

# Chapitre 5: Conception des Objets

3<sup>ème</sup> Année Ingénieur

Semestre: S6

Ecole Supérieure en Informatique



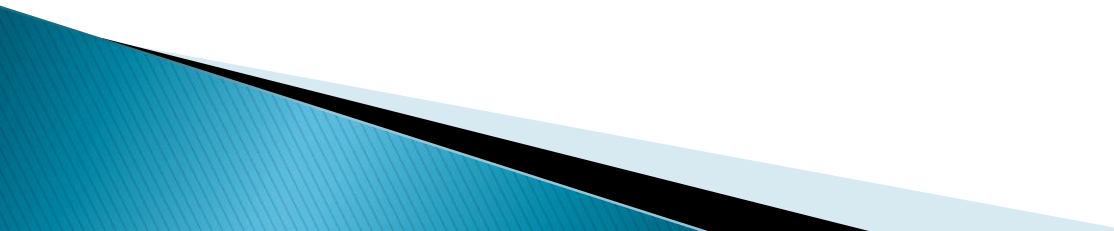
# PLAN

## ▶ I. Introduction

## ▶ II. Méthode UP/XP

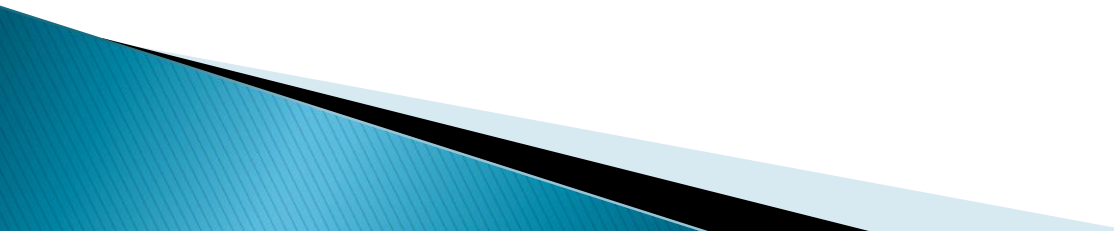
- Identification des besoins
  - Diagramme de cas d'utilisation
  - Diagramme de séquences
  - Maquette IHM
- Phase d'analyse
  - Analyse du domaine : modèle du domaine
  - Diagramme de classes participantes
  - Diagramme de navigation
- Phase de conception
  - Diagramme de classe de conception

## ▶ III. Introduction aux Patterns de Conception

- Notion de Pattern
  - Nature des patterns
  - Hiérarchie des objets réutilisables
  - Catégories des Patterns de conception
- 

# PLAN (Suite)

## IV. Conception de systèmes d'information orientés Web (ou Applications Web) :

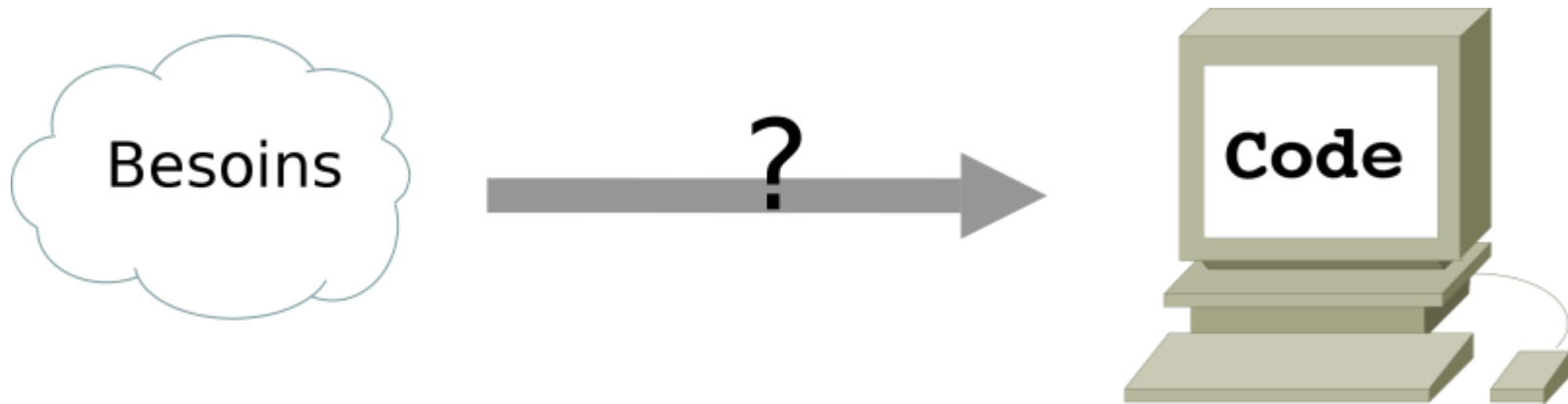
- Qualité de systèmes d'information orientés web
  - Buts de conception
  - Pyramide de Conception :
  - Conception de l'interface
  - Conception de l'aspect esthétique
  - Conception du contenu
  - Conception de la navigation
  - Conception de l'architecture
  - Conception de composant
- 

# Références

1. Roger Pressman, SOFTWARE ENGINEERING: A PRACTITIONER'S APPROACH, EITH EDITION, Published by McGraw-Hill
2. Ian Sommerville SOFTWARE ENGINEERING, Ninth Edition, Addison Wisley
3. John W. Satzinger, Robert B. Jackson, Stephen D. Burd SYSTEMS ANALYSIS AND DESIGN IN A CHANGING WORLD, Sixth Edition, CENPAGE
4. Laurent AUDIBERT UML 2.0: de l'apprentissage à la pratique. Edt. Ellipse 2014
5. Pascal Roques. **UML - Modéliser un site e-commerce**. Les cahiers du programmeur. Eyrolles, 2002.
6. Debrauwer, L. 2007. Design Patterns: Les 23 modèles de conception : descriptions et solutions illustrées en UML 2 et Java: 3<sup>ème</sup> Édition ENI, 2013.
7. Lasater, C.G. 2006. Design Patterns: Wordware Publishing.
8. Gamma, E. 1995. Design patterns: elements of reusable object-oriented software:Addison-Wesley
9. Larman 2004 - Applying Uml And Patterns- An Introduction To Object-Oriented Analysis And Design And The RUP, 3<sup>ème</sup> Edition Prentice Hall
10. Steven John Metsker et William C. Wake. 2009 Les Design Patterns en Java : Les 23 modèles de conception fondamentaux. Edition Pearson
11. J. Coplien et D. Schmidt Pattern Languages of Program Design,, Addison-Wesley 1995

# I. Introduction

- ❑ La problématique que pose la mise en œuvre d'UML est simple : comment passer de l'expression des besoins au code de l'application ?



# I. Introduction (1)

Bien qu'UML ne soit pas une méthode, ses auteurs précisent néanmoins qu'une méthode basée sur l'utilisation UML doit être :

## ❑ Pilotée par les cas d'utilisation :


la principale qualité d'un logiciel étant son utilité, c'est-à-dire son adéquation avec les besoins des utilisateurs, toutes les étapes, de la spécification des besoins à la maintenance, doivent être guidées par les cas d'utilisation qui modélisent justement les besoins des utilisateurs ;

## ❑ Centrée sur l'architecture :

l'architecture est conçue pour satisfaire les besoins exprimés dans les cas d'utilisation, mais aussi pour prendre en compte les évolutions futures et les contraintes de réalisation. La mise en place d'une architecture adaptée conditionne le succès d'un développement. Il est important de la stabiliser le plus tôt possible ;

## ❑ Itérative et incrémentale :

l'ensemble du problème est décomposé en petites itérations, définies à partir des cas d'utilisation et de l'étude des risques. Les risques majeurs et les cas d'utilisation les plus importants sont traités en priorité. Le développement procède par des itérations qui conduisent à des livraisons incrémentales du système.



# I. Introduction (2)

❑ Nous allons présenter une méthode simple et générique qui se situe à mi-chemin entre UP (*Unified Process*), qui constitue un cadre général très complet de processus de développement, et XP (*eXtreme Programming*) qui est une approche minimaliste à la mode centrée sur le code.

❑ Cette méthode est issue de celle présentée par [5] dans son livre « *UML - Modéliser un site e-commerce* ». Elle a donc montré son efficacité dans la pratique et est :

- conduite par les cas d'utilisation, comme UP, mais bien plus simple ; relativement légère et restreinte, comme XP, mais sans négliger les activités de modélisation en analyse et conception ;
- fondée sur l'utilisation d'un sous-ensemble nécessaire et suffisant du langage UML (modéliser 80 % des problèmes en utilisant 20 % d'UML).

❑ Le fait de produire des diagrammes UML selon un ordre établi n'est en aucun cas une garantie de réussite.

❑ Une méthode ne sert qu'à canaliser et ordonner les étapes de la modélisation.

❑ La valeur n'est pas dans la méthode, mais dans les personnes qui la mettent en œuvre.

## II. Méthode UP/XP

### Identification des besoins

Diagramme de cas d'utilisation

Diagramme de séquences

Maquette IHM

### Phase d'analyse

**Analyse du domaine : modèle du domaine**

Diagramme de classes participantes

Diagramme de navigation

### Phase de conception

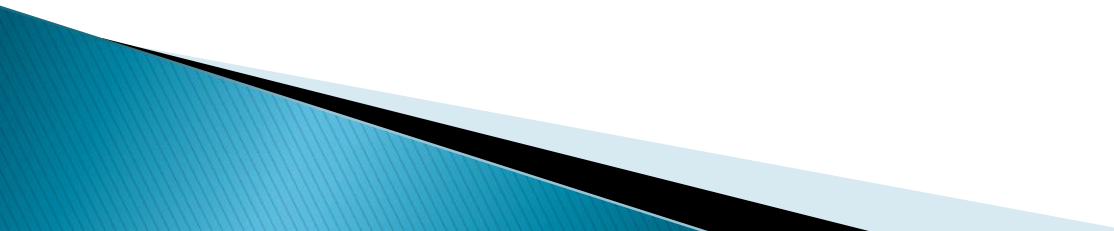
Diagramme de classe de conception

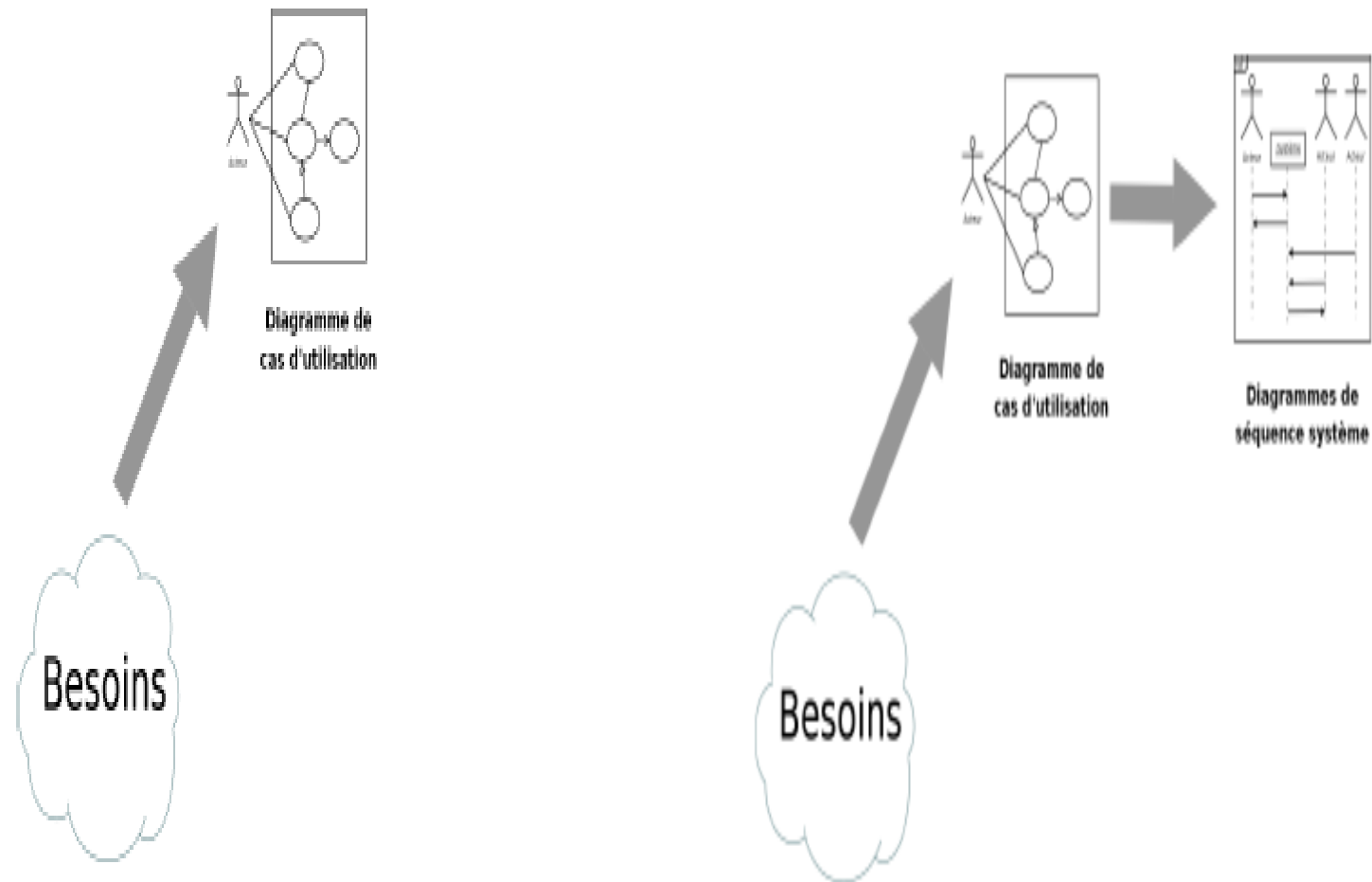


# Identification des besoins

- ❑ Les **cas d'utilisation** sont utilisés tout au long du projet. Dans un premier temps, on les crée pour identifier et modéliser les besoins des utilisateurs.
- ❑ Durant cette étape, vous devrez déterminer les limites du système, identifier les acteurs et recenser les cas d'utilisation.
- ❑ Si l'application est complexe, vous pourrez organiser les cas d'utilisation en paquetages.
- ❑ Dans le cadre d'une approche itérative et incrémentale, il faut affecter un degré d'importance et un coefficient de risque à chacun des cas d'utilisation pour définir l'ordre des incréments à réaliser.

Les interactions entre les acteurs et le système (au sein des cas d'utilisation) seront explicitées sous forme textuelle et sous forme graphique au moyen de **diagrammes de séquence..**



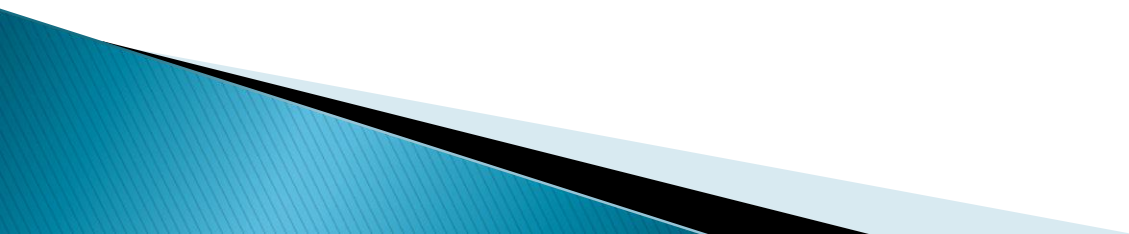


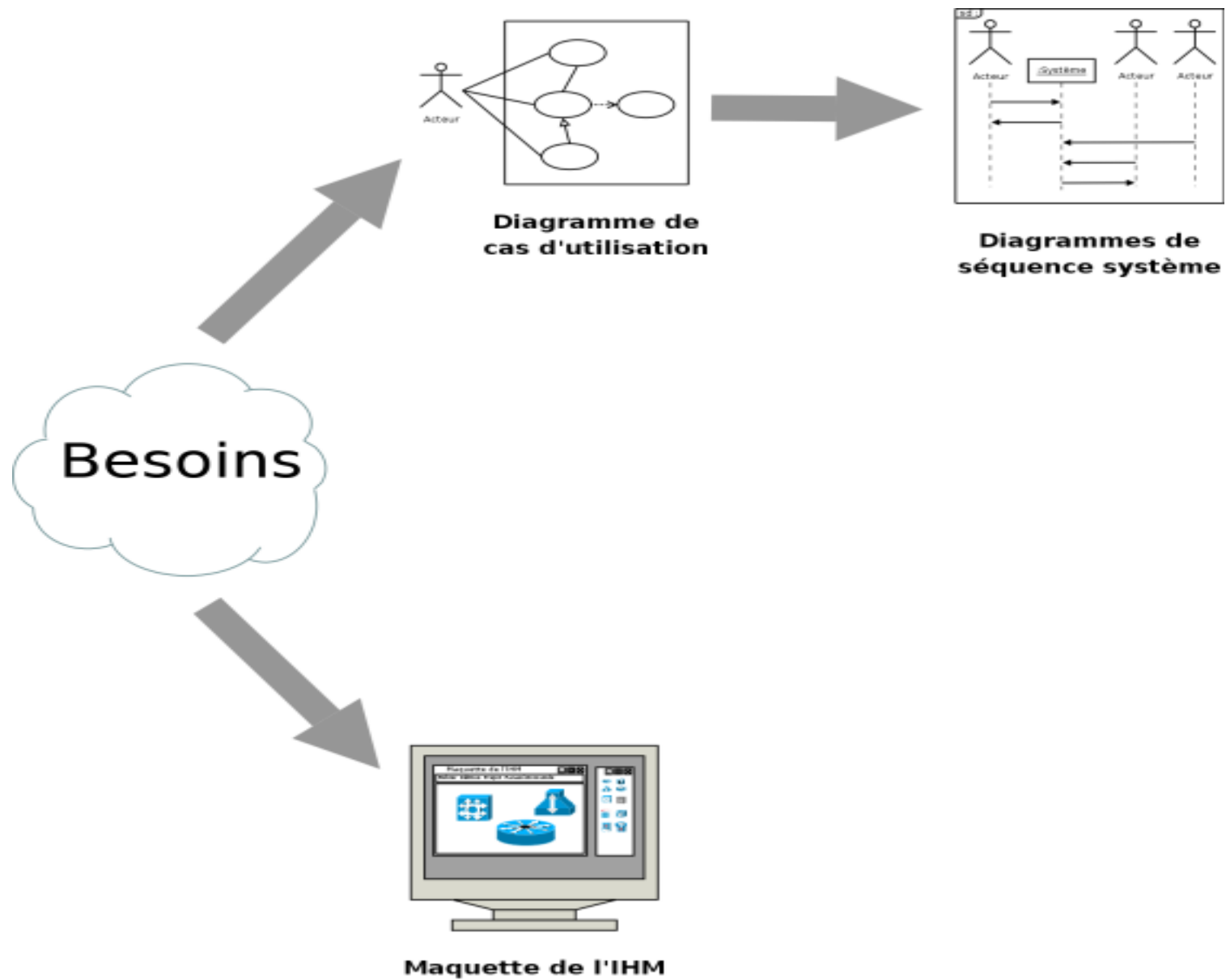
Les besoins sont modélisés par un diagramme de cas d'utilisation. Les diagrammes de séquence système illustrent la description textuelle des cas d'utilisation.

