- Les classes
- Encapsulation
- Héritage
- Abstraction
- Polymorphisme
- Interface
- Design Pattern

LES CLASSES

INTRODUCTION À L'APPROCHE ORIENTÉ OBJET

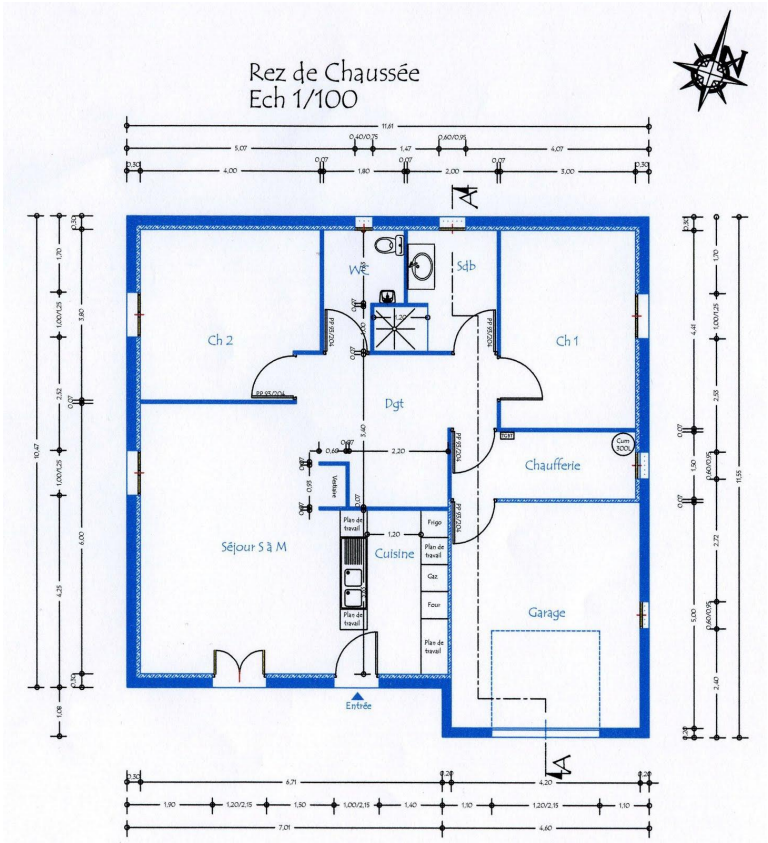
- Notion de classe
- Instance de classe
- Diagramme de classes UML 2.0

NOTION DE CLASSE

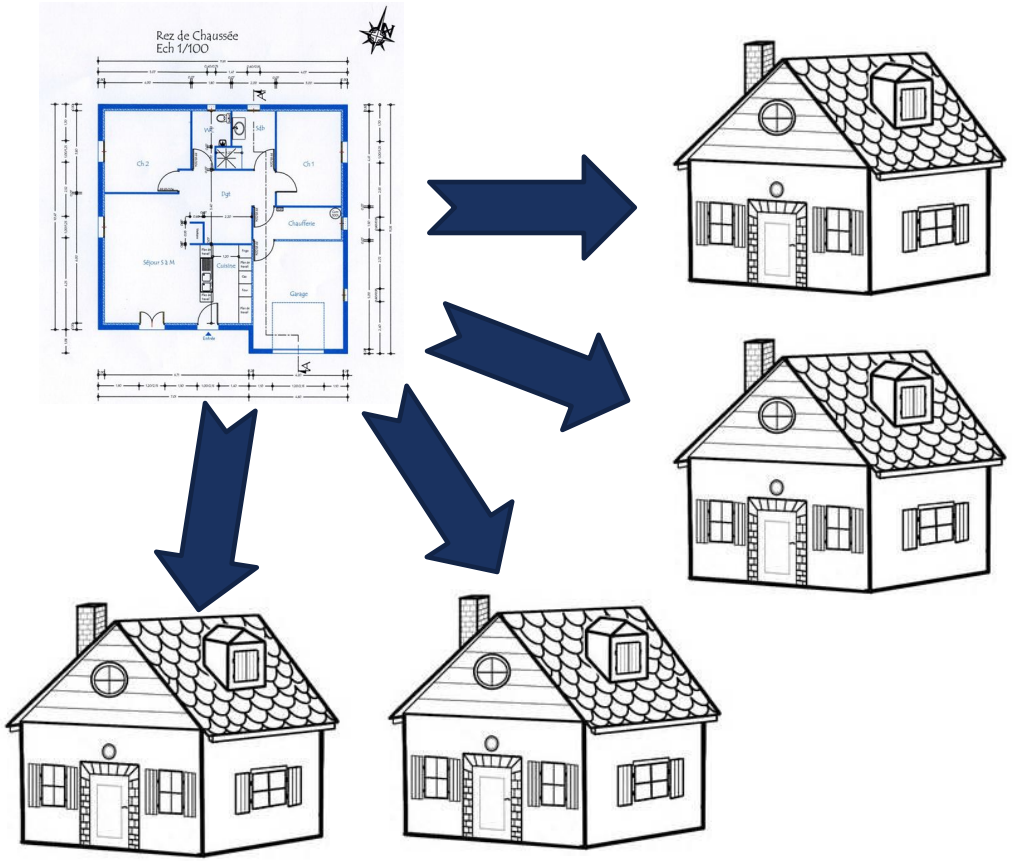
Une classe sert essentiellement à créer de nouveaux types de données au sein de nos programmes. Il s'agit de réaliser un **modèle** à partir duquel il sera possible de créer des objets (on parle d'instanciation).

Chacune, de ces classes, pourra contenir des **membres**, qu'ils s'agissent **d'attributs** appelés aussi « variable membre », de **méthodes** ou de constructeurs voir même de **sous-types**.

Une classe à pour but de définir quelles sont les données et fonctionnalités que chaque instanciation contiendra.



INSTANCE DE CLASSE



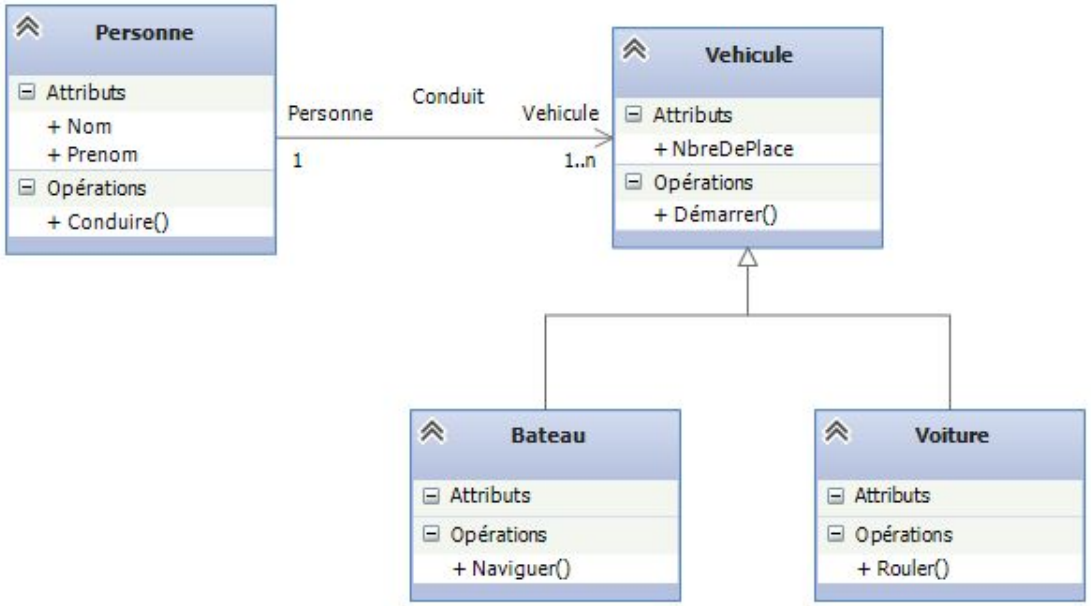
Pour utiliser ces classes, nous devrons le plus souvent les **instancier**.

