- Permet de paramétriser les clients ayant des demandes différentes
- Permet d'enregistrer la demande dans une file d'attente



# Commande

## Patrons comportementaux

```
public interface Order {  
    void execute();  
}  
  
class StockTrade {  
    public void buy(){  
        System.out.println("You want to buy stocks");  
    }  
    public void sell(){  
        System.out.println("You want to sell stocks");  
    }  
}
```



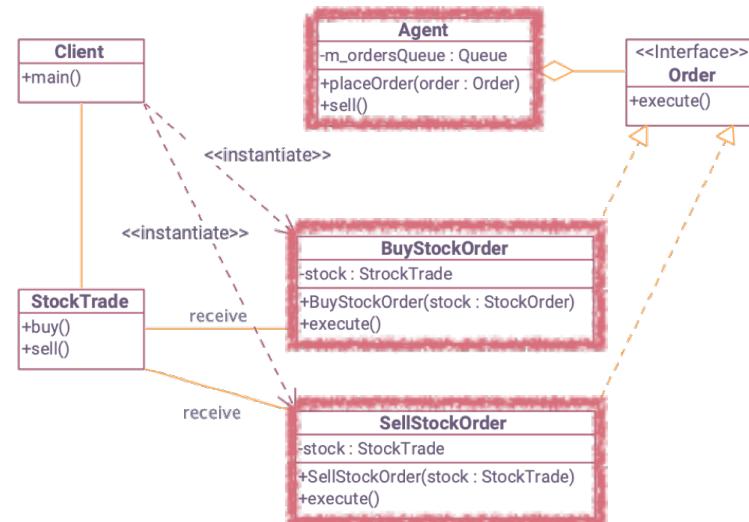
```
public class Client {  
    public static void main(String[] args){  
        StockTrade stock = new StockTrade();  
        BuyStockOrder bso = new BuyStockOrder(stock);  
        SellStockOrder sso = new SellStockOrder(stock);  
        Agent agent = new Agent();  
        agent.placeOrder(bso);  
        agent.placeOrder(sso);  
    }  
}
```



# Commande

## Patrons comportementaux

```
class BuyStockOrder implements Order {  
    private StockTrade stock;  
    public BuyStockOrder (StockTrade st){  
        stock = st;  
    }  
    public void execute(){  
        stock.buy();  
    }  
  
class SellStockOrder implements Order {  
    private StockTrade stock;  
    public SellStockOrder (StockTrade st){  
        stock = st;  
    }  
    public void execute(){  
        stock.sell();  
    }  
}
```



```
class Agent {  
    private List m_ordersQueue = new ArrayList();  
  
    public Agent(){}
  
  
    void placeOrder(Order order){  
        m_ordersQueue.addLast(order);  
        order.execute(m_ordersQueue.getFirstAndRemove())  
    }
}
```