❑ Une maquette d'IHM (Interface Homme-Machine) est un produit jetable permettant aux utilisateurs d'avoir une vue concrète, mais non définitive de la future interface de l'application.

❑ La maquette peut très bien consister en un ensemble de dessins produits par un logiciel de présentation ou de dessin.

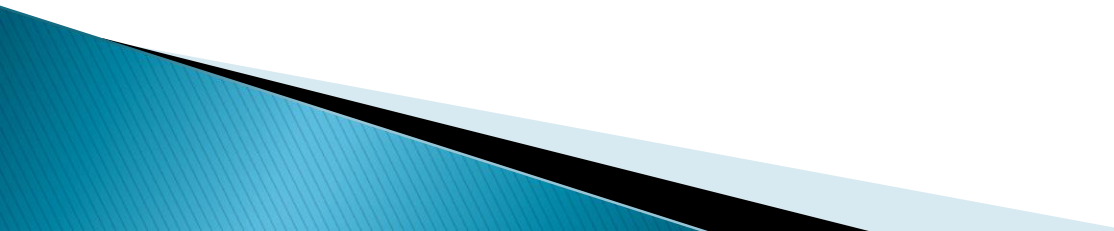
❑ Par la suite, la maquette pourra intégrer des fonctionnalités de navigation permettant à l'utilisateur de tester l'enchaînement des écrans ou des menus, même si les fonctionnalités restent fictives. La maquette doit être développée rapidement afin de provoquer des retours de la part des utilisateurs.

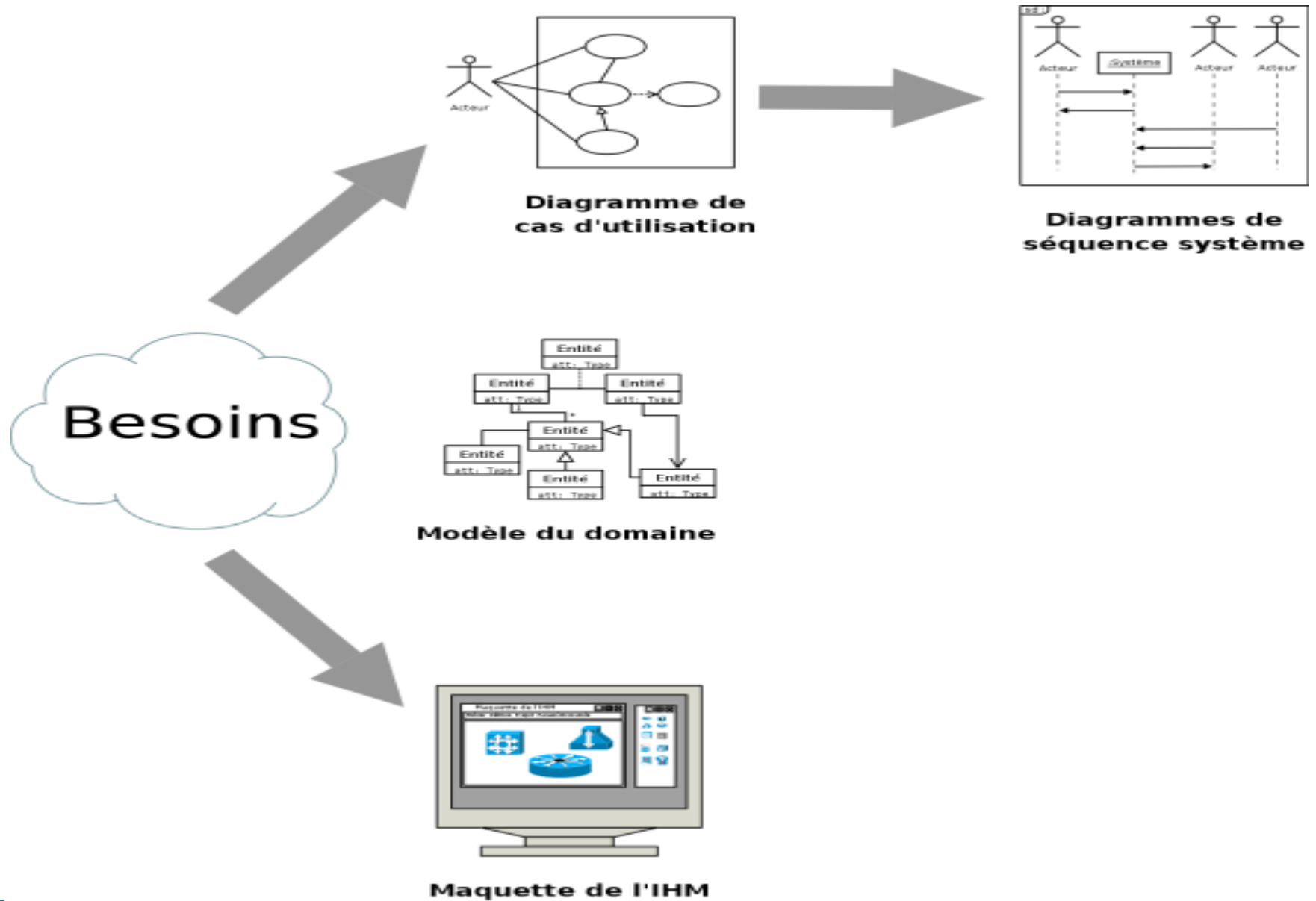




*Une maquette d'IHM facilite les discussions avec les futurs utilisateurs..*

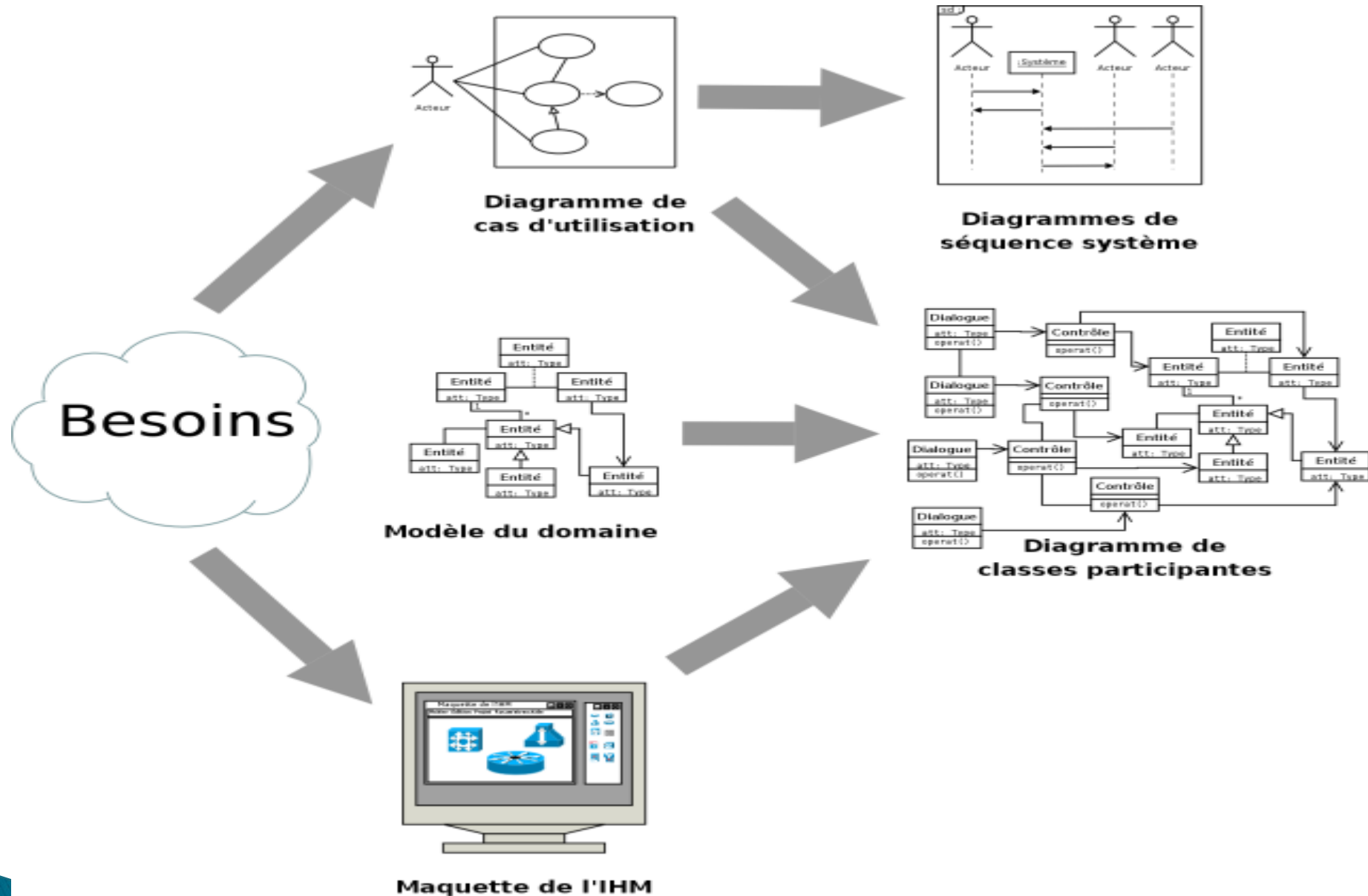
# Phase d'analyse

- ❑ La phase d'analyse du domaine permet d'élaborer la première version du diagramme de classes appelée **modèle du domaine**.
  - ❑ Ce modèle doit définir les classes qui modélisent les entités ou concepts présents dans le domaine (on utilise aussi le terme de *métier*) de l'application.
  - ❑ Il s'agit donc de produire un modèle des objets du monde réel dans un domaine donné.
  - ❑ Ces entités ou concepts peuvent être identifiés directement à partir de la connaissance du domaine ou par des entretiens avec des experts du domaine.
  - ❑ Les classes du modèle du domaine ne doivent pas contenir d'opération, mais seulement des attributs.
- 



*La phase d'analyse du domaine permet d'élaborer la première version du diagramme de classe.*

- ❑ **Le diagramme de classes participantes** est particulièrement important puisqu'il effectue la jonction entre, d'une part, les cas d'utilisation, le modèle du domaine et la maquette, et d'autre part, les diagrammes de conception logicielle que sont les diagrammes d'interaction et le diagramme de classes de conception.
- ❑ Il n'est pas souhaitable que les utilisateurs interagissent directement avec les instances des classes du domaine par le biais de l'interface graphique.
- ❑ En effet, le modèle du domaine doit être indépendant des utilisateurs et de l'interface graphique.
- ❑ De même, l'interface graphique du logiciel doit pouvoir évoluer sans répercussion sur le cœur de l'application. C'est le principe fondamental du découpage en couches d'une application.
- ❑ le diagramme de classes participantes modélise trois types de classes d'analyse, les *dialogues*, les *contrôles* et les *entités* ainsi que leurs relations.



*Le diagramme de classes participantes effectue la jonction entre les cas d'utilisation, le modèle du domaine et les diagrammes de conception logicielle.*



### ❑ Les classes de dialogues

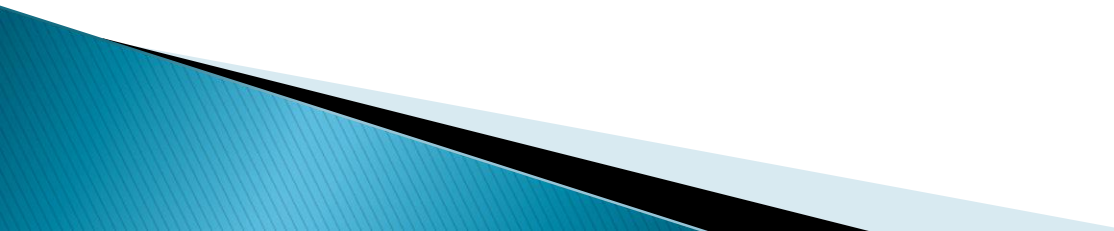
Les classes qui permettent les interactions entre l'IHM et les utilisateurs sont qualifiées de *dialogues*. Il y a au moins un dialogue pour chaque association entre un acteur et un cas d'utilisation.

### ❑ Les classes de contrôles

Les classes qui modélisent la cinématique de l'application sont appelées *contrôles*. Elles font la jonction entre les dialogues et les classes métier en permettant aux différentes vues de l'application de manipuler des informations détenues par un ou plusieurs objets métier. Elles contiennent les règles applicatives et les isolent à la fois des dialogues et des entités.

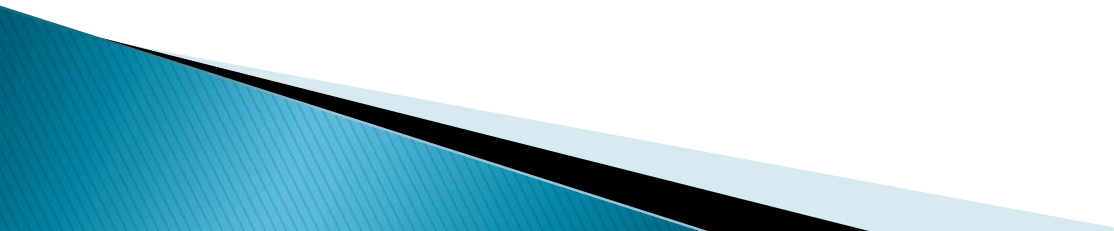
### ❑ Les classes entités

Les classes métier, qui proviennent directement du modèle du domaine, sont qualifiées d'*entités*. Ces classes sont généralement persistantes, c'est-à-dire qu'elles survivent à l'exécution d'un cas d'utilisation particulier et qu'elles permettent à des données et des relations d'être stockées dans des fichiers ou des bases de données. Lors de l'implémentation, ces classes peuvent ne pas se concrétiser par des classes, mais par des relations, au sens des bases de données relationnelles.



❑ Lors de l'élaboration du diagramme de classes participantes, il faut veiller au respect des règles suivantes :

- les entités, qui sont issues du modèle du domaine, ne comportent que des attributs;
- les entités ne peuvent être en association qu'avec d'autres entités ou avec des contrôles, mais, dans ce dernier cas, avec une contrainte de navigabilité interdisant de traverser une association d'une entité vers un contrôle ;
- les contrôles ne comportent que des opérations. Ils implémentent la logique applicative (*i.e.* les fonctionnalités de l'application), et peuvent correspondre à des règles transverses à plusieurs entités. Chaque contrôle est généralement associé à un cas d'utilisation, et *vice versa*. Mais rien n'empêche de décomposer un cas d'utilisation complexe en plusieurs contrôles ;
- les contrôles peuvent être associés à tous les types de classes, y compris d'autres contrôles.
- les dialogues comportent des attributs et des opérations. Les attributs représentent des informations ou des paramètres saisis par l'utilisateur ou des résultats d'actions. Les opérations réalisent (généralement par délégation aux contrôles) les actions que l'utilisateur demande par le biais de l'IHM ;
- les dialogues peuvent être en association avec des contrôles ou d'autres dialogues, mais pas directement avec des entités ;
- il est également possible d'ajouter les acteurs sur le diagramme de classes participantes en respectant la règle suivante : un acteur ne peut être lié qu'à un dialogue.