L'instanciation est le processus de création d'un objet en mémoire possédant toutes les fonctionnalités de la classe dont il est le produit.

DIAGRAMME DE CLASSES UML 2.0

Pour nous aider à représenter nos classes, nous pouvons utiliser un diagramme de classes UML 2.0.

Le diagramme de classes est un schéma utilisé pour présenter les types de données des systèmes ainsi que les différentes relations entre ceux-ci.



Quelques logiciels UML
Argo Uml
Microsoft Office Visio
Visual Paradigm
Star UML
Visual Studio

ENCAPSULATION

INTRODUCTION À L'APPROCHE ORIENTÉ OBJET

- Définition
- Attributs
- Méthodes
- Constructeurs
- Modificateur d'accès
- Accesseurs
- Implémentations UML
- Exercices

ENCAPSULATION

DÉFINITION

L'encapsulation est un principe fondamental au sein de l'orienté objet.

Elle va nous permettre de déclarer les membres, de les protéger et de contrôler leur accessibilité afin de garantir l'intégrité des informations qu'ils contiennent.

Pour gérer l'accessibilité, nous devrons donner à nos membres un **modificateur d'accès** qui limitera l'accès à ceux-ci.

ATTRIBUTS

Une personne peut être caractérisée par :

- Numéro de registre national
- Nom
- Surnom
- Prénom
- Date de naissance
- Sexe
- Est en vie

Les attributs sont utilisés pour stocker des valeurs, on parle également de variables membres.

Ces valeurs sont souvent de types primitifs, c'est-à-dire de type :

- Entier
- Décimal
- Alphanumérique
- Date
- Booléenne
- ...

MÉTHODES

les méthodes décrivent les fonctionnalités comportementales des objets.

Ces méthodes peuvent prendre des valeurs en entrée et/ou retourner une valeur.

Elles ont pour but principal de travailler avec les attributs pour les modifier ou produire des résultats sur base de ceux-ci.

Une personne peut :

- Parler
- Manger
- Dormir
- Pleurer
- Réfléchir

CONSTRUCTEURS

Pour créer une personne il me faut :

- Son numéro de registre national
- Son nom
- Son prénom
- Sa date de naissance
- Son sexe

Les constructeurs sont des méthodes particulières qui ne sont utilisées qu'au moment de l'instanciation.

Ils servent le plus souvent à initialiser les variables membres à la création de nos objets.

En règle générale, ces méthodes portent le nom de la classe et n'ont pas de type de retour spécifié*.

*Peut varier en fonction des langages de programmation

MODIFICATEUR D'ACCÈS

Les modificateurs d'accès sont utilisés pour limiter l'accès des membres par rapport à leur environnement.

Nous retrouvons le plus souvent 4 niveaux d'accès pour nos membres :

- Public
- Private
- Protected
- Package

Accessibilité	Signification
public	L'accès n'est pas limité.
protected	L'accès est restreint à la classe conteneur ou à ses types dérivés. (Notion d'héritage)
private	L'accès est restreint au type conteneur
package	L'accès est restreint au package courant

Peut varier en fonction des langages de programmation

ACCESSEURS

Les accesseurs pour personnes

- Numéro de registre national
  - Getter public
  - Setter private
- Nom
  - Getter public
  - Setter private
- Surnom
  - Getter public
  - Setter public
- …

Dans la bonne pratique, nos **attributs** se verront attribuer un modificateur d'accès **« private »**.

La raison est principalement fonctionnelle, une variable est un conteneur où nous pourrons lire la valeur ou l'affecter directement.

Cependant, nous n'avons aucun contrôle sur la valeur affectée ni sur la valeur retournée.

De plus, le modificateur d'accès de la variable gère à la fois l'accès à la récupération et l'accès à l'affectation de manière indissociable.

Pour résoudre cela, nous utiliserons des accesseurs qui ne sont rien d'autre que des méthodes dédiées à la gestion des attributs.

ACCESSEURS

Les accesseurs pour personnes

- Numéro de registre national
  - Getter public
  - Setter private
- Nom
  - Getter public
  - Setter private
- Surnom
  - Getter public
  - Setter public
- …

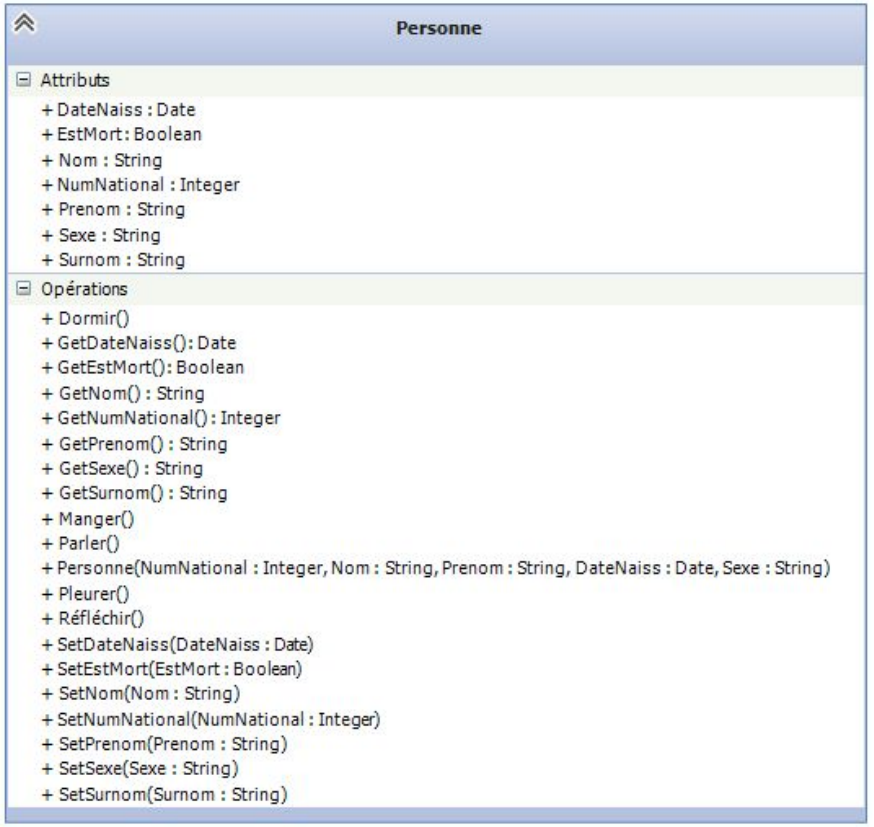
Nous retrouvons deux types d'accesseurs :

- **Getter** : Retourne la valeur de l'attribut.
- **Setter** : Affecte la valeur à l'attribut.