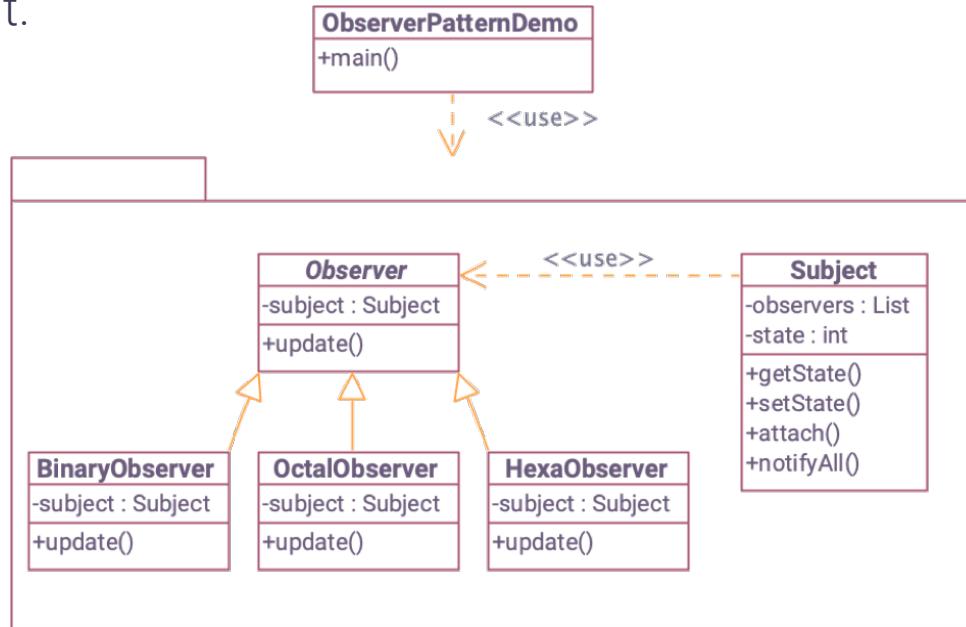


# Observer

31

Patrons comportementaux

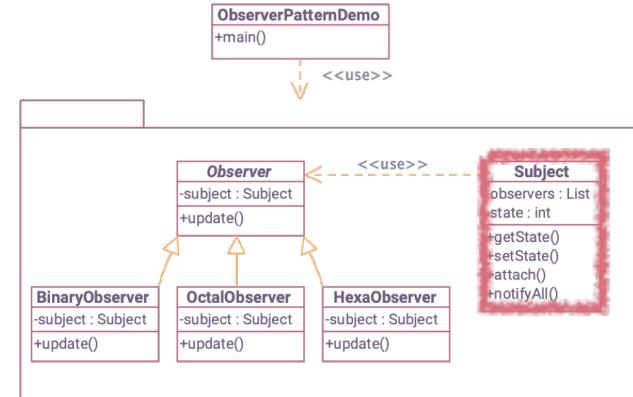
- Définit une dépendance de un à plusieurs objets de sorte que lorsqu'un objet change d'état, toutes ses dépendances sont notifiées et mises à jour automatiquement.



# Observer

## Patrons comportementaux

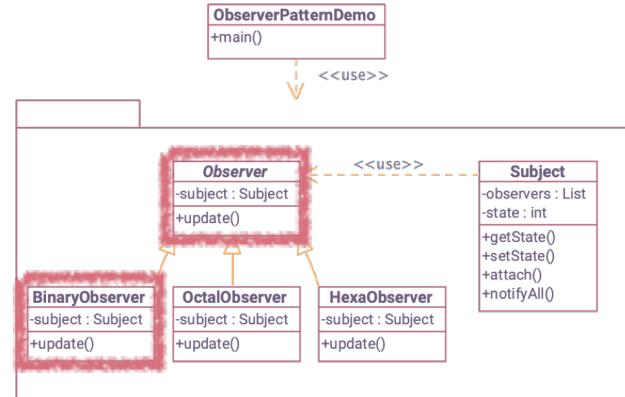
```
public class Subject {  
  
    private List<Observer> observers = new ArrayList<Observer>();  
    private int state;  
  
    public int getState() {  
        return state;  
    }  
  
    public void setState(int state) {  
        this.state = state;  
        notifyAllObservers();  
    }  
  
    public void attach(Observer observer){  
        observers.add(observer);  
    }  
  
    public void notifyAllObservers(){  
        for (Observer observer : observers) {  
            observer.update();  
        }  
    }  
}
```



# Observer

## Patrons comportementaux

```
public abstract class Observer {  
  
    protected Subject subject;  
    public abstract void update();  
  
}  
  
  
public class BinaryObserver extends Observer{  
  
    public BinaryObserver(Subject subject){  
        this.subject = subject;  
        this.subject.attach(this);  
    }  
  
    @Override  
    public void update() {  
        System.out.println("Binary String: " +  
            Integer.toBinaryString(subject.getState()));  
    }  
}
```



# Observer

Patrons comportementaux

```
public class ObserverPatternDemo {  
    public static void main(String[] args) {  
        Subject subject = new Subject();  
  
        new HexaObserver(subject);  
        new OctalObserver(subject);  
        new BinaryObserver(subject);  
  
        System.out.println("First state change: 15");  
        subject.setState(15);  
        System.out.println("Second state change: 10");  
        subject.setState(10);  
    }  
}
```