- ❑ Certaines classes possèdent un comportement dynamique complexe. Ces classes auront intérêt à être détaillées par des diagrammes d'états-transitions.
  - ❑ L'attribution des bonnes responsabilités aux bonnes classes est l'un des problèmes les plus délicats de la conception orientée objet. Ce problème sera affronté en phase de conception lors de l'élaboration des diagrammes d'interaction et du diagramme de classes de conception.
  - ❑ Lors de la phase d'élaboration du diagramme de classes participantes, le chef de projet a la possibilité de découper le travail de son équipe d'analystes par cas d'utilisation.
  - ❑ L'analyse et l'implémentation des fonctionnalités dégagées par les cas d'utilisation définissent alors les itérations à réaliser.
  - ❑ L'ordonnancement des itérations étant défini par le degré d'importance et le coefficient de risque affecté à chacun des cas d'utilisation.
- 

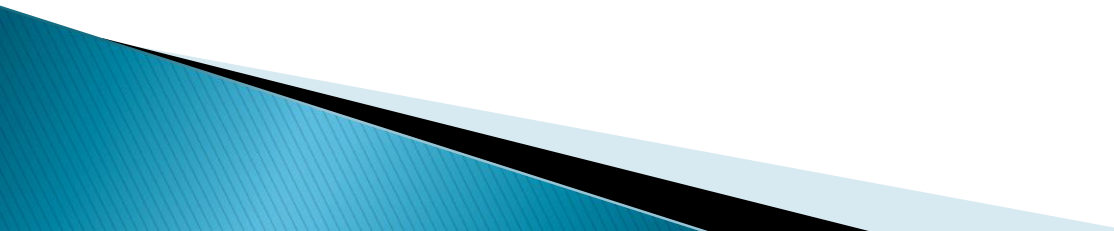
❑ Les **IHM modernes** facilitent la communication entre l'application et l'utilisateur en offrant toute une gamme de moyens d'action et de visualisation comme des menus déroulants ou contextuels, des palettes d'outils, des boîtes de dialogues, des fenêtres de visualisation, etc.

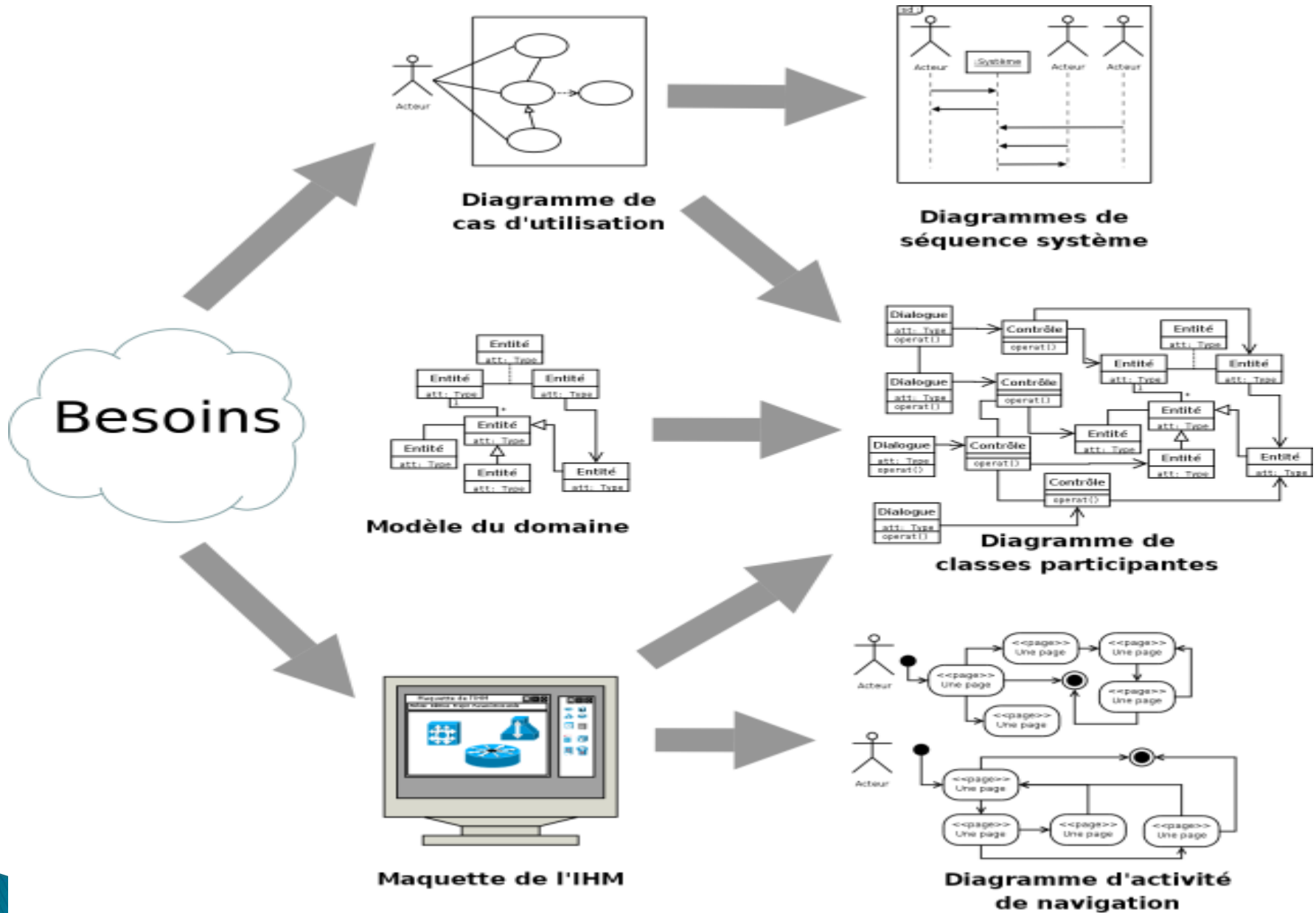
❑ UML offre la possibilité de représenter graphiquement cette activité de navigation dans l'interface en produisant des diagrammes dynamiques ( ou **diagrammes de navigation** ).

❑ Le concepteur a le choix d'opter pour cette modélisation entre des diagrammes d'états-transitions et des diagrammes d'activités. **Les diagrammes d'activités** constituent peut-être un choix plus souple et plus judicieux.

❑ Les diagrammes d'activités de navigation sont à relier aux classes de dialogue du diagramme de classes participantes. Les différentes activités du diagramme de navigation peuvent être stéréotypées en fonction de leur nature : « *fenêtre* », « *menu* », « *menu contextuel* », « *dialogue* », etc.

❑ La modélisation de la navigation a intérêt à être structurée par acteur.

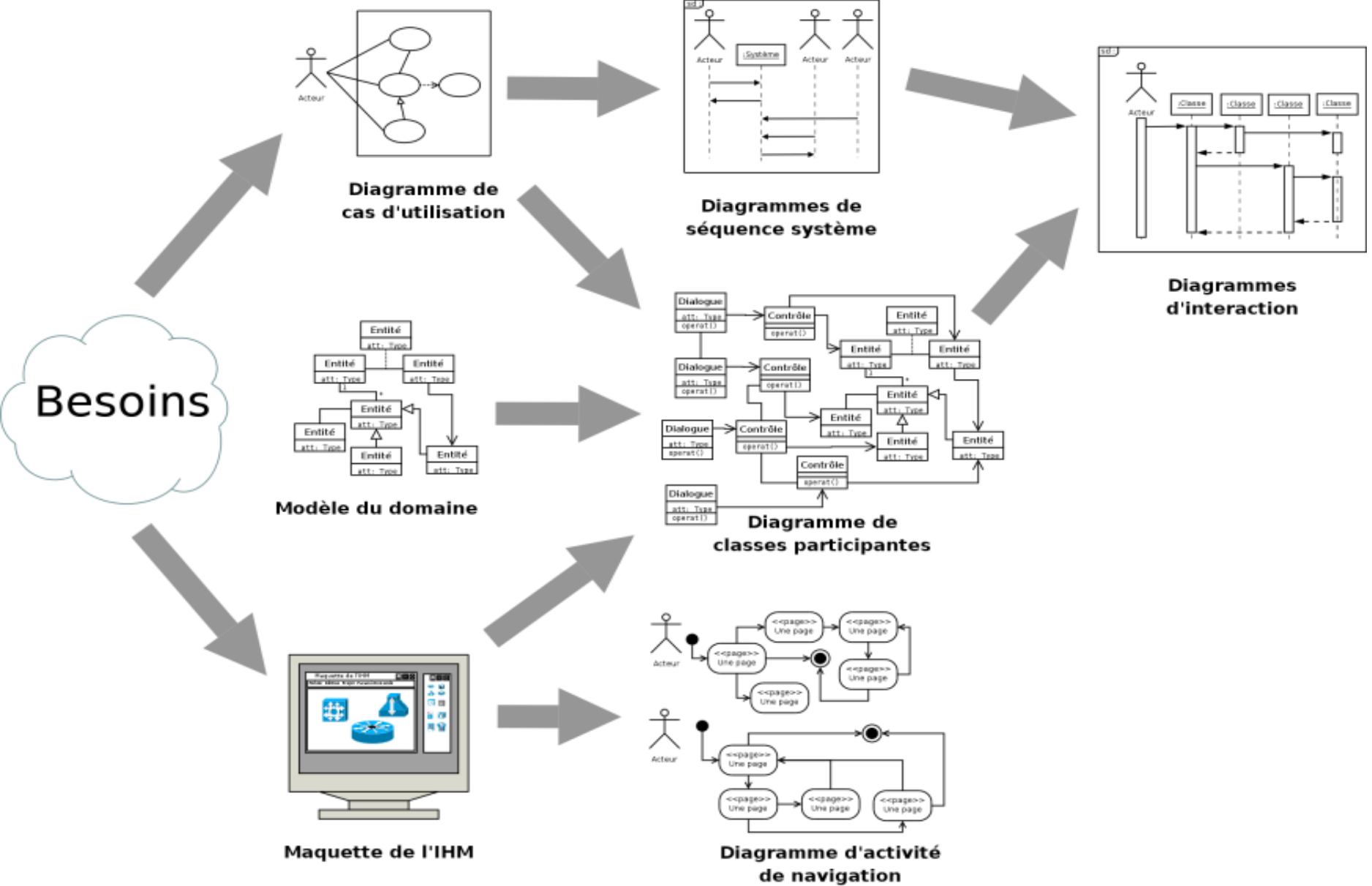





*Les diagrammes d'activités de navigation représentent graphiquement l'activité de navigation dans l'IHM.*

## Phase de conception

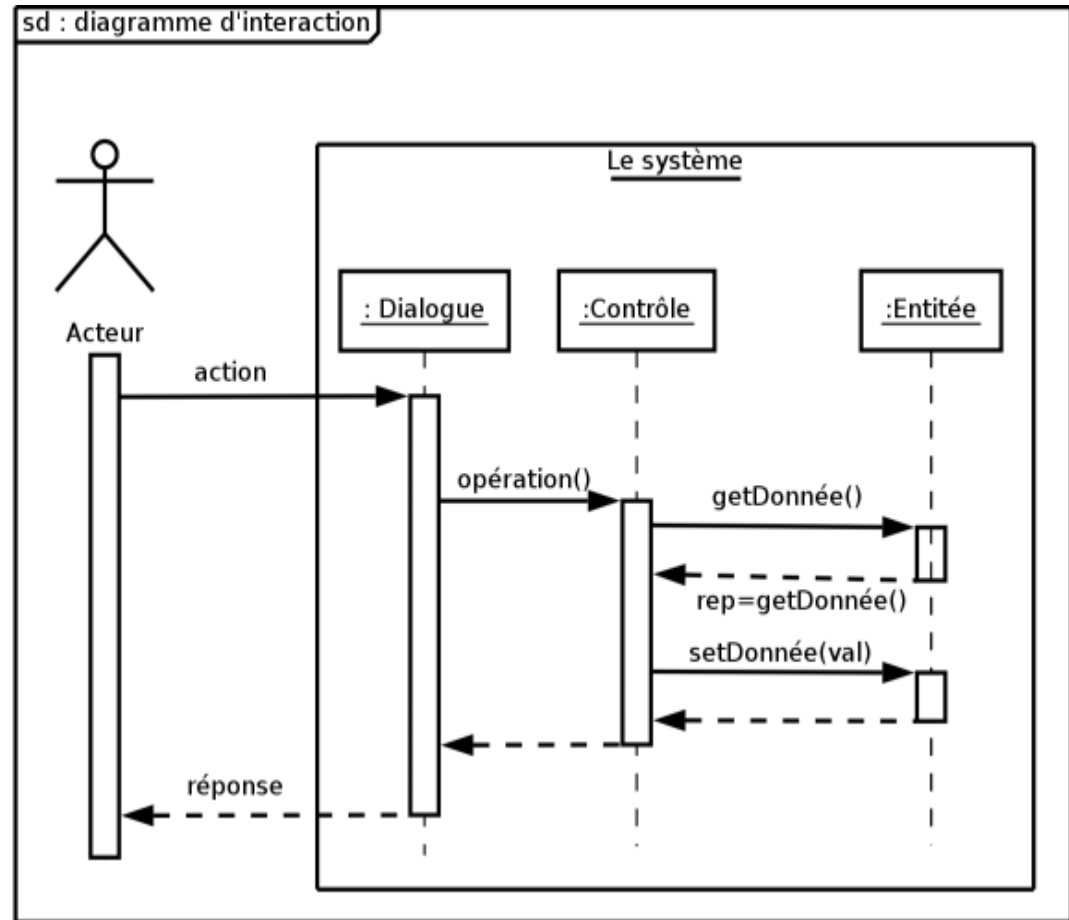
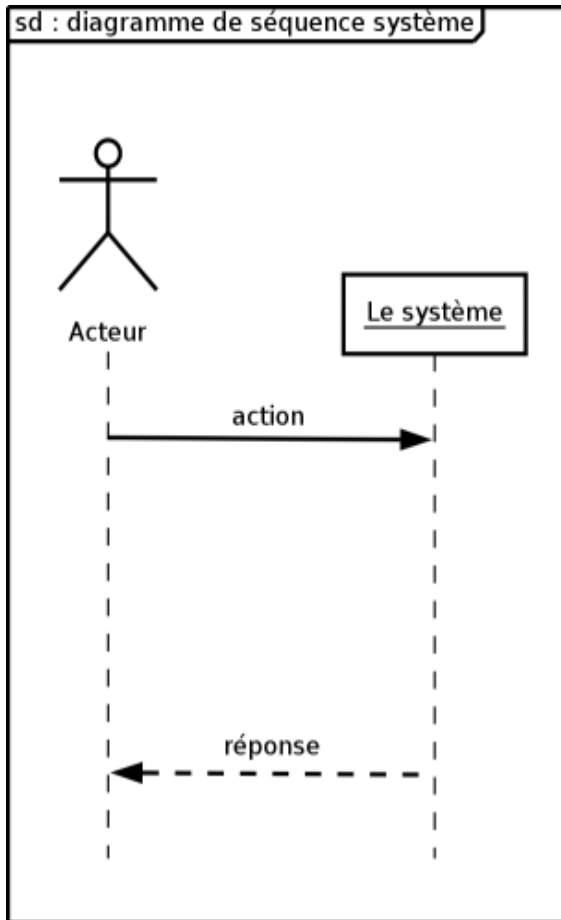
- ❑ Maintenant, il faut attribuer précisément les responsabilités de comportement, dégagées par le diagramme de séquence système aux classes d'analyse du diagramme de classes participantes élaboré.
- ❑ Les résultats de cette réflexion sont présentés sous la forme de **diagrammes d'interaction UML (figure )**.
- ❑ Inversement, l'élaboration de ces diagrammes facilite grandement la réflexion.
- ❑ Parallèlement, une première ébauche de la vue statique de conception, c'est-à-dire du diagramme de classes de conception, est construite et complétée. Durant cette phase, l'ébauche du diagramme de classes de conception reste indépendante des choix technologiques qui seront faits ultérieurement.



*Les diagrammes d'interaction permettent d'attribuer précisément les responsabilités de comportement aux classes d'analyse.*

- ❑ Pour chaque service ou fonction, il faut décider quelle est la classe qui va le contenir.
  - ❑ Les diagrammes d'interactions (i.e. de séquence ou de communication) sont particulièrement utiles au concepteur pour représenter graphiquement ces décisions d'allocations des responsabilités.
  - ❑ Chaque diagramme va représenter un ensemble d'objets de classes différentes collaborant dans le cadre d'un scénario d'exécution du système.
  - ❑ Dans les diagrammes d'interaction, les objets communiquent en s'envoyant des messages qui invoquent des opérations sur les objets récepteurs.
  - ❑ Il est ainsi possible de suivre visuellement les interactions dynamiques entre objets, et les traitements réalisés par chacun d'eux.
  - ❑ Avec un outil de modélisation UML (comme Rational Rose ou PowerAMC), la spécification de l'envoi d'un message entre deux objets crée effectivement une opération publique sur la classe de l'objet cible.
  - ❑ Ce type d'outil permet réellement de mettre en œuvre l'allocation des responsabilités à partir des diagrammes d'interaction.
- 





***Le système des diagrammes de séquences système, vu comme une boîte noire, est remplacé par un ensemble d'objets en collaboration.***

## Diagramme de classe de conception

- ❑ L'objectif de cette étape est de produire le diagramme de classes qui servira pour l'implémentation (figure).
- ❑ Une première ébauche du diagramme de classes de conception a déjà été élaborée en parallèle du diagramme d'interaction.
- ❑ Il faut maintenant le compléter en précisant les opérations privées des différentes classes. Il faut prendre en comptes les choix techniques, comme le choix du langage de programmation, le choix des différentes bibliothèques utilisées (notamment pour l'implémentation de l'interface graphique), etc.
- ❑ Pour une classe, le *couplage* est la mesure de la quantité d'autres classes auxquelles elle est connectée par des associations, des relations de dépendances, etc.
- ❑ Durant toute l'élaboration du diagramme de classes de conception, il faut veiller à conserver un couplage faible pour obtenir une application plus évolutive et plus facile à maintenir.
- ❑ L'utilisation des *design patterns* est fortement conseillée lors de l'élaboration du diagramme de classes de conception.
- ❑ Pour le passage à l'implémentation. Parfois, les classes du type entités ont intérêt à être implémentées dans une base de données relationnelle.