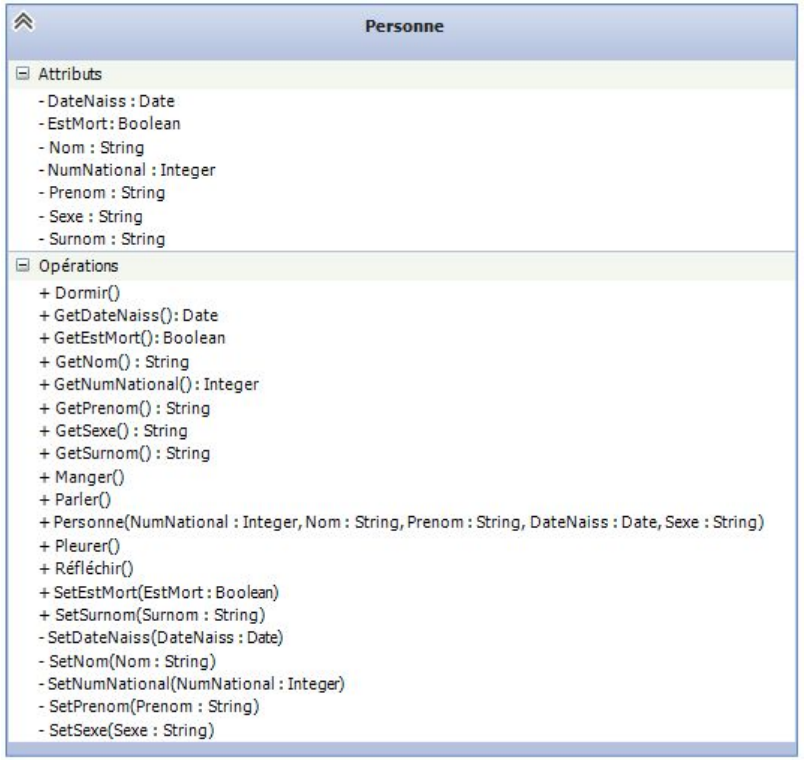
IMPLÉMENTATION UML

Pour nous représenter nos classes et les relations qu'elles ont entre elles, nous pouvons utiliser les diagrammes de classes UML.

Ces derniers nous permettent de schématiser rapidement et simplement nos classes.



IMPLÉMENTATION UML



Pour les modificateurs d'accès, nous devons l'indiquer avec un symbole qui précède le membre.

Accessibilité	Symbole
public	+ (Plus)
protected	# (Dièse)
private	- (Moins)
package	~ (Tilde)

EXERCICES

- Déterminer sur papier les attributs, les accesseurs, les méthodes, le(s) constructeur(s) et les modificateurs d'accès pour chacun d'entre eux afin de représenter un compte bancaire type compte courant.
- Définir la classe à l'aide d'un diagramme de classe

HÉRITAGE

INTRODUCTION À L'APPROCHE ORIENTÉ OBJET

- Notion d'héritage
- Héritage multiple
- Héritage simple
- Implémentation UML
- Exercices

HÉRITAGE

NOTION D'HÉRITAGE

Toujours dans l'optique de la maintenabilité et la réutilisation, l'orienté objet utilise un deuxième concept fondamental : « L'héritage ».

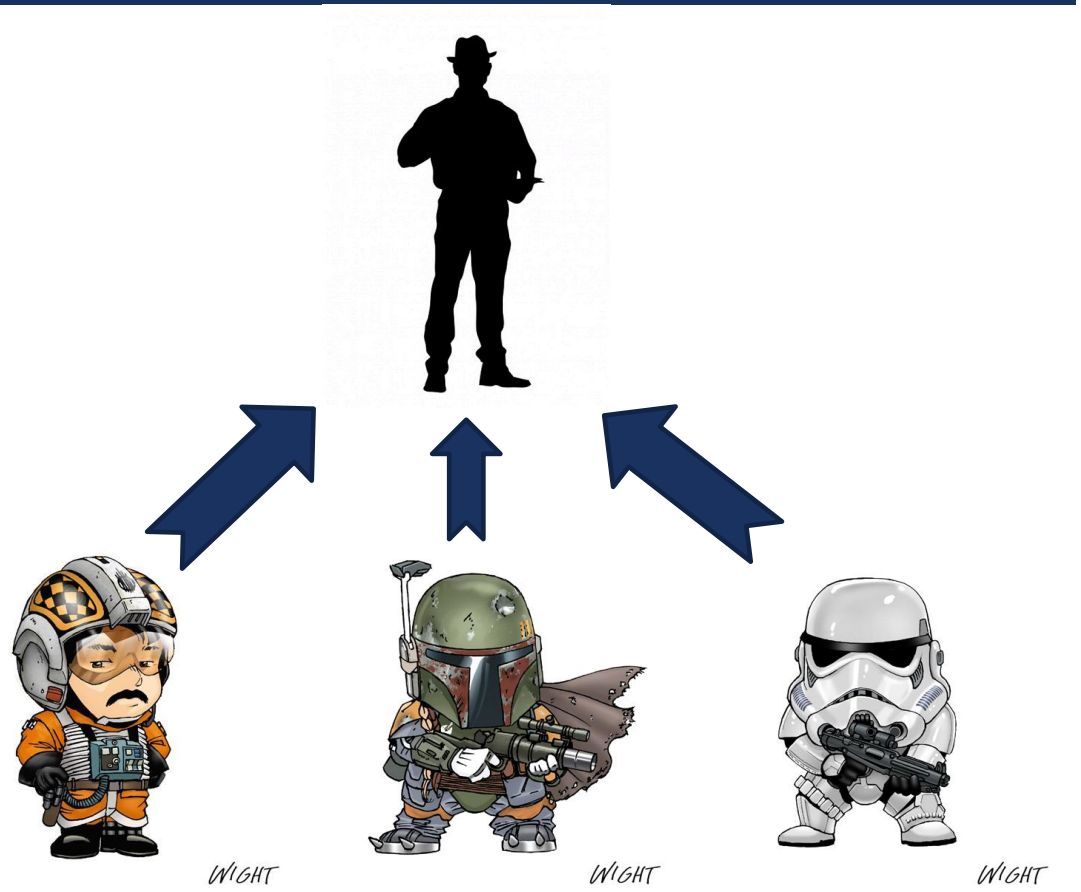
Lorsque plusieurs classes d'une même famille possèdent des fonctionnalités connexes, nous avons tout intérêt à utiliser l'héritage.

Le principe est de déclarer une classe, appelée classe parent, regroupant ces fonctionnalités et de spécifier que nos classes, qui deviennent des classes enfants, héritent de cette « superclasse ».