Besoins

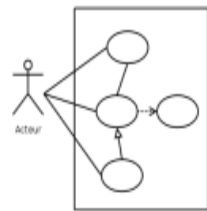
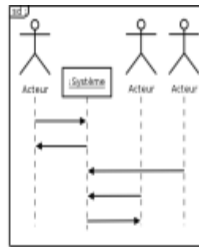
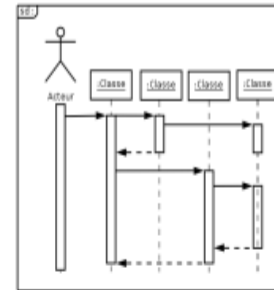


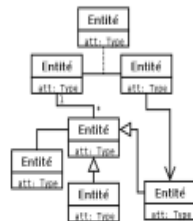
Diagramme de cas d'utilisation



Diagrammes de séquence système



Diagrammes d'interaction



Modèle du domaine

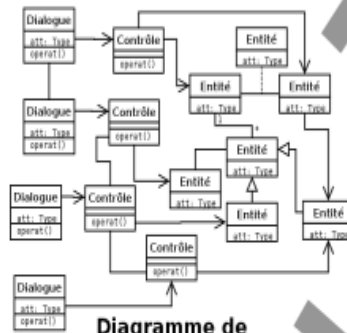
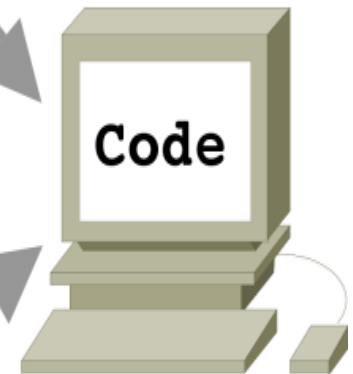


Diagramme de classes participantes



Maquette de l'IHM

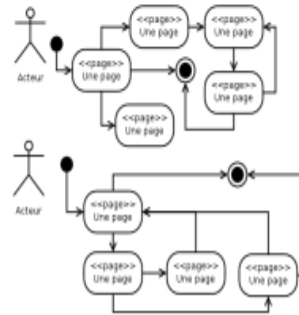


Diagramme d'activité de navigation

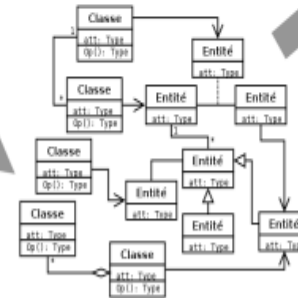


Diagramme de classes de conception

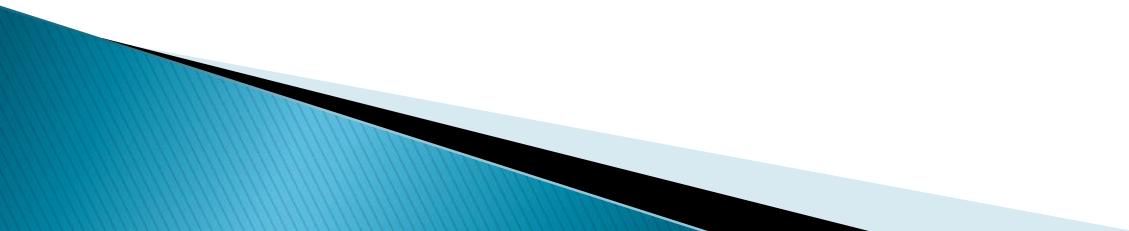
# III. Introduction aux Patterns de Conception

Notion de Pattern

Nature des patterns

Hiérarchie des objets réutilisables

Catégories des Patterns de conception



# III. Introduction aux Patterns de conception

## Notions sur la Conception Objet

- ❑ **But** : définir une solution informatique au problème décrit dans la phase d'analyse
- ❑ La conception consiste à enrichir le modèle objet issu de l'analyse :
  - typage des attributs, et niveau de visibilité
  - orientation des associations et agrégations  
(rendre "navigable" l'association dans un sens ou dans l'autre, ou les deux)
  - type d'implantation (par "valeur" ou par "référence") pour les associations
  - définition des méthodes en s'aidant des scénarios

# Conception d'objets réutilisables

- ❑ Concevoir des logiciel orienté-objet est difficile
- ❑ Concevoir du logiciel orienté-objet réutilisable, l'est plus encore
- ❑ Il faut:
  - déterminer les objets appropriés
  - les décomposer en classes, avec le niveau de détail adéquat
  - définir les interfaces des classes et les hiérarchies d'héritage
  - établir les relations clés entre elles
- ❑ Une conception doit être spécifique du problème à résoudre et être assez générale pour répondre aux problèmes et aux exigences futures

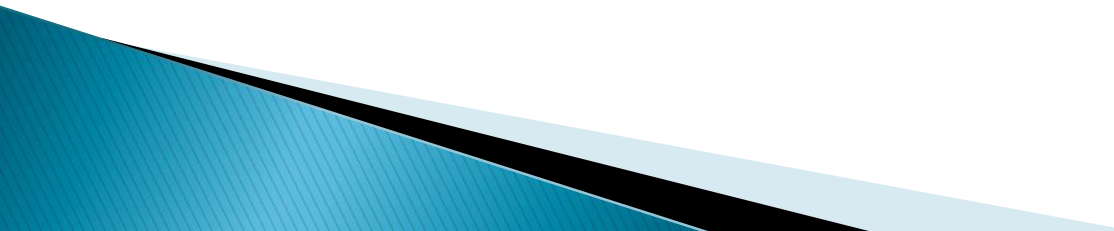
# Notion de Pattern

## Qu'est-ce qu'un pattern ?

- ❑ Un **pattern**, ou modèle, est un moyen d'accomplir quelque chose, un moyen d'atteindre un objectif, une technique.
- ❑ Le principe est de compiler les méthodes éprouvées qui s'appliquent à de nombreux types d'efforts, tels que la fabrication d'aliments, d'artifices, de logiciels, ou autres.
- ❑ Christopher Alexander a été un des premiers auteurs à compiler les meilleures pratiques d'un métier en documentant ses modèles.
- ❑ Dans « *A Pattern Language: Towns, Buildings Construction* » (Alexander, Ishikouwa, et Silverstein 1977), il décrit des modèles permettant de bâtir avec succès des immeubles et des villes.
- ❑ Cet ouvrage est puissant et a influencé la communauté logicielle.

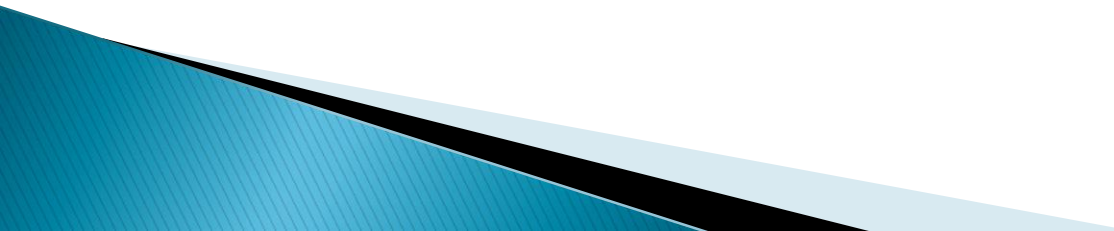
# Notion de Pattern ( 1 )

## Qu'est-ce qu'un pattern de conception ?

- ❑ Un pattern de conception (design pattern) est un modèle qui utilise des classes et leurs méthodes dans un langage orienté objet.
  - ❑ Les développeurs commencent souvent à s'intéresser à la conception seulement lorsqu'ils maîtrisent un langage de programmation et écrivent du code depuis longtemps.
  - ❑ Les patterns de conception interviennent un niveau au-dessus du code et indiquent typiquement comment atteindre un but en n'utilisant que quelques classes. Un pattern représente une idée, et non une implémentation particulière.
- 



# Nature des patterns

- ❑ Solutions indépendantes des langages
  - ❑ Solutions abstraites, de haut niveau
  - ❑ Souvent orientées vers le bon découpage en packages (modularité, flexibilité, réutilisabilité)
  - ❑ Exprimées sous forme d'architecture reliant quelques classes très abstraites
  - ❑ Reposent beaucoup sur des interfaces
- 

# Nature des patterns (1)

- ❑ Christopher Alexander décrit chaque modèle comme :
  - Un problème qui se manifeste constamment dans notre environnement.
  - Le cœur de la solution de ce problème réutilisable
- ❑ Un modèle possède quatre éléments essentiels:
  - Le nom de modèle
  - Le problème décrit les situations où le modèle s'applique, expose le sujet à traiter et son contexte
  - La solution décrit les éléments qui constituent la conception, les entre eux, leurs part dans la solution, leur coopération
  - Les conséquences sont les effets résultant, de la mise en du modèle et les variantes de compromis que celle-ci entraîne

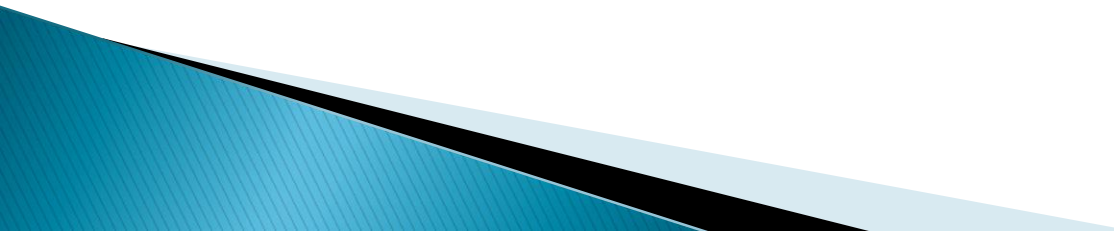
# Nature des patterns (2)

D. Schmidt (1995) : joueur d'échec versus développeur de logiciel

## ❑ Devenir un maître aux échecs

- ✓ apprendre les règles: nom des pièces, mouvement légal. . .
- ✓ apprendre les principes, les valeur relative des pièces...
- ✓ étudier le jeu des autres maîtres: certaines parties contiennent des modèles qu'il faut comprendre, mémoriser.
- ✓ il y a des milliers de modèles de parties

## ❑ Devenir un maître dans la conception logiciel

- ✓ apprendre les règles: les algorithmes, les structures de données, les langages...
  - ✓ apprendre les principes de programmation orientée objet...
  - ✓ étudier aussi les modèles (designs) des autres maîtres, les apprendre, les mémoriser...
  - ✓ il y a des centaines de modèle de conception
- 

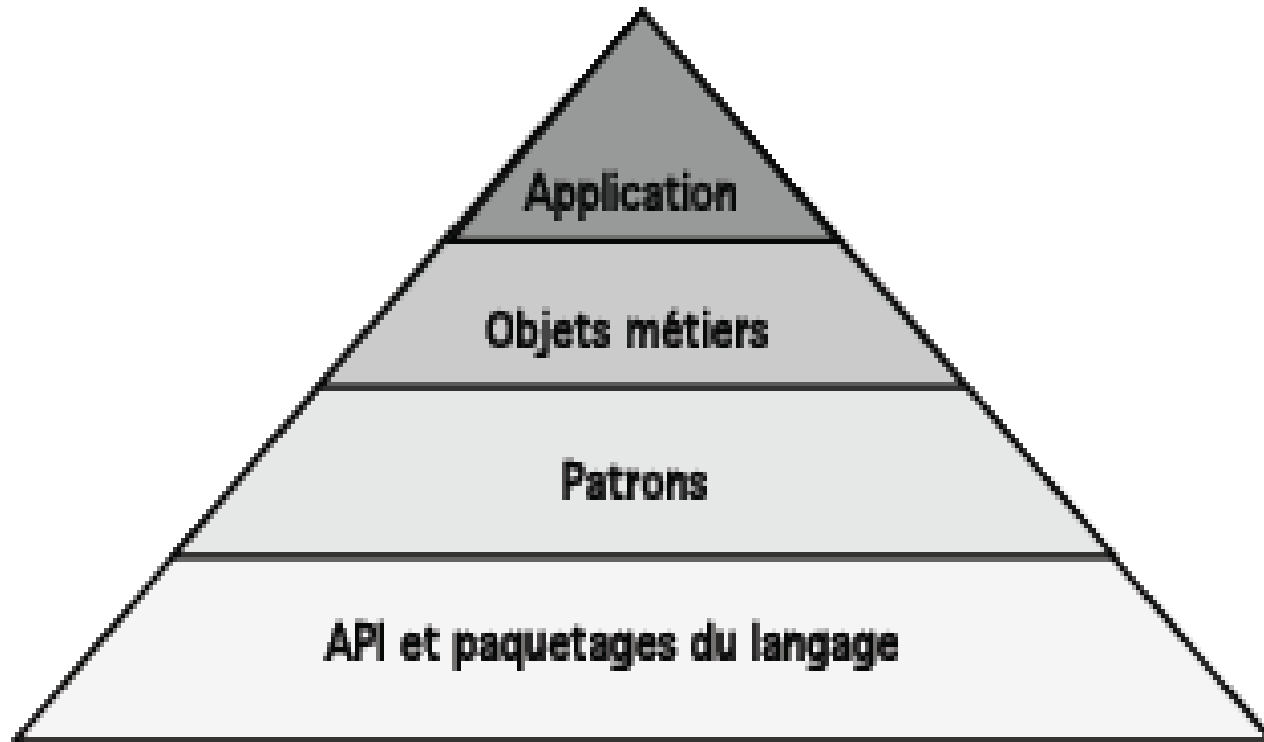
# Nature des patterns (3)

- ❑ Format de E. Gamma, R. Helm, R. Johnson et J. Vlissides:
  - ✓ but du patron
  - ✓ motivations du patron
  - ✓ situations dans lesquelles le patron peut s'appliquer
  - ✓ structure du patron
  - ✓ solution proposée (participants, collaborations)
  - ✓ façon dont le patron répond aux objectifs (détails)
  - ✓ conséquences d'utilisation du patron
  - ✓ Exemples
  - ✓ patrons qui sont liés à celui qui vient d'être décrit

# Nature des patterns (4)

- ❑ La forme la plus générale pour décrire un patron est :
  - ✓ Le nom du patron : il doit être bien choisi
  - ✓ Le problème que le patron essaye de résoudre : Quand on sait quel problème le patron résout, on sait quand l'appliquer
  - ✓ Le contexte : Un patron résout un problème dans un contexte donné et n'a pas de sens dans un autre contexte
  - ✓ Les forces impliquées, les efforts à effectuer ou les compromis
  - ✓ La solution : cette partie décrit la structure, le comportement de la solution
  - ✓ Des exemples
  - ✓ Le contexte résultant : ce qui reste à résoudre. Il doit aussi montrer de quel façon le contexte a été changé par le patron. Le "Design Rationale" explique d'où le patron provient, pourquoi il marche et aussi pourquoi les experts l'utilisent

# Hiérarchie des objets réutilisables



# Hiérarchie des objets réutilisables (1)

## Objet métier

- ❑ Un objet métier (“framework”) correspond au meilleur compromis trouvé par les concepteurs experts dans son domaine d'application:
  - il impose une architecture à une application
  - il définit sa structure globale (partitionnement en classes et en objets)
  - il déduit la tâche de contrôle, et donc,
  - il déduit les responsabilités essentielles (la façon de collaborer des classes et des objets)
  - il a la maîtrise des décisions de conception courantes dans son domaine d'application
  - il est plus concret et moins élémentaire que les “patrons”
  - il est construit en utilisant des “patrons”

# Hiérarchie des objets réutilisables (2)

## Patrons

- ❑ Un “patron” (“pattern”) correspond à la solution à un problème dans un certain contexte
- ❑ Les “patrons” sont comme ceux utilisés en couture (exemple: tailler une chemise), ils répondent à un problème
- ❑ L'utilisation du patron devra être modifiée en fonction du contexte dans lequel il est utilisé (taille de la personne, tissu utilisé, etc ...)
- ❑ Les patrons permettent une réutilisation des connaissances en une certaine formalisation concernant les explications pour une solution à un problème



Transmission de la connaissance facilitée



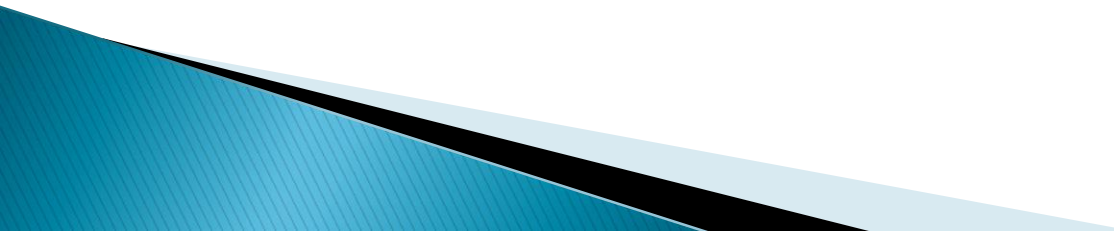
# Catégories des Patterns de conception

- Il existe 23 Patterns de conception qui font l'unanimité.
- Ces Patterns sont classifiés en 3 classes chacune spécialisée dans :
  - ✓ Création/Construction d'objets (*creational patterns*):
    - liés au problème de choisir la classe responsable des créations, de choisir le type créé
    - permet l'encapsulation des classes effectives
  - ✓ structure des relations entre objets (*structural patterns*)
    - liés aux problèmes d'organisation des objets dans un logiciel
    - Composition des classes et des objets
  - ✓ comportement des objets (*behavioral patterns*)
    - liés aux problèmes de communication entre les objets
    - Distribution des responsabilités

		ROLE		
		Créateur	Structurel	Comportement
DOMAINE	CLASSE	Fabrication	Adaptateur(classe)	Interprete Patron de méthode
	OBJET	Fabrique abstraite Monteur Prototype Singleton	Adaptateur(objet) Pont Décorateur Facade Composite Poids mouche Procuration	Chaine de responsabilité Commande Itérateur Médiateur Memento Observateur Etat Stratégie Visiteur

# patterns de construction

## ■ Présentation

- ❑ Les patterns de construction ont pour vocation d'abstraire les mécanismes de création d'objets.
  - ❑ Un système utilisant ces patterns devient indépendant de la façon dont les objets sont créés et, en particulier, des mécanismes d'instanciation des classes concrètes.
  - ❑ Ces patterns encapsulent l'utilisation des classes concrètes et favorisent ainsi l'utilisation des interfaces dans les relations entre objets, augmentant les capacités d'abstraction dans la conception globale du système.
- 
- Ainsi, le pattern Singleton permet de construire une classe possédant au maximum une instance.
  - Le mécanisme gérant l'accès unique à cette seule instance est entièrement encapsulé dans la classe. Il est transparent pour les clients de cette classe.
- 

# patterns de construction (1)

## ▪ Problématique

- ❑ Dans la plupart des langages à objets, la création d'objets se fait grâce au mécanisme d'instanciation qui consiste à créer un nouvel objet par appel de l'opérateur new paramétré par une classe (et éventuellement des arguments du constructeur de la classe dont le but est de donner aux attributs leur valeur initiale).
- ❑ Un tel objet est donc une instance de cette classe.
- ❑ Les langages les plus utilisés aujourd'hui comme Java, C++ ou C# utilisent le mécanisme de l'opérateur new.

**En Java instruction de création d'un objet :: objet = new Classe();**

- ❑ Dans certains cas, il est nécessaire de paramétrer la création d'objets. Prenons l'exemple d'une méthode construitDoc qui crée des documents.
- ❑ Elle peut construire des documents PDF, RTF ou HTML. Généralement le type du document à créer est transmis en paramètre à la méthode sous forme d'une chaîne de caractères, ce qui donne le code suivant :

```
public Document construitDoc(String typeDoc)
{
    Document resultat;
    if (typeDoc.equals("PDF"))
        resultat = new DocumentPDF();
    else if (typeDoc.equals("RTF"))
        resultat = new DocumentRTF();
    else if (typeDoc.equals("HTML"))
        resultat = new DocumentHTML();
    // suite de la methode
}
```

- ❑ Cet exemple nous montre qu'il est difficile de paramétrer le mécanisme de création d'objets, la classe transmise en paramètre à l'opérateur new ne pouvant être substituée par une variable.
- ❑ L'utilisation d'instructions conditionnelles dans le code du client est souvent pratiquée avec l'inconvénient que chaque changement dans la hiérarchie des classes à instancier demande des modifications dans le code des clients ( ex. ajout de nouvelles classes de document).
- ❑ La difficulté est encore plus grande quand il faut construire des objets composés dont les composants peuvent être instanciés à partir de classes différentes. Par exemple, une liasse de documents peut être formée de documents PDF, RTF ou HTML. Le client doit alors connaître toutes les classes possibles des composants et des composés.
- ❑ Chaque modification dans ces ensembles de classes devient alors très lourde à gérer.

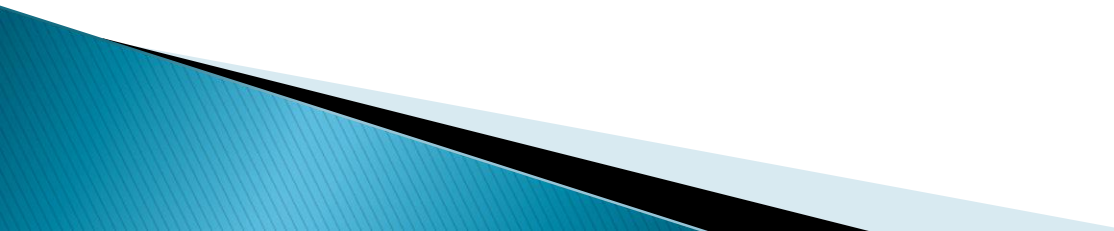
# Patterns fondamentaux

- Délégation
- Interface
- Classe abstraite
- Interface et Classe abstraite

**Remarque :** Les patrons fondamentaux découlent souvent directement des concepts présents dans les langages objets.

⇒ Suivant les auteurs, ils sont souvent considérés comme implicites

**Intérêt :**

- Donnent un autre éclairage aux concepts objet
  - Sont utilisés par les autres patrons
- 

# Délégation

❑ **Contexte** : Parfois, utiliser l'héritage conduit à une mauvaise conception. La délégation est alors un mécanisme plus général (même si plus lourd).

❑ **Cas où ne pas utiliser l'héritage (mais la délégation) :**

- *Une classe métier veut réutiliser une classe « utilitaire ».*
  - Évolutions de l'« utilitaire » compatibles avec la classe « métier » ?
- *Une classe veut cacher des éléments de la superclasse (Anti-patron !).*
  - impossible à faire (si héritage implique sous-typage, ex. Java) !

**Exemple** : Pile ne doit pas hériter de ArrayList.

- *Les sous-classes modélisent des rôles différents de la superclasse.*

**Exemple** : Modéliser un joueur de foot.

Spécialiser Joueur en Gardien, Stoppeur, Attaquant... ou définir une délégation sur Poste qui est spécialisée en Gardien, Stoppeur, Attaquant...

Un joueur peut changer de poste, voire occuper plusieurs postes.

**Remarque** : L'héritage correspond à « est une sorte de » mais n'est pas adapté pour représenter « est un rôle joué par » car, sinon, le rôle ne peut pas changer pendant la durée de vie de l'objet.

# Interface

❑ **Intention** : On souhaite qu'un client reste indépendant de la classe qui fournit effectivement le service. C'est justement l'objectif des interfaces ! (patron fondamental)

**Exemple** : Un client veut accéder à un fournisseur JDBC pour accéder à une base de données. Le fournisseur effectif dépendra de la BD considérée. Le code client n'a pas à le connaître, seulement les interfaces qui définissent les services JDBC.

❑ **Intérêt** : Le fournisseur de service peut changer sans impact sur le code client.

Interface





# Classe abstraite

## ❑ Intention :

- Garantir que la logique commune à plusieurs classes est implantée de manière cohérente pour chaque classe
- Éviter le code redondant (et les efforts de maintenance associés)
- Faciliter l'écriture de nouvelles classes implantant la même logique

## ❑ Solution :

- Utiliser une classe abstraite pour factoriser le code commun implantant la logique
- Les méthodes peuvent éventuellement être définies comme **final**

❑ **Exemple** : La classe abstraite `java.util.AbstractCollection` définit toutes les opérations de `Collection` sauf `size` et `iterator`.

- Collection concrète non modifiable : définir seulement `size` et `iterator`
- Collection concrète modifiable : définir aussi `add` (et `remove` sur l'iterator).

# III. Le pattern Abstract Factory

## Description

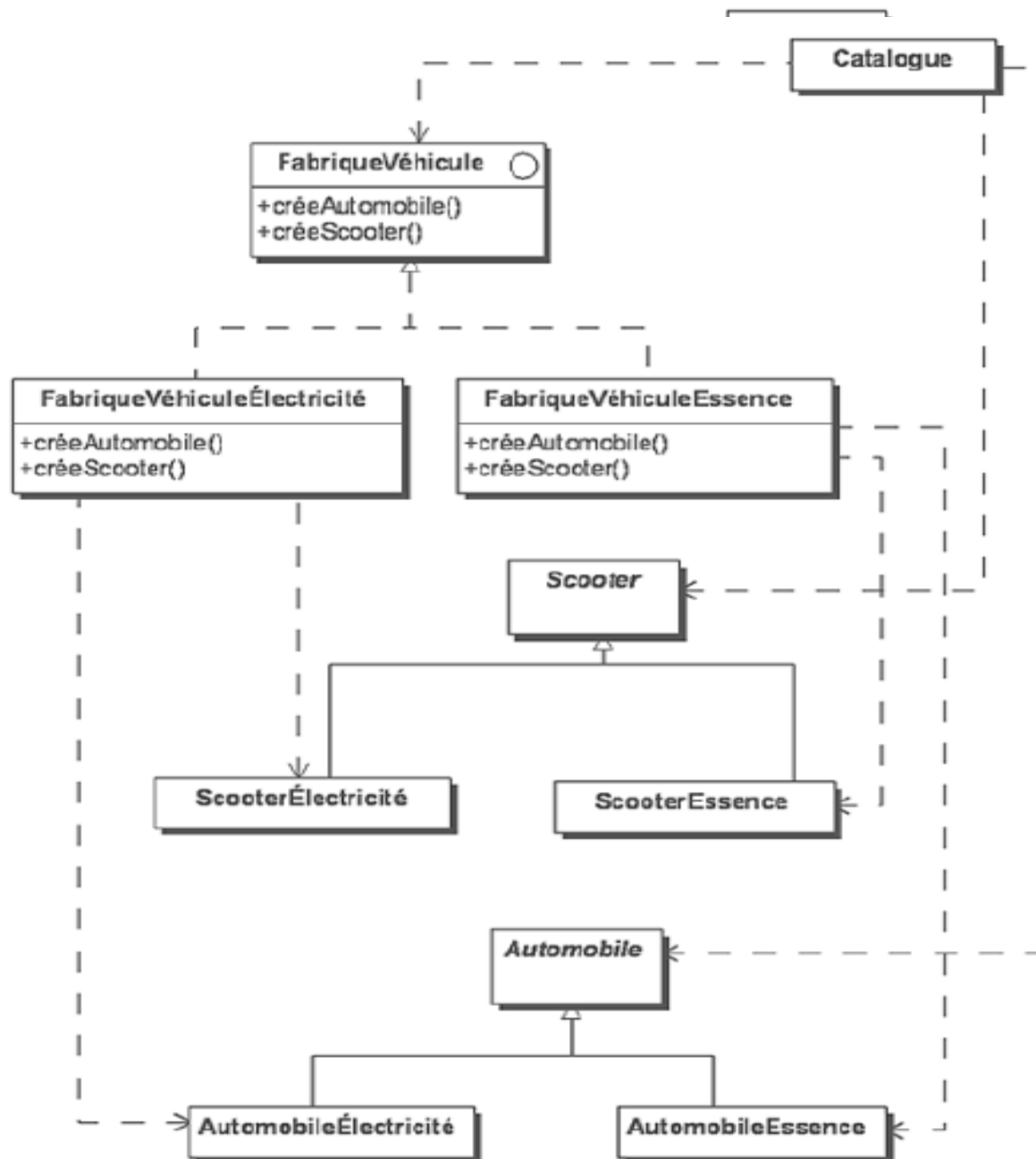
Le but du pattern Abstract Factory est la création d'objets regroupés en familles sans devoir connaître les classes concrètes destinées à la création des ces objet

## Exemple

- ❑ Le système de vente de véhicules gère des véhicules fonctionnant à l'essence et des véhicules fonctionnant à l'électricité. Cette gestion est confiée à l'objet Catalogue qui crée de tels objets. Pour chaque produit, nous disposons d'une classe abstraite, d'une sousclasse concrète décrivant la version du produit fonctionnant à l'essence et d'une sousclasse décrivant la version du produit fonctionnant à l'électricité.
- ❑ Par exemple, à la figure ci-dessous, pour l'objet scooter, il existe une classe abstraite Scooter et deux sousclasses concrètes ScooterÉlectricité et ScooterEssence.
- ❑ L'objet Catalogue peut utiliser ces sousclasses concrètes pour instancier les produits. Cependant si, par la suite, de nouvelles familles de véhicules doivent être prises en compte par la suite (diesel ou mixte essenceélectricité), Les modifications à apporter à l'objet Catalogue peuvent être assez lourdes.

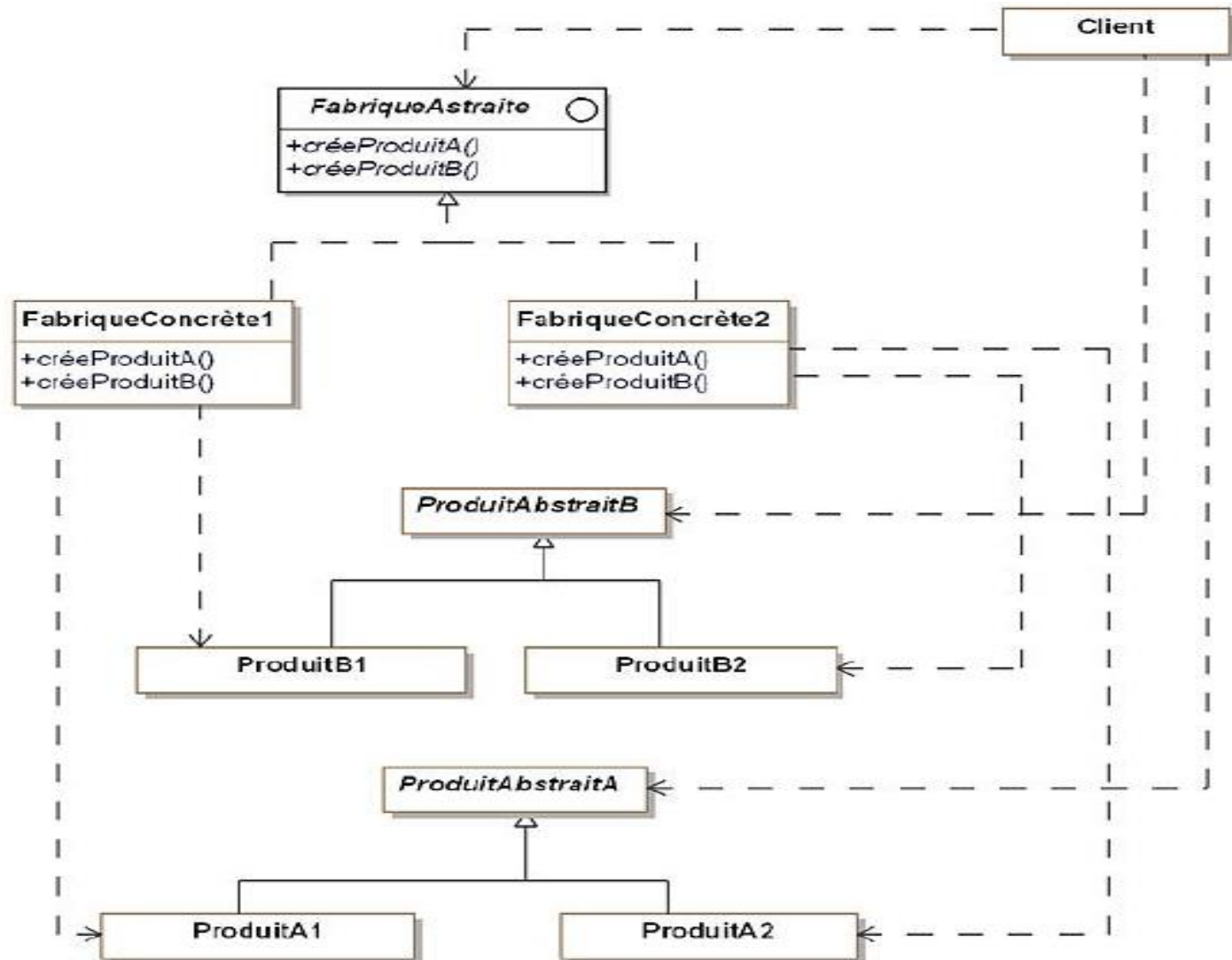
- ❑ Le pattern Abstract Factory résout ce problème en introduisant une interface `FabriqueVéhicule` qui contient la signature des méthodes pour définir chaque produit. Le type de retour de ces méthodes est constitué par l'une des classes abstraites de produit. Ainsi l'objet `Catalogue` n'a pas besoin de connaître les sousclasses concrètes et reste indépendant des familles de produit.
- ❑ Une sousclasse d'implantation de `FabriqueVéhicule` est introduite pour chaque famille de produit, à savoir les sousclasses `FabriqueVéhiculeÉlectricité` et `FabriqueVéhiculeEssence`.
- ❑ Une telle sousclasse implante les opérations de création du véhicule appropriée pour la famille à laquelle elle est associée.
- ❑ L'objet `Catalogue` prend alors pour paramètre une instance répondant à l'interface `FabriqueVéhicule` c'est à dire soit une instance de `FabriqueVéhiculeÉlectricité`, soit une instance de `FabriqueVéhiculeEssence`. Avec une telle instance, le catalogue peut créer et manipuler des véhicules sans devoir connaître les familles de véhicule et les classes concrètes d'instanciation correspondantes.

L'ensemble des classes du pattern Abstract Factory pour cet exemple est détaillé à la figure ci-après



# Structure

Ce diagramme de classe détaille la structure générique du patron



## Participants

❑ Les participants au pattern sont les suivants :

- FabriqueAbstraite (FabriqueVéhicule) est une interface spécifiant les signatures des méthodes créant les différents produits ;
- FabriqueConcrète1, FabriqueConcrète2 (FabriqueVéhiculeÉlectricité, FabriqueVéhiculeEssence) sont les classes concrètes implantant les méthodes créant les produits pour chaque famille de produit. Connaissant la famille et le produit, elles sont capables de créer une instance du produit pour cette famille ;
- ProduitAbstraitA et ProduitAbstraitB (Scooter et Automobile) sont les classes abstraites des produits indépendamment de leur famille. Les familles sont introduites dans leurs sousclasses concrètes ;
- Client est la classe qui utilise l'interface de FabriqueAbstraite.