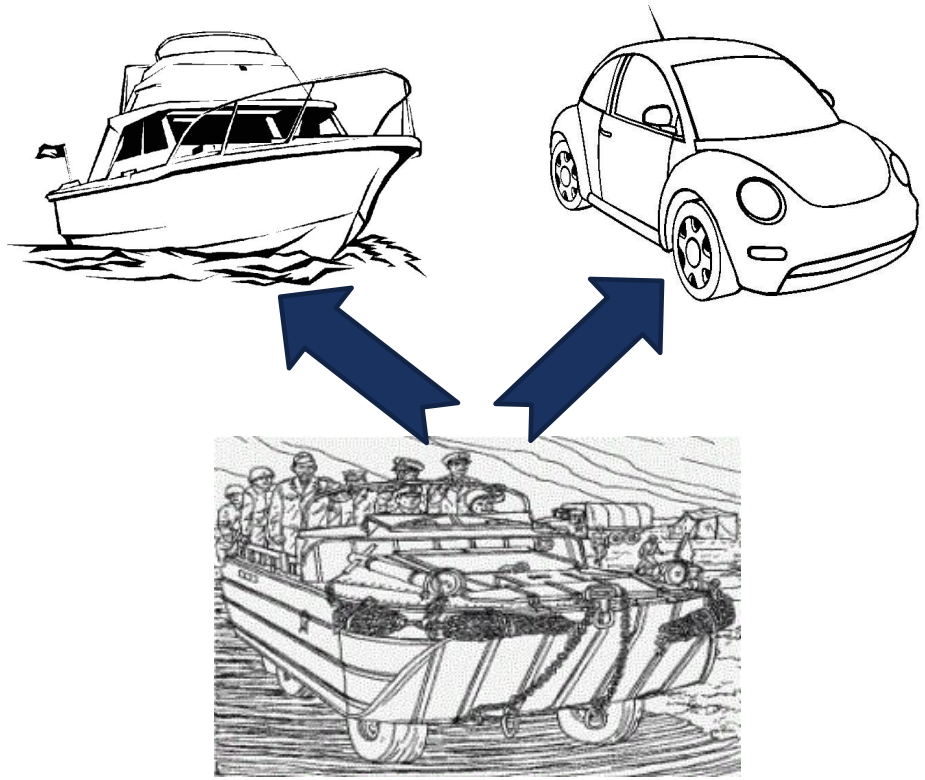
Cependant, il est important de conserver en tête les concepts d'encapsulation.

Par héritage on hérite de tout ce qui vient du parent, excepté les membres privés à celle-ci.

On peut résumer l'héritage par une association « est un(e) »



HÉRITAGE MULTIPLE



Dans le cadre de l'orienté objet, ils nous est permis de faire de l'héritage multiple. C'est-à-dire que nous avons la possibilité **d'hériter de plusieurs classes parents**.

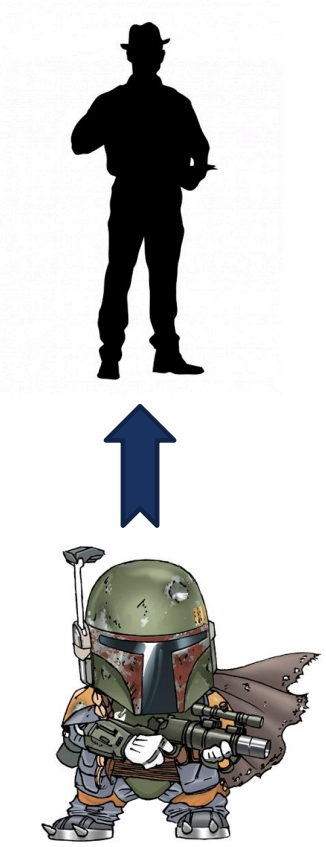
Cette technique permet de regrouper au sein d'une seule et même classe les attributs et méthodes de plusieurs classes parents.

HÉRITAGE SIMPLE

Cependant, développer en tenant compte de l'héritage multiple peut s'avérer rapidement être ardus.

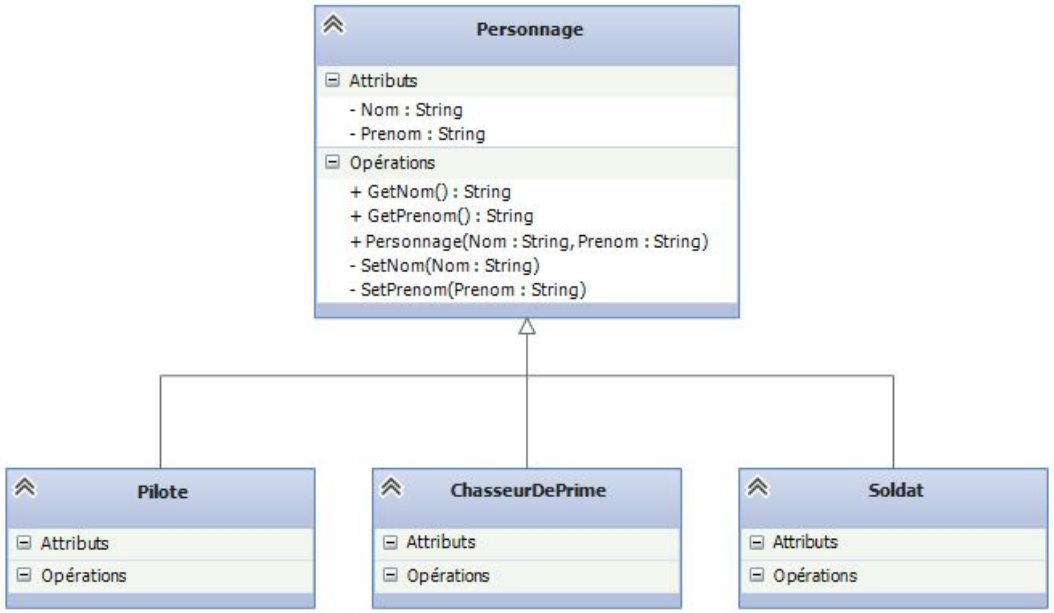
Par conséquent, les concepteurs de certains langages de programmation, comme le C# ou le Java, ont prit la décision de limiter l'héritage à une seule classe à la fois.

On parle donc d'héritage simple.

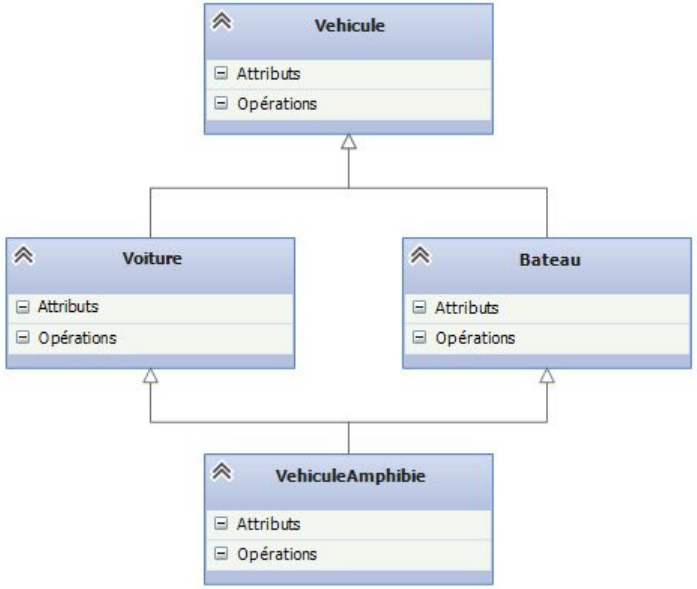


IMPLÉMENTATION UML

Héritage simple



Héritage multiple



EXERCICES

- Déterminer sur papier les attributs, les accesseurs, les méthodes, le(s) constructeur(s) et les modificateurs d'accès pour chacun d'entre eux afin de représenter un compte bancaire type livret d'épargne.
- Déterminer sur papier, sur base de l'exercice lié à l'encapsulation, s'il y a lieu de faire de l'héritage et l'implémenter
- Définir les classes à l'aide d'un diagramme de classe

POLYMORPHISME

INTRODUCTION À L'APPROCHE ORIENTÉ OBJET

- Définition
- Le polymorphisme paramétrique
- Le polymorphisme Ad hoc
- Le polymorphisme d'héritage

DÉFINITION

Polymorphisme impactant les membres
Le polymorphisme paramétrique
Le polymorphisme Ad Hoc

Polymorphisme impactant les types
Le polymorphisme d'héritage

Le mot polymorphisme est formé à partir du grec ancien πολύς (polús) qui signifie « nombreux » et μορφή (morphê) qui signifie « forme ».