## Collaborations

- ❑ La classe Client utilise une instance de l'une des fabriques concrètes pour créer ses produits au travers de l'interface de FabriqueAbstraite.

## Domaines d'utilisation

- ❑ Le pattern est utilisé dans les domaines suivants :
  - Un système utilisant des produits a besoin d'être indépendant de la façon dont ces produits sont créés et regroupés ;
  - Un système est paramétré par plusieurs familles de produits qui peuvent évoluer.

# Patterns de Structuration

## ■ Présentation

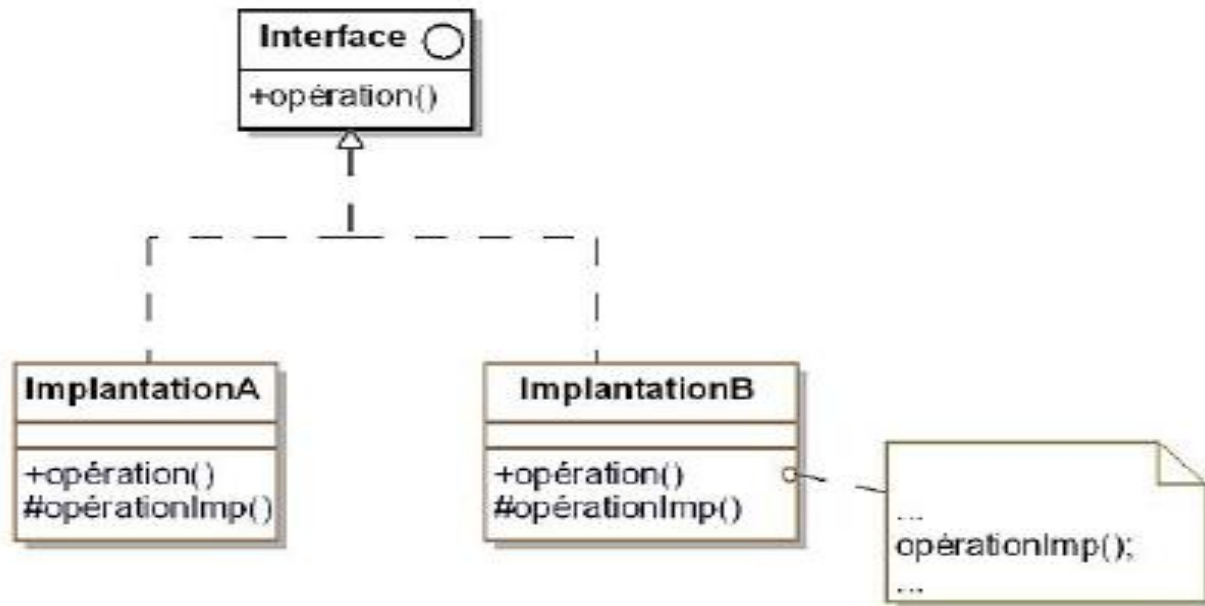
- ❑ L'objectif des patterns de structuration est de faciliter l'indépendance de l'interface d'un objet ou d'un ensemble d'objets vis-à-vis de son implantation. Dans le cas d'un ensemble d'objets, il s'agit aussi de rendre cette interface indépendante de la hiérarchie des classes et de la composition des objets.
- ❑ En fournissant les interfaces, les patterns de structuration encapsulent la composition des objets, augmentant le niveau d'abstraction du système à l'image des patterns de création qui encapsulent la création des objets. Les patterns de structuration mettent en avant les interfaces.
- ❑ L'encapsulation de la composition est réalisée non pas en structurant l'objet lui-même mais en transférant cette structuration à un second objet lié premier objet. Ce transfert de structuration signifie que le premier objet détient l'interface vis-à-vis des clients et gère la relation avec le second objet qui lui gère la composition et n'a aucune interface avec les clients externes.
- ❑ Cette réalisation offre une autre propriété qui est la souplesse de la composition qui peut être modifiée dynamiquement.



# patterns de Structuration (1)

## ▪ Composition statique et dynamique

- ❑ Nous prenons l'exemple des aspects d'implantation d'une classe. Nous nous plaçons dans un cadre où il est possible d'avoir plusieurs implantations possibles. La solution classique consiste à les différencier au niveau des sousclasses.
- C'est le cas de l'utilisation de l'héritage d'une interface dans plusieurs classes d'implantation comme l'illustre la figure ci-dessous.
- Cette solution consiste à réaliser une composition statique. En effet, une fois que le choix de la classe d'implantation d'un objet est effectué, il n'est plus possible d'en changer.



## patterns de Structuration (2)

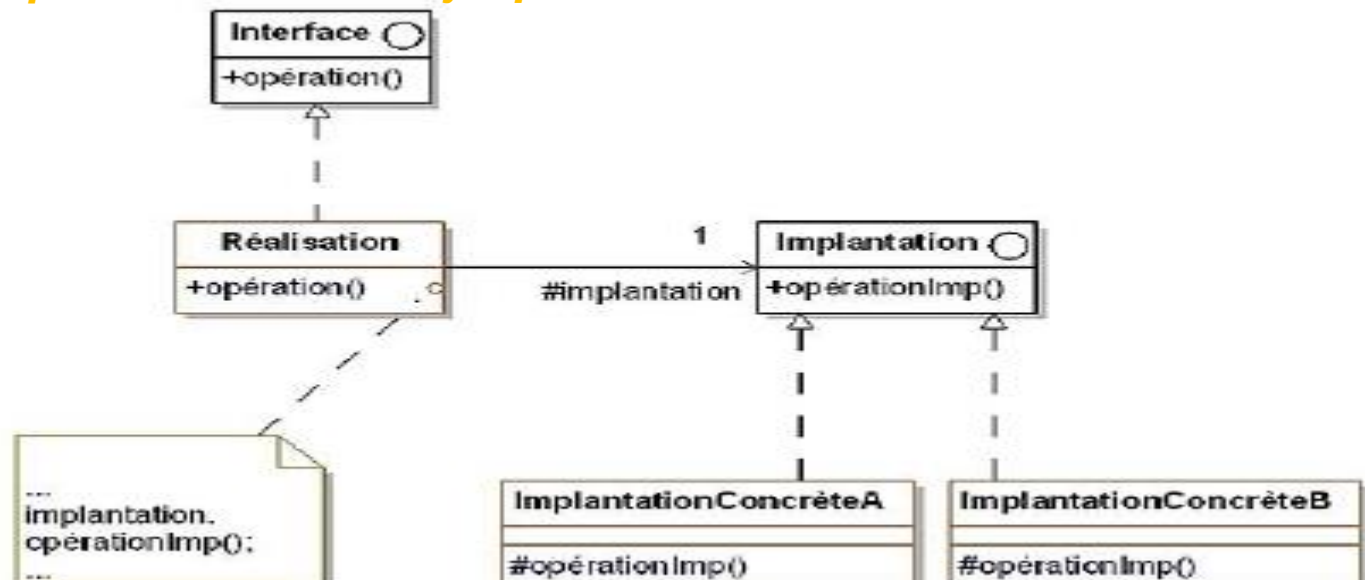
### ▪ Composition statique et dynamique

- ❑ Une autre solution est de séparer l'aspect d'implantation dans un autre objet comme l'illustre la figure ci-après.
- ❑ Les parties d'implantation sont gérées par une instance de la classe `ImplantationConcrèteA` ou par une instance de la classe `ImplantationConcrèteB`.
- ❑ Cette instance est référencée par l'attribut `implantation`. Elle peut être substituée facilement par une autre instance lors de l'exécution.



**Par conséquent, la composition est dynamique.**

### *Exemple d'Implantation d'un objet par association*



# Introduction aux patterns de Structuration (3)

## ■ Composition statique et dynamique

- ❑ Tous les patterns de structuration sont basés sur l'utilisation d'un ou de plusieurs objets déterminant la structuration.
- ❑ La liste suivante décrit la fonction que remplit cet objet pour chaque pattern:
  1. **Adapter** : adapte un objet existant.
  2. **Bridge** : implante un objet.
  3. **Composite** : organise la composition hiérarchique d'un objet.
  4. **Decorator** : se substitue à l'objet existant en lui ajoutant de nouvelles fonctionnalités.
  5. **Facade** : se substitue à un ensemble d'objets existants en leur conférant une interface unifiée.
  6. **Flyweight** : est destiné au partage et détient un état indépendant des objets qui le référencent.
  7. **Proxy** : se substitue à l'objet existant en fournissant un comportement adapté à des besoins d'optimisation ou de protection.

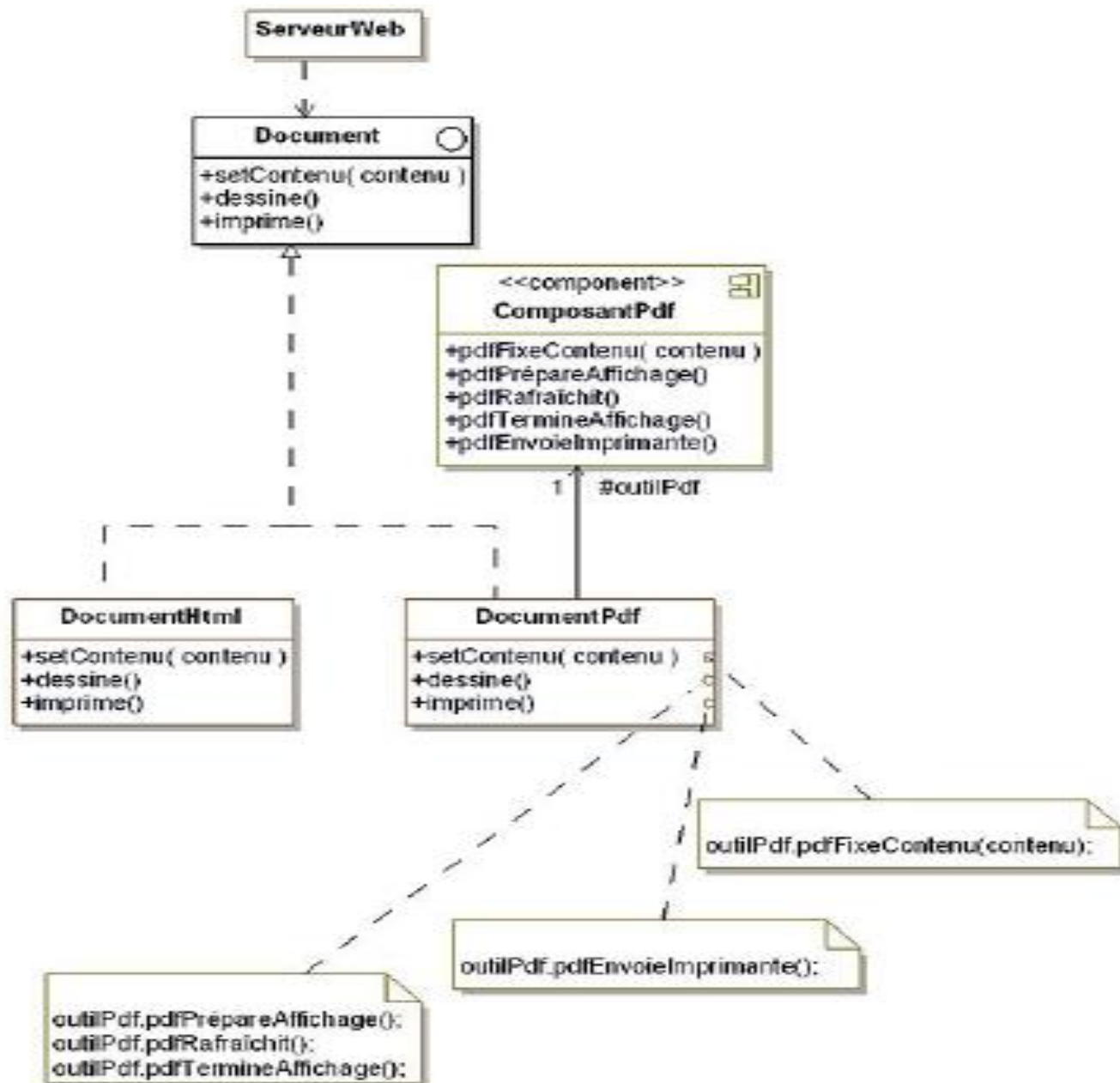
# II. Le Pattern Adapter

## Description

- ❑ Le but du pattern Adapter est de convertir l'interface d'une classe existante en l'interface attendue par des clients.
- ❑ Il s'agit de conférer à une classe existante une nouvelle interface pour répondre aux besoins de clients.

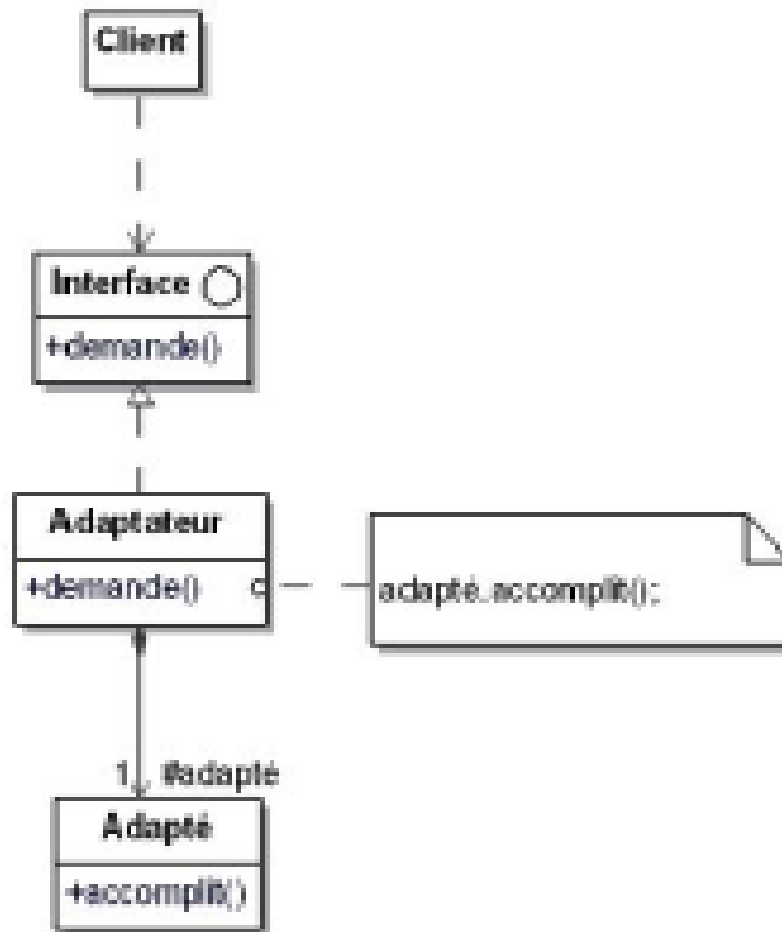
## Exemple :

- Le serveur web du système de vente de véhicules crée et gère des documents destinés aux clients.
- L'interface à Document a été définie pour cette gestion ainsi que ses trois méthodes setContenu, dessine et imprime.
- Une première classe d'implantation de cette interface a été réalisée : la classe DocumentHtml qui implante ces trois méthodes.
- Par la suite, l'ajout des documents PDF a posé un problème car ceux-ci sont plus complexes à construire et à gérer que des documents HTML.
- Un composant du marché a été choisi mais dont l'interface ne correspond à l'interface Document.
- La figure ci-dessous montre le composant ComposantPdf dont l'interface introduit plus de méthodes et dont la convention de nommage est de surcroît différente (préfixe pdf).
- Le pattern Adapter propose une solution qui consiste à créer la classe DocumentPdf implantant l'interface Document et possédant une association avec ComposantPdf.
- L'implantation des trois méthodes de l'interface Document consiste à déléguer



## Structure

Le Diagramme de classes détaille la structure générique du pattern.



## Participants

- ❑ Interface (Document) introduit la signature des méthodes de l'objet ;
- ❑ Client (ServeurWeb) interagit avec les objets répondant à Interface ;
- ❑ Adaptateur (DocumentPdf) implante les méthodes de Interface en invoquant les méthodes de l'objet adapté ;
- ❑ Adapté (ComposantPdf) introduit l'objet dont l'interface doit être adaptée pour correspondre à Interface.

## Collaborations

- ❑ Le client invoque la méthode demande de l'adaptateur qui, en conséquence, interagit avec l'objet adapté en appelant la méthode accomplit.

## Domaines d'application

- ❑ Pour intégrer dans un système un objet dont l'interface ne correspond pas à l'interface requise au sein de ce système ;
- ❑ Pour fournir des interfaces multiples à un objet lors de sa conception.

# Patterns de Comportement

## Présentation

- ❑ La conception orientée objet est réalisée suivant deux aspects fondamentaux:
  - Les patterns de structuration qui apportent des solutions aux problèmes de structuration des données et des objets.
  - Les patterns de comportement qui fournissent des solutions pour distribuer les traitements et les algorithmes entre les objets.