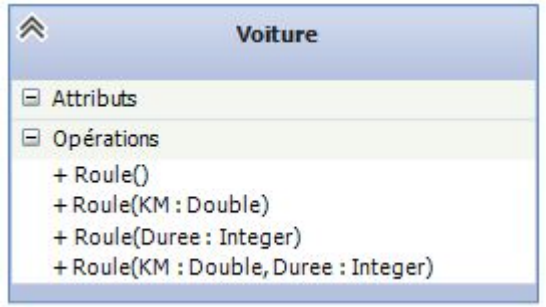
Dans le cadre de l'orienté objet, nous retrouvons trois types de polymorphismes dont deux qui impactent les membres et un qui concerne les types.

POLYMORPHISME PARAMÉTRIQUE

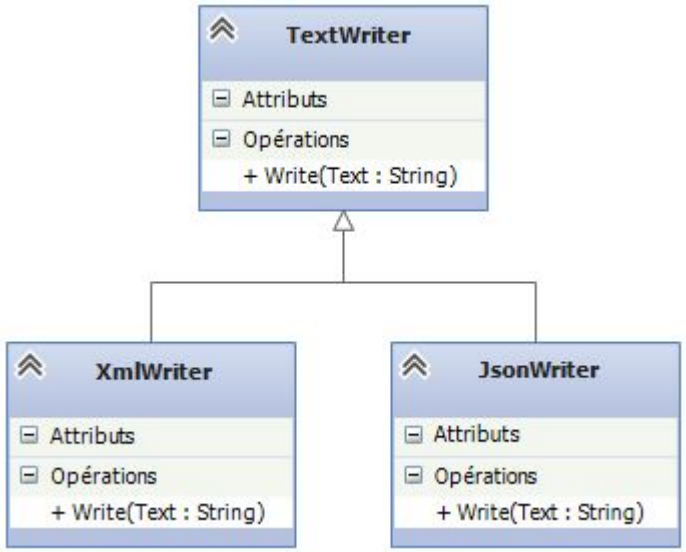
Appelé communément, **surcharge de méthodes**, ce type de polymorphisme s'applique sur la signature de méthodes.

La **signature d'une méthode** reprend le **nom** et les **paramètres** qu'elle reçoit en entrée.

Cela nous permet de déclarer plusieurs méthodes portant le même nom à condition qu'elles reçoivent des paramètres en nombre ou de types différents.



POLYMORPHISME AD HOC



Le polymorphisme « Ad hoc » quant à lui est souvent appelé **redéfinition de méthodes**.

Il permet de redéfinir le fonctionnement des méthodes des classes dérivées par rapport au fonctionnement de la classe parent.

Ces méthodes ont pour particularité d'avoir la **même signature**.

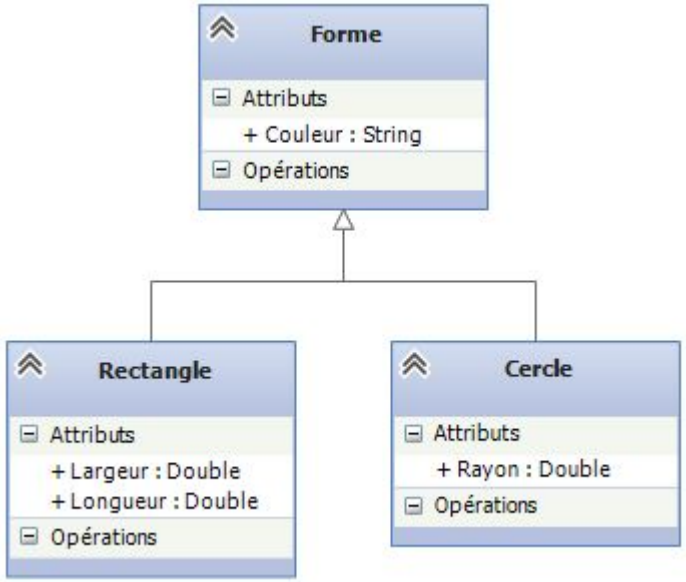
POLYMORPHISME D'HÉRITAGE

Le polymorphisme d'héritage utilise les types eux-mêmes.

En effet une valeur peut être utilisée avec son propre type ou tout type de son(es) parent(s).

Par exemple, nous pourrions déclarer une variable de type « Rectangle » ou « Cercle » et la stocker dans une variable de type « Forme ».

Il faut cependant garder en tête que la variable déclarée sera de type « Forme » et ne nous fournira, au final, que les fonctionnalités déclarées dans la classe « Forme ».



EXERCICES

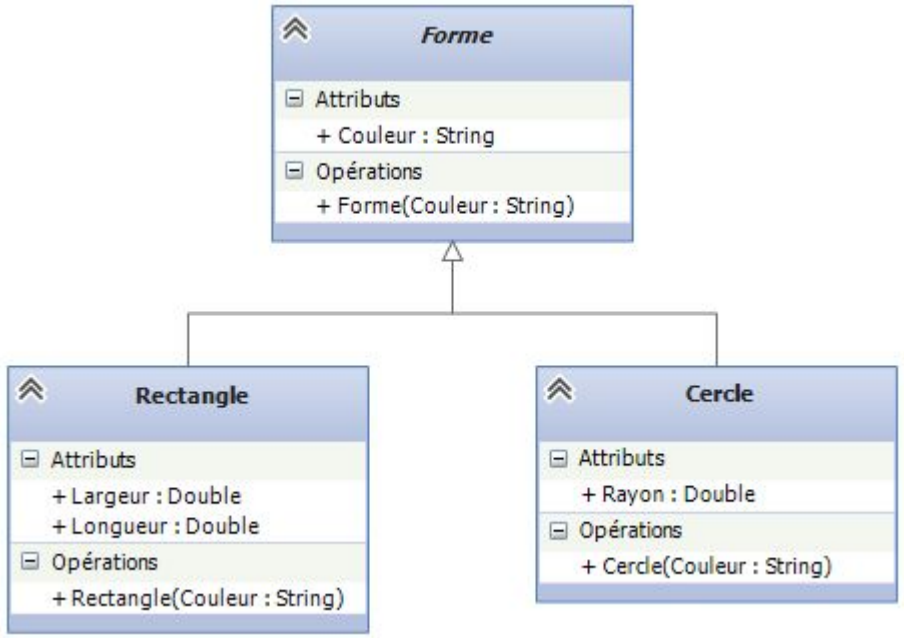
- Définir le modificateur d'accès de la méthode SetSolde à private dans la classe Compte
- Déterminer les problèmes que cela pose
- Si besoin, adapter la solution pour que cela puisse fonctionner à nouveau tout en laissant la méthode SetSolde en private

ABSTRACTION

INTRODUCTION À L'APPROCHE ORIENTÉ OBJET

- Classe abstraite
- Membre abstrait
- Exercices

CLASSE ABSTRAITE



Une classe abstraite est une classe qui est définie, non pas pour être instanciée, mais pour regrouper des fonctionnalités connexes.

De ce fait, elle ne peut servir qu'à l'héritage et ne peut donc pas être instanciée.

Cependant, cette dernière pouvant contenir des attributs, elle peut contenir des constructeurs d'instance.

MEMBRE ABSTRAIT

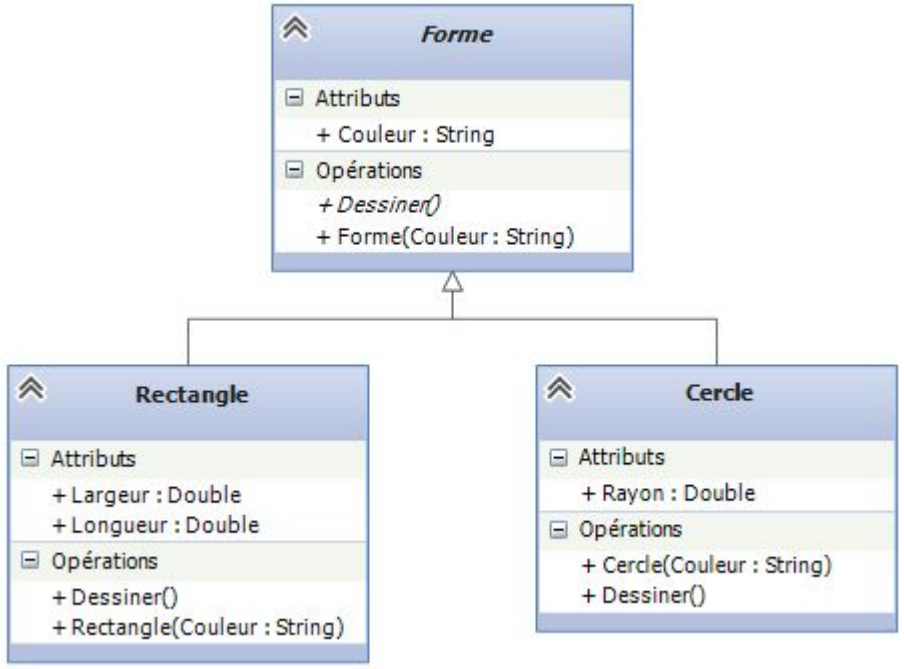
Définir un membre abstrait permet de déclarer une fonctionnalité dans la classe parent sans en définir le fonctionnel.

Cependant, cette fonctionnalité devra être redéfinie, au plus tard, dans la première classe enfant non abstraite.

De plus, dans cette optique, il sera interdit de déclarer un membre abstrait avec le niveau d'accessibilité private.

Enfin, si une classe déclare un membre abstrait, cette classe doit être également abstraite.

Attention que l'inverse n'est pas vrai.



EXERCICES

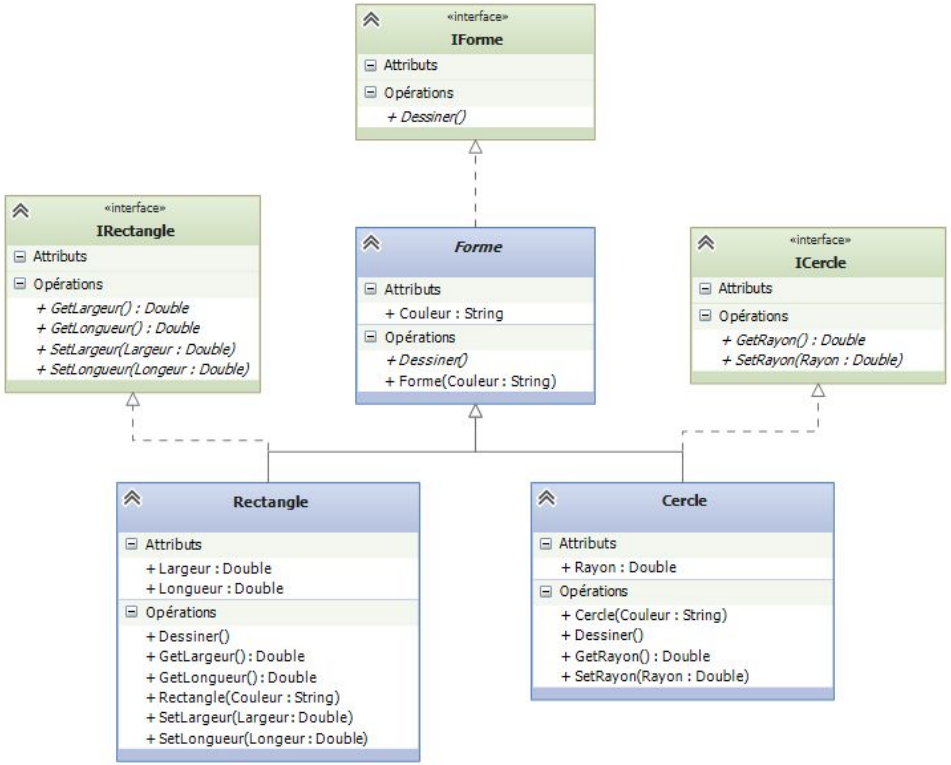
- Définir une méthode abstraite « + AppliquerInteret() » dans la classe compte.
- Corriger le diagramme de classes en conséquence.

INTERFACE

INTRODUCTION À L'APPROCHE ORIENTÉ OBJET

- Définition et contrainte
- Interface et polymorphisme
- Classes abstraites vs Interfaces

DÉFINITION ET CONTRAINTE



Les interfaces décrivent un groupe de fonctionnalités communes qui peuvent appartenir à n'importe quelle classe.

Les interfaces sont composées de méthodes* qui ne contiennent pas de fonctionnel.

D'ailleurs, on les compare souvent à un contrat.

Lorsque nous disons qu'une classe implémente une interface, cela signifie que la classe s'engage à implémenter tous les membres définis par l'interface avec le modificateur d'accès public.

Elles sont également utiles, pour les langages qui n'implémente pas l'héritage multiple.

* Peut varier en fonction des langages de programmation

INTERFACE ET POLYMORPHISME

Dans un contrat, nous avons des obligations mais nous en retirons le plus souvent un certain bénéfice.

Lorsque nous implémentons une interface nous sommes tenu d'implémenter les fonctionnalités de la dite interface.

Mais d'un autre côté, nous y gagnons un nouveau type. Ce nouveau type sera donc utilisable avec le principe de polymorphisme d'héritage.

Il est bon à rappeler, que l'utilisation du polymorphisme d'héritage limite aux fonctionnalités du type de la variable.

CLASSES ABSTRAITES VS INTERFACES

Reste à ne pas confondre interfaces et classes abstraites!!

Car bien que pouvant être très proche dans le concept d'implémentation, plusieurs notions différencie l'utilisation des classes abstraites par rapport aux interfaces.