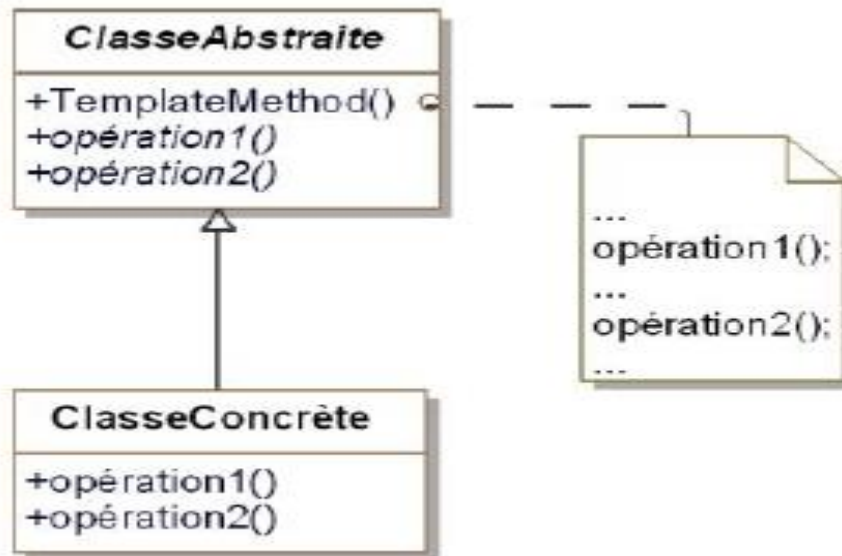
Ces patterns organisent les objets ainsi que leurs interactions en spécifiant les flux de contrôle et de traitement au sein d'un système d'objets.



# Distribution par héritage ou par délégation

- ❑ Une première approche pour distribuer un traitement est de le répartir dans les sousclasses:
  - Cette répartition se fait par l'utilisation dans la classe de méthodes abstraites qui sont implantées dans les sousclasses.
  - Comme une classe peut posséder plusieurs sousclasses, cette approche autorise la possibilité d'obtenir des variantes des parties décrites dans les sousclasses.

Cette possibilité est mise en œuvre par le pattern Template Method comme l'illustre ci-après

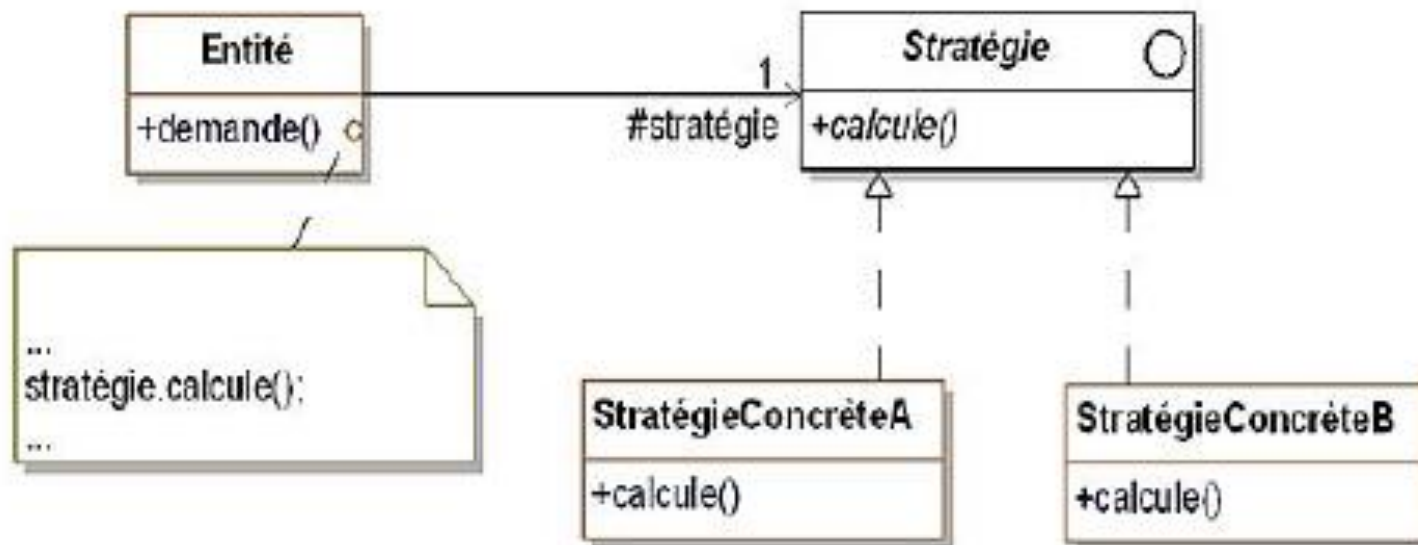


# Distribution par héritage ou par délégation (1)

- ❑ Une seconde possibilité de répartition est mise en œuvre par la distribution des traitements dans des objets dont les classes sont indépendantes.
- ❑ Dans cette approche, un ensemble d'objets coopérant entre eux concourent à la réalisation d'un traitement ou d'un algorithme.

Le pattern Strategy illustre ce mécanisme à la figure suivante.

- La méthode demande de la classe Entité invoque pour la réalisation de son traitement la méthode calcule spécifiée par l'interface Stratégie. Il convient de noter que cette dernière peut avoir plusieurs implantations.



## Distribution par héritage ou par délégation (2)

- ❑ Le tableau suivant indique pour chaque pattern de comportement le type de répartition utilisé.

Pattern de comportement	Type de Répartition
Chain of responsibility	Délégation
Command	Délégation
Interpreter	Héritage
Iterator	Délégation
Mediator	Délégation
Memento	Délégation
Observer	Délégation
State	Délégation
Strategy	Délégation
Template Method	Héritage
Visitor	Délégation

# Chain of Responsibility

## Description

- ❑ Le pattern Chain of Responsibility construit une chaîne d'objets telle que si un objet de la chaîne ne peut pas répondre à une requête, il puisse la transmettre à son successeur et ainsi de suite jusqu'à ce que l'un des objets de la chaîne y réponde.

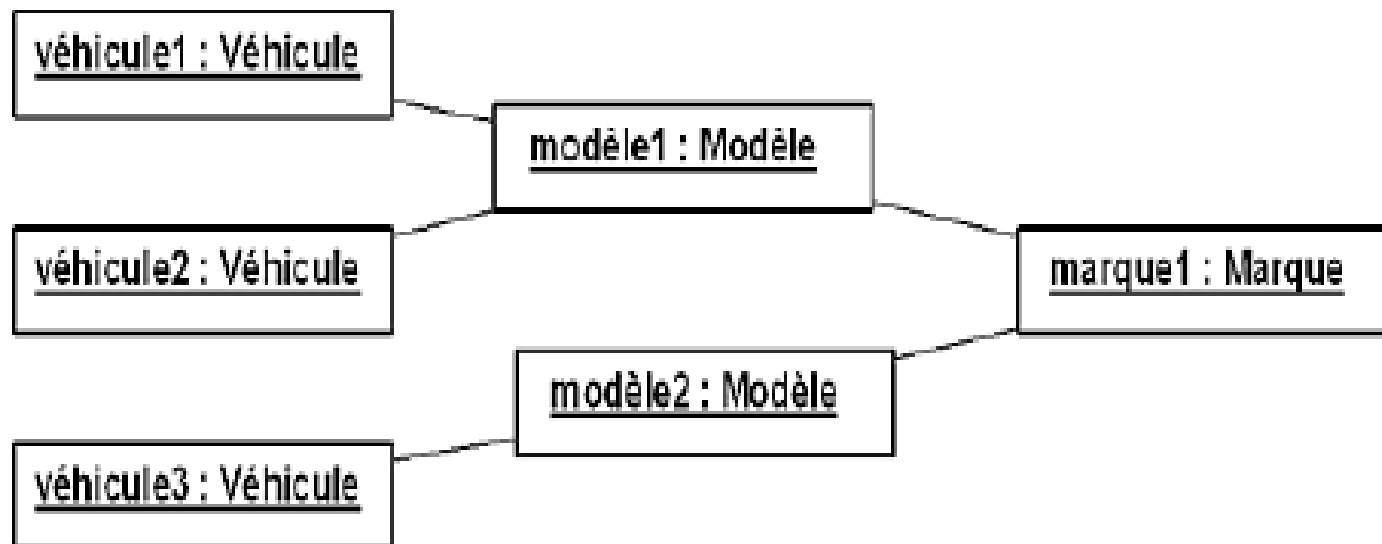
### Exemple :

- Nous nous plaçons dans le cadre de la vente de véhicules d'occasion.
- Lorsque le catalogue de ces véhicules est affiché, l'utilisateur peut demander une description de l'un des véhicules mis en vente.
  - Si une telle description n'a pas été fournie, le système doit alors renvoyer la description associée au modèle de ce véhicule.
  - Si à nouveau, cette description n'a pas été fournie, il convient de renvoyer la description associée à la marque du véhicule.
  - Une description par défaut est renvoyée s'il n'y a pas non plus de description associée à la marque.
- Ainsi, l'utilisateur reçoit la description la plus précise qui est disponible dans le système.

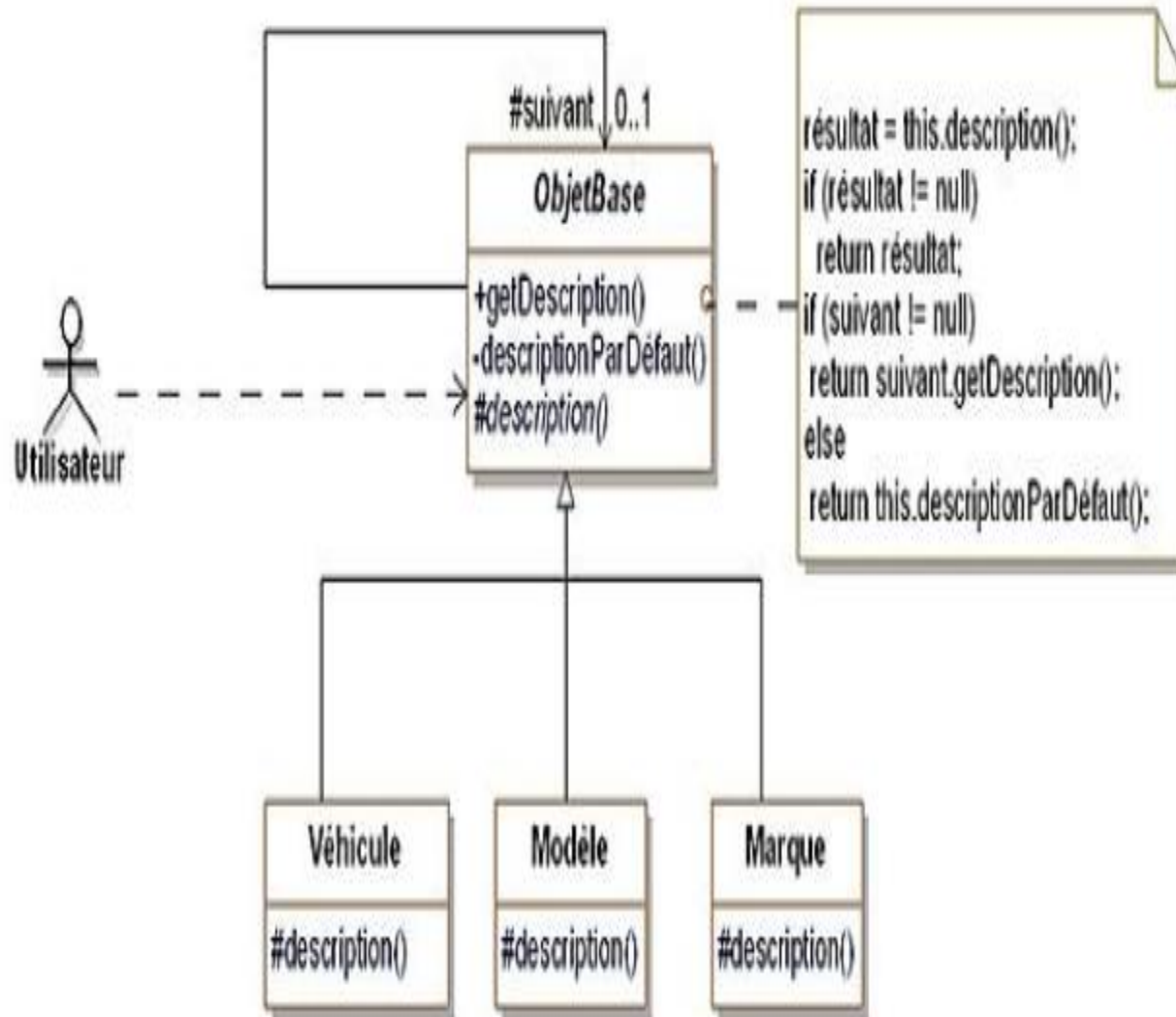
## Solution à ce Problème

- Le pattern Chain of Responsibility fournit une solution pour mettre en œuvre ce mécanisme.
- Celle-ci consiste à lier les objets entre eux du plus spécifique (le véhicule) au plus général (la marque) pour former la chaîne de responsabilité.
- La requête de description est transmise le long de cette chaîne jusqu'à ce qu'un objet puisse la traiter et renvoyer la description.

Le diagramme d'objets UML illustre ce mécanisme et montre les différentes chaînes de responsabilité (de la gauche vers la droite).

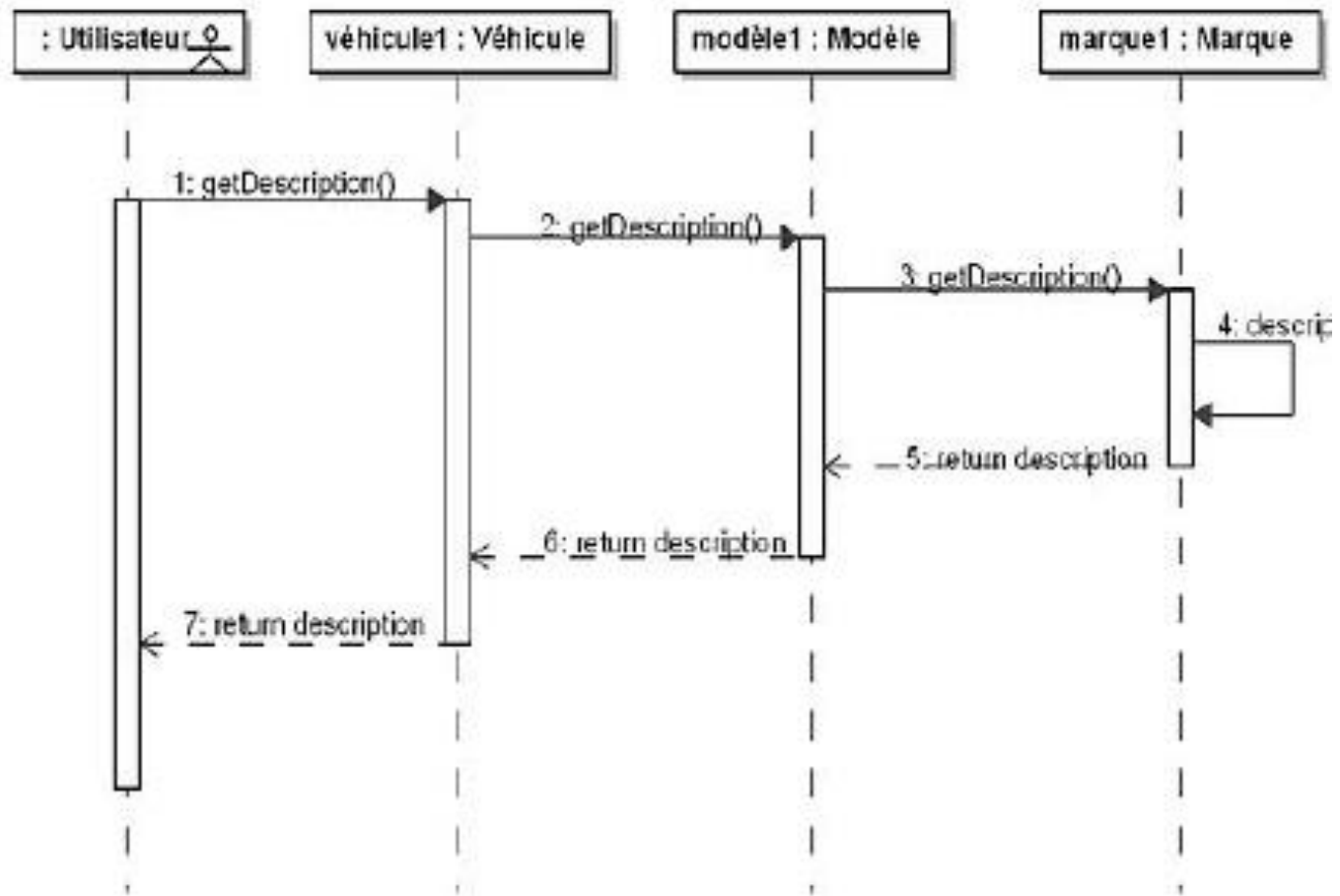


## Le pattern Chain of Responsibility pour organiser la description de véhicules d'occasion

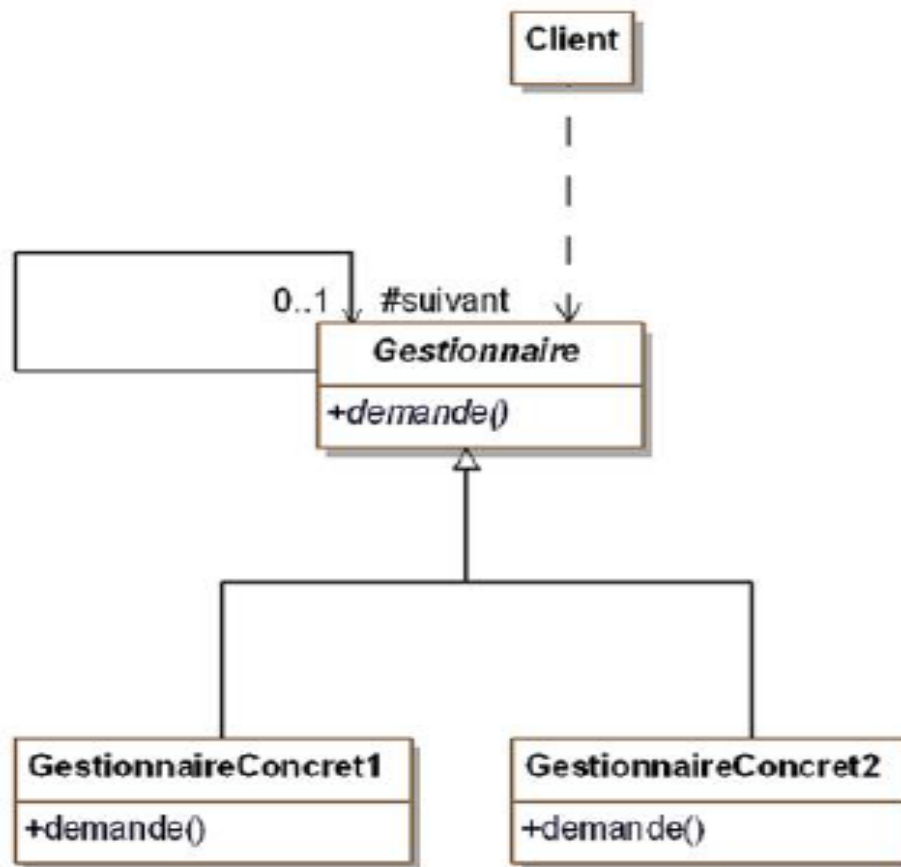


- ❑ Cette figure représente le diagramme des classes du pattern Chain of Responsibility appliqué à l'exemple.
- ❑ Les véhicules, modèles et marques sont décrits par des sousclasses concrètes de la classe ObjetBase.
- ❑ Cette classe abstraite introduit l'association suivant qui implante la chaîne de responsabilité. Elle introduit également trois méthodes :
  - ✓ description est une méthode abstraite. Elle est implantée dans les sousclasses concrètes. Cette implantation doit retourner soit la description si elle existe soit la valeur null dans le cas contraire ;
  - ✓ descriptionParDéfaut retourne une valeur de description par défaut, valable pour tous les véhicules du catalogue ;
  - ✓ getDescription est la méthode publique destinée à l'utilisateur. Elle invoque la méthode description. Si le résultat est null, alors s'il y a un objet suivant, sa méthode getDescription est invoquée à son tour sinon c'est la méthode descriptionParDéfaut qui est utilisée.

- ❑ La figure ci-dessus montre un diagramme de séquence qui est un exemple de requête d'une description basée sur le diagramme d'objets de l'exemple.
- ❑ Dans cet exemple, ni le véhicule1, ni le modèle1 ne possèdent de description. Seule marque1 possède une description qui est donc utilisée pour le véhicule1.







## Participants

- ❑ Gestionnaire (ObjetBase) est une classe abstraite qui implante sous forme d'une association la chaîne de responsabilité ainsi que l'interface des requêtes ;
- ❑ GestionnaireConcret1 et GestionnaireConcret2 (Véhicule, Modèle et Marque) sont les classes concrètes qui implantent le traitement des requêtes en utilisant la chaîne de responsabilité si elles ne peuvent pas les traiter ;
- ❑ Client (Utilisateur) initie la requête initiale auprès d'un objet de l'une des classes GestionnaireConcret1 ou GestionnaireConcret2.

## Collaborations

- ❑ Le client effectue la requête initiale auprès d'un gestionnaire.
- ❑ Cette requête est propagée le long de la chaîne de responsabilité jusqu'au moment où l'un des gestionnaires la traite.

## Domaines d'application

- ❑ une chaîne d'objets gère une requête selon un ordre qui est défini dynamiquement ;
- ❑ la façon dont une chaîne d'objets gère une requête ne doit pas être connue de ses clients.

## IV. Conception de systèmes d'information orientés Web (ou Applications Web) :

- Buts de conception
- Pyramide de Conception :
- Conception de l'interface
- Conception de l'aspect esthétique
- Conception du contenu
- Conception de la navigation
- Conception de l'architecture
- Conception de composant

Roger S. Pressman, *Boca Raton*, David Lowe: Web Engineering A PRACTITIONER'S APPROACH. Edition McGraw-Hill , 2009

# IV. Conception de systèmes d'information orientés Web (ou Applications Web)

## Buts de conception

❑ Jean Kaiser [2002] a suggéré un ensemble d'objectifs de conception design applicables pratiquement à toutes les applications Web, quel que soit le domaine d'application (taille ou complexité:

✓ **Simplicité** : Le **contenu** devrait être informatif mais succinct en utilisant par exemple ( texte, graphiques, vidéo, audio) approprié à l'information livré.

**L'esthétique** devrait être agréable, mais pas écrasant (par exemple, utiliser trop de couleurs pour distraire l'utilisateur plutôt qu'à améliorer l'interaction),

et **l'architecture** devrait atteindre les objectifs de la WebApp de la manière la plus simple possible.

**La navigation** devrait être simple, et les mécanismes de navigation devraient être intuitivement évidents pour l'utilisateur final.

**Les fonctions** devraient être facile à utiliser et plus facile à comprendre.

# IV. Conception de systèmes d'information orientés Web (ou Applications Web)

## Buts de conception

✓ **Cohérence:** Cet objectif de conception s'applique pratiquement à tous les éléments du modèle de la conception.

**Le contenu** doit être construit de manière cohérente (par exemple, le formatage du texte et les styles de police doivent être les mêmes pour tous les documents texte et l'art graphique doit avoir un aspect cohérent, un schéma de couleurs et un style).

**La Conception graphique** (esthétique) devrait présenter un regard cohérent dans toutes les parties de la WebApp.

**La conception architecturale** devrait établir des modèles qui conduisent à une structure de navigation hypermédia.

**La conception de l'interface** doit définir des modes d'interaction, de navigation et d'affichage du contenu cohérents.

**Les mécanismes de navigation** doivent être utilisés de manière cohérente dans tous les éléments de l'application Web.

# IV. Conception de systèmes d'information orientés Web (ou Applications Web)

## Buts de conception

- ✓ **Identité.** L'esthétique, l'interface et la conception de navigation d'une AppWeb doivent être cohérents avec le domaine d'application pour lequel il doit être construit.
  - Un site Web d'une chaîne de télévision a sans doute un look différent à celui d'une société de services financiers.
  - L'architecture d'AppWeb sera entièrement différente, les interfaces seront construites pour accueillir différentes catégories d'utilisateurs, et la navigation sera organisée pour atteindre différents objectifs.
  - Un ingénieur Web (et d'autres contributeurs de conception) devrait travailler pour établir une identité pour l'application Web à travers la conception.

# IV. Conception de systèmes d'information orientés Web (ou Applications Web)

## Buts de conception

- ✓ **Robustesse** : Basé sur l'identité qui a été établie, une AppWeb fait souvent une promesse implicite à un utilisateur. L'utilisateur s'attend à du **contenu robuste** et les **fonctions** qui correspondent aux besoins de l'utilisateur.

Si ces éléments sont manquants ou insuffisants, il est probable que la AppWeb échouera.

- ✓ **Navigabilité**. Nous avons déjà noté que la navigation devrait être simple et cohérent. Il devrait également être conçu de manière intuitive et prévisible pour les utilisateurs. Autrement dit, les utilisateurs devraient être en mesure de comprendre comment déplacer à propos de la AppWeb sans avoir à rechercher des liens de navigation ou instructions.

**Par exemple**, si une page contient un champ d'icônes graphiques ou d'images, dont certains peuvent être utilisés comme mécanismes de navigation, ceux-ci doivent être identifiés de manière évidente. Rien n'est plus frustrant que d'essayer pour trouver le lien direct approprié parmi de nombreuses images graphiques.

# IV. Conception de systèmes d'information orientés Web (ou Applications Web)

## Buts de conception

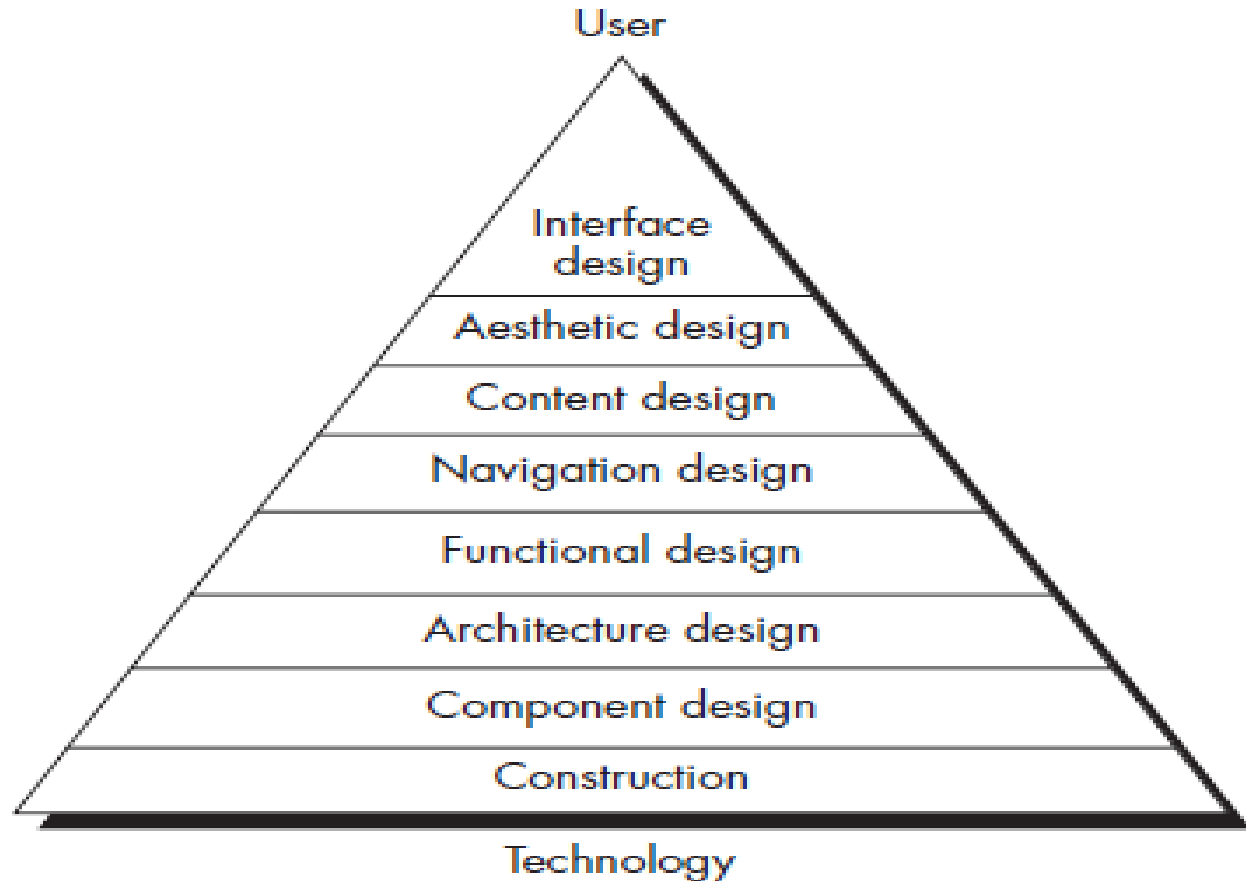
- ✓ **Attrait visuel.** Dans toutes les catégories de logiciels, les applications Web sont incontestablement les plus visuelles, les plus dynamiques et les plus élégantes (sur le plan esthétique). L'attrait visuel est sans aucun doute dans l'œil du spectateur, mais de nombreuses caractéristiques de conception (par exemple, l'aspect et la convivialité du contenu, la disposition de l'interface, la coordination des couleurs, l'équilibre du texte, des graphiques et d'autres médias, et les mécanismes de navigation) contribuent à l'attrait visuel.
- ✓ **Compatibilité.** La plupart des AppWeb sont utilisées dans divers environnements (par exemple, différents matériels, types de connexion Internet, systèmes d'exploitation et navigateurs) et doivent être conçus pour être compatibles avec chacun.

Une conception qui atteint chacune ces objectifs sera agréable à l'utilisateur final et apportera de l'assurance et la confiance à l'équipe WebE.ng



## IV. Conception de systèmes d'information orientés Web (ou Applications Web) :

- Pyramide de Conception :



## IV. Conception de systèmes d'information orientés Web (ou Applications Web) :

### ○ Pyramide de Conception :

- ❑ Cette Figure illustre une pyramide de conception contenant les actions de conception (niveaux) qui se produisent comme un modèle de conception complet.
- ❑ Les deux premiers niveaux de la conception de l'interface pyramidale et la conception esthétique forment la base de l'interaction de l'utilisateur avec le système, et peuvent donc être regroupés en tant que conception d'interaction.
  - **La conception de l'interface** décrit la structure et l'organisation de l'interface utilisateur. Il comprend une représentation de la disposition de l'écran, une définition des modes d'interaction, et une description des mécanismes de navigation. Pour les sites avec fonctionnalité plus que simpliste, il peut également inclure des descriptions des séquences de workflows et problèmes connexes.
  - **Le design esthétique**, également appelé design graphique, décrit l'aspect et la convivialité de l'application Web. Il comprend des schémas de couleurs, la disposition géométrique, la taille du texte, la police et l'emplacement, l'utilisation de graphiques et les décisions esthétiques connexes.

## IV. Conception de systèmes d'information orientés Web (ou Applications Web) :

### ○ Pyramide de Conception :

- ❑ Les deux niveaux suivants de la pyramide se concentrent sur l'information: le contenu qui sous-tend AppWeb, la manière dont ce contenu est organisé et la manière dont que l'utilisateur parcourt la AppWeb pour accéder au contenu ou à la fonctionnalité.
  - **La conception de contenu** définit la mise en page, la structure et les grandes lignes de tout le contenu présenté dans le cadre de la AppWeb. Elle établit les relations entre objets de contenu.
  - **La conception de navigation** représente le flux de navigation entre les objets de contenu et toutes les fonctions de la AppWeb. Elle décrit les manières dont l'utilisateur localise et interagit avec le contenu.
- ❑ Les Apps Web ont évolué pour fournir des fonctionnalités importantes en plus d'un large variété de contenu, tous liés au domaine d'application considéré.

## IV. Conception de systèmes d'information orientés Web (ou Applications Web) :

### ○ Pyramide de Conception :

- ❑ Certaines applications Web offrent un large éventail de fonctionnalités (par exemple, elles permettent aux utilisateurs de définir des mises en page d'espace, de placer des commandes de produits, de surveiller une maison via Internet) en plus d'une large collection de contenus (p. descriptions, spécifications, photos).
- ❑ Dans certains cas, la fonctionnalité suivra des modèles bien établis. Par exemple, les applications Web business-to-consumer (B2C) utilisent des catalogues, des carte de crédits, ...
- ❑ Dans d'autres cas, la fonctionnalité peut être hautement spécialisé. Lorsque le design se concentre explicitement sur la fonctionnalité, nous le décrivons de la manière suivante:
  - **La conception fonctionnelle** identifie le comportement global et les fonctionnalités prises en charge par la WebApp incluant des aspects tels que le support de workflow, le contenu, l'adaptation et/ou la personnalisation de l'interface, la saisie d'ordres, la gestion des base de données, les fonctions de calcul, et d'autres

## IV. Conception de systèmes d'information orientés Web (ou Applications Web) :

### ○ Pyramide de Conception :

- ❑ Les niveaux inférieurs de la pyramide de conception - classés en tant que conception technique - examinent comment l'interface, la navigation, le contenu et les fonctions seront intégrés ensemble et misent en œuvre. La conception technique fournit une base pour la construction qui suit. Elle considère à la fois l'architecture WebApp et la conception de Composants WebApp
  - **La conception de l'architecture** identifie la structure globale de la WebApp. Elle identifie les composants conceptuels de la WebApp et la manière dont ceux-ci sont interconnectés. Elle s'assure que les différents aspects de l'application sont correctement intégrés. Elle identifie également les composants techniques et les interactions entre les composants requis pour construire l'application Web.
  - **La conception des composants** développe la logique de traitement détaillée requise pour la mise en œuvre composants fonctionnels prenant en charge un ensemble complet des fonctions de la AppWeb.

## IV. Conception de systèmes d'information orientés Web (ou Applications Web) :

### ○ Pyramide de Conception :

❑ Il convient de clarifier la distinction entre l'architecture conceptuelle et l'architecture technique.

✓ **L'architecture conceptuelle** décrit les principales fonctions et informations nécessaires pour le système.

#### Par exemple :

- ❖ une simple application e-commerce peut nécessiter un catalogue de produits (avec contenu et navigation dans la recherche),
  - ❖ un système d'achat (avec gestion du panier et fonctionnalité de paiement),
  - ❖ un système de gestion de la clientèle (pour suivre les achats des utilisateurs et effectuer des adaptations de site et des recommandations),
  - ❖ et éventuellement un sous-système d'administration (pour la gestion des inscriptions, des demandes d'achat, etc.).
- ✓ Cela se compare à **l'architecture technique** qui montre comment les technologies spécifiques sont utilisées pour implémenter l'application Web.

#### Par exemple:

- ❖ une application de commerce électronique peut inclure un serveur Web, une passerelle de paiement, un système de gestion de contenu et un nombre de pages HTML et de scripts ASP.