Cas	Interface	Classe abstraite
Les classes peuvent Hériter/implémenter plusieurs …	OUI	Dépend du langage de programmation
Peut contenir des variables membres	NON	OUI
Peut contenir des blocs d'instructions	NON	OUI
Peut contenir des membres private, protected ou package	NON	OUI
Peut contenir des classes imbriquées	NON	OUI
…	…	…

DESIGN PATTERN

INTRODUCTION À L'APPROCHE ORIENTÉ OBJET

- Introduction
- Composite
- Observer
- Abstract Factory

INTRODUCTION

Ils ont été introduit la première fois en 1995 par le « Gang of Four » à travers leur livre intitulé :
'Design Patterns - Elements of Reusable Object-Oriented Software'.

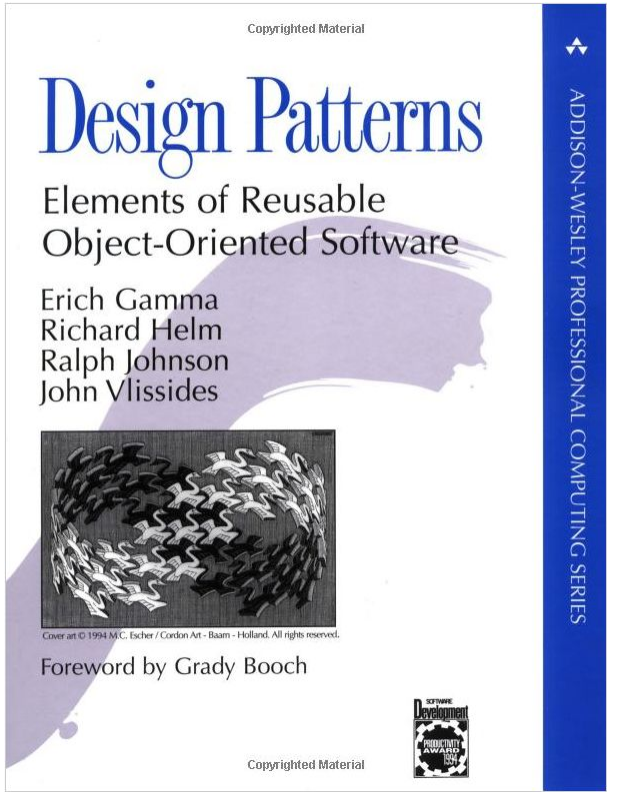
Le 'GoF' est constitué de 4 auteurs : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides.

Un « Design Pattern » est techno agnostique et consiste à solutionner un problème clairement défini que nous rencontrons régulièrement dans notre développement.

Ceux-ci sont constitués d'objets, définis par des classes et des interfaces, reliés entre eux par les concepts fondamentaux de l'orienté objet, le tout en respectant les « bonnes » pratiques de programmation.

Il va de soit que ceux-ci devront être adaptés en fonction de différents critères comme le langage, la situation, nos besoins.

Exemple le plus parlant : C++ gère l'héritage multiple tandis que le C# non, de ce fait nous implémenterons différemment le même pattern.



INTRODUCTION

Les modèles de conception du « Gang of Four » reprennent trois axes spécifiques.

Orienté « Création » :

- Fabrique abstraite (Abstract Factory)
- Monteur (Builder)
- Fabrique (Factory Method)
- Prototype (Prototype)
- Singleton (Singleton)

INTRODUCTION

Orienté « Structure » :

- Adaptateur (Adapter)
- Pont (Bridge)
- Objet composite (Composite)
- Décorateur (Decorator)
- Façade (Facade)
- Poids-mouche ou poids-plume (Flyweight)
- Proxy (Proxy)

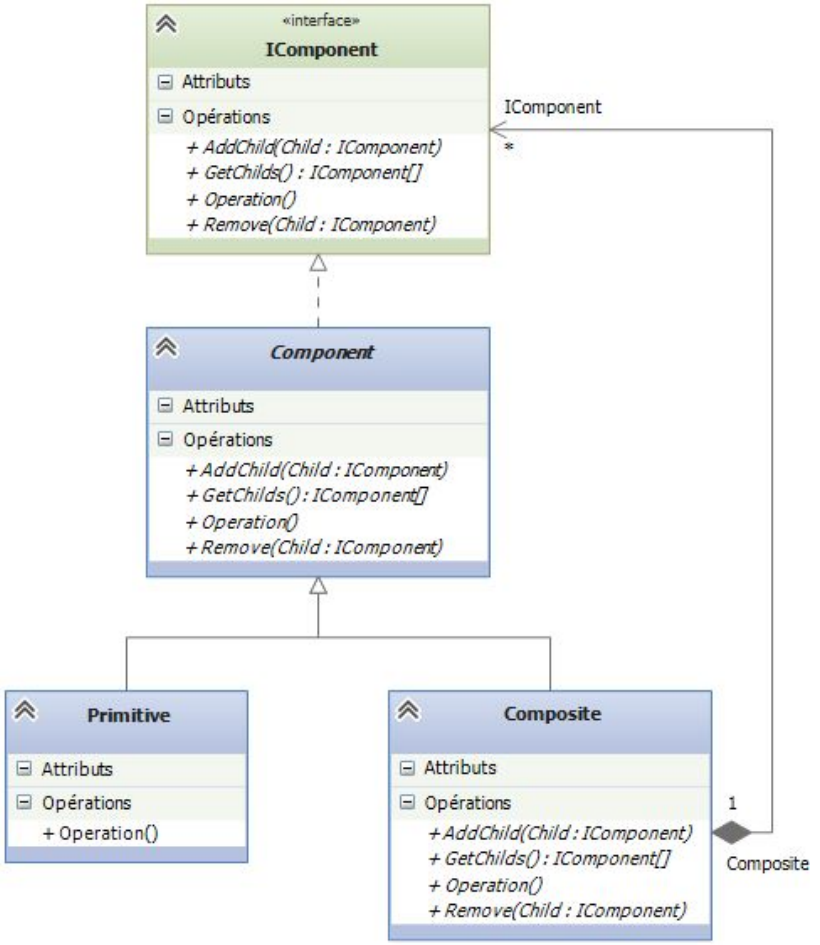
INTRODUCTION

Orienté « Comportement » :

- Chaîne de responsabilité (Chain of responsibility)
- Commande (Command)
- Interpréteur (Interpreter)
- Itérateur (Iterator)
- Médiateur (Mediator)
- Mémento (Memento)
- Observateur (Observer)
- État (State)
- Stratégie (Strategy)
- Patron de méthode (Template Method)
- Visiteur (Visitor)
- Fonction de rappel (Callback)

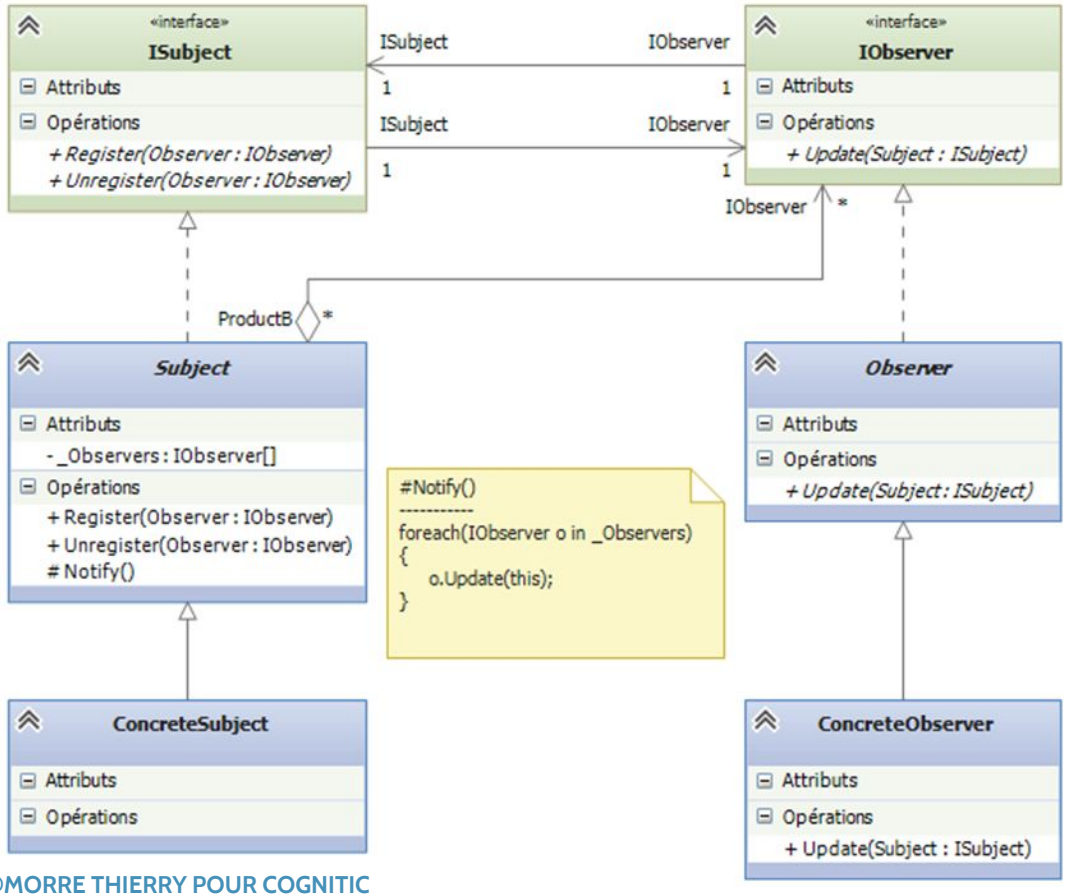
COMPOSITE

- IComponent : est une interface qui représente les fonctionnalités nécessaire d'un composant.
- Component : est une classe abstraite qui implémente, si nécessaire, les fonctionnalités par défaut de tous les composants.
- Primitive : est une classe qui représente un composant primitif, celui-ci n'a pas d'enfant.
- Composite : est une classe qui représente un composant qui peut avoir des enfants.



Notez la notion de composition qui relie « Composite » à « Icomponent ».

OBSERVER

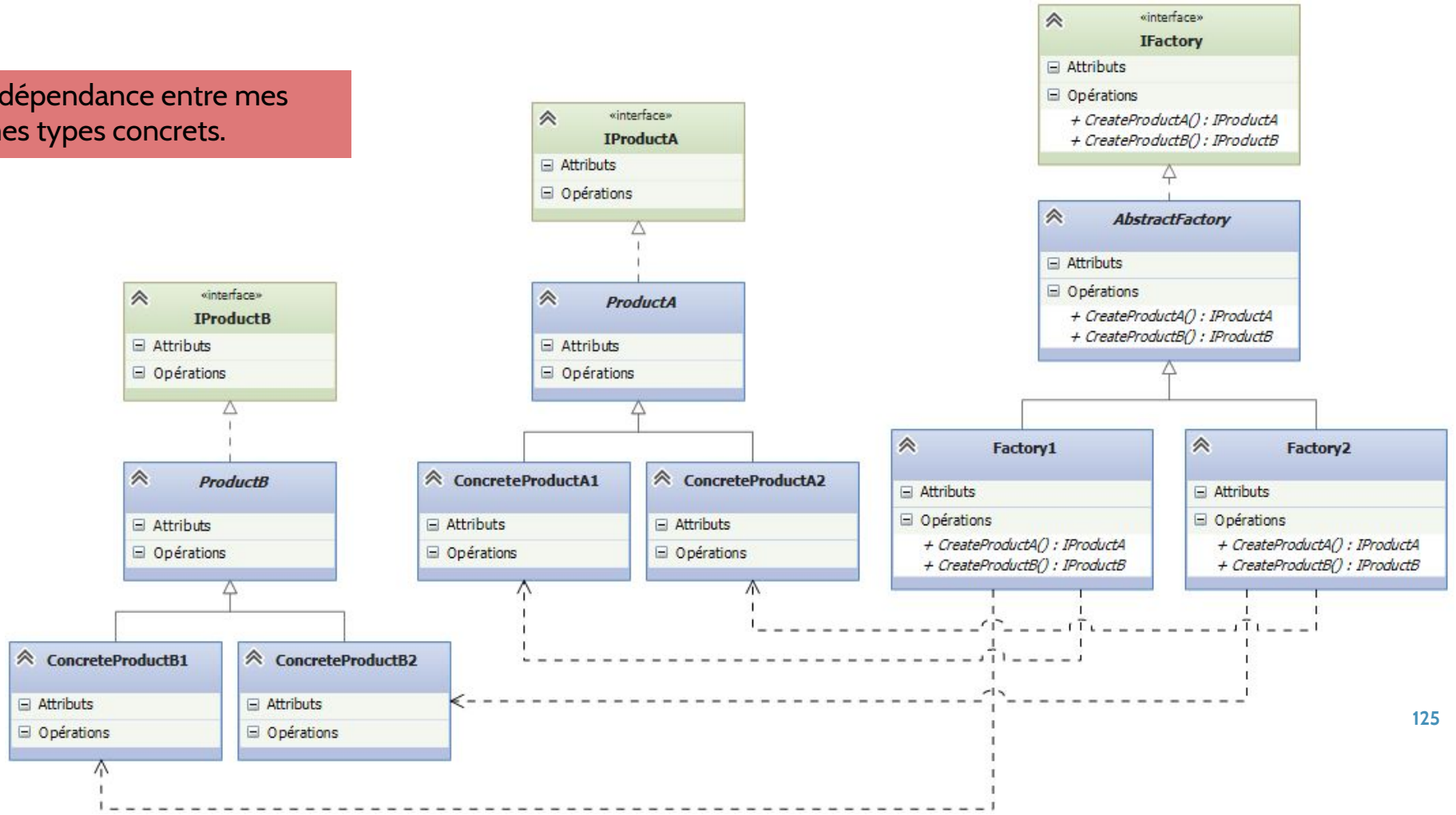


- ISubject est une interface qui représente les fonctionnalités nécessaire d'un sujet.
- Subject est une classe abstraite qui implémente les fonctionnalités par défaut de tous les sujets.
- IObserver est une interface qui représente les fonctionnalités nécessaire d'un observateur.
- Observer est une classe abstraite qui représente, si nécessaire, le fonctionnel commun à tous les observateurs.

Notez la notion d'agrégation qui relie « Subject » à « IObserver ».

ABSTRACT FACTORY

Notez la notion de dépendance entre mes fabriques et mes types concrets.



RESSOURCES

INTRODUCTION À L'APPROCHE ORIENTÉ OBJET

- Design Patterns: Elements of Reusable Object-Oriented Software
  ISBN-13 : 978-0201633610
- Design Pattern :
  http://www.dofactory.com
- Images Star Wars :
  http://joewight.deviantart.com/gallery
- Cours UML 2.1 Cognitic
- UML pour les développeurs, Isabelle Mounier et Xavier Blanc

RESSOURCES