

POO

Programmation Orientée Objet

Xavier Tabuteau

Objectifs de la formation

- ▶ Découvrir le concept de programmation orientée objet
- ▶ Modéliser un système informatique à l'aide d'UML
- ▶ Mettre en oeuvre les bonnes pratiques d'analyse et de conception

Table des matières

- ▶ **I. S'initier aux concepts de base de la POO.**
 - ▶ Programmation procédurale vs programmation orientée objet.
 - ▶ Découvrir la classe en programmation.
 - ▶ Connaître les objets en programmation.
 - ▶ Faire le point sur l'abstraction.
 - ▶ Approcher la notion d'encapsulation.
 - ▶ Savoir ce qu'est l'héritage.
 - ▶ Comprendre le polymorphisme.
- ▶ **II. Analyser un problème en programmation.**
 - ▶ Comprendre l'analyse orientée objet et son processus.
 - ▶ Définir les besoins en programmation.
 - ▶ S'initier à UML.
- ▶ **III. Voir des cas d'utilisation de la conception orientée objet.**
 - ▶ Comprendre les cas d'utilisation.
 - ▶ Identifier les acteurs.
 - ▶ Identifier les scénarios.
 - ▶ Réaliser un diagramme de cas d'utilisation.
 - ▶ Découvrir la user story.

Table des matières

- ▶ **IV. S'initier à la modélisation.**
 - ▶ Créer un modèle conceptuel.
 - ▶ Identifier les classes.
 - ▶ Identifier les relations.
 - ▶ Identifier les responsabilités.
 - ▶ Utiliser les CRC (Classe Responsabilité Collaborateur).
- ▶ **V. Créer des classes orientées objet**
 - ▶ Créer le diagramme de classe.
 - ▶ Convertir le diagramme de classe en code.
 - ▶ Découvrir la durée de vie d'un objet.
 - ▶ Utiliser les membres statiques ou partagés.
- ▶ **VI. Découvrir l'héritage, l'interface, l'agrégation et la composition.**
 - ▶ Identifier les situations d'héritage.
 - ▶ Utiliser l'héritage.
 - ▶ Utiliser les classes abstraites.
 - ▶ Utiliser les interfaces.
 - ▶ Utiliser les agrégations et la composition.
- ▶ **VII. Connaître les concepts avancés de la programmation orientée objet.**
 - ▶ Créer des diagrammes de séquence.
 - ▶ Travailler avec les diagrammes UML avancés.
 - ▶ Utiliser les outils UML.

I.

S'initier aux concepts de base de la conception orientée objet.

Dans ce chapitre, découvrons les bases indispensables de la programmation orientée objet.

1.

Procédural vs POO

Avantages et inconvénients

Programmation procédurale

Consiste à écrire son code en suivant une procédure logique, en suivant de façon séquentielle une suite de fonctions qui sont appelées dans le code avec parfois des fonctions qui s'appellent entre elles.

Ces fonctions sont appelées dans le code source en leur passant ou non des paramètres. Ces fonctions renvoient parfois des résultats exploitables dans la suite de la procédure.

Cette façon de coder est plus abordable aux débutants et demande moins d'analyse avant le développement.

Cependant le code source sera souvent plus long, difficilement réutilisable et plus difficile à maintenir.

Cela dit, certains langages comme le C sont prévus pour écrire du code en procédural.

Programmation orientée objet

Cette façon de coder s'approche un peu plus proche du raisonnement humain.
Dans cette approche on crée des objets qu'on appelle des classes.

Chaque classe va avoir ses propres attributs et ses propres méthodes.

L'héritage et le polymorphisme vont nous permettre de factoriser certaines opérations entre les différents modèles de données similaires.

De plus nous allons pouvoir aller du général vers le particulier en créant des classes parents et des classes enfants.

La POO donne un code plus lisible car il est constitué de petites entités qui vont interagir.

Enfin les classes enfants héritent des modifications apportées aux classes mères ce qui limite les erreurs et oubli plus fréquents en procédural.

En **conculsion**

Procédural.

- ▶ A base de fonctions qui ne contiennent pas les données.
- ▶ Code difficilement réutilisable tel quel.
- ▶ Chaque fonction ou procédure est associée à un traitement particulier.
- ▶ Chaque fonction peut être utilisée à différents emplacements du programme.
- ▶ Plus simple à conceptualiser.

POO.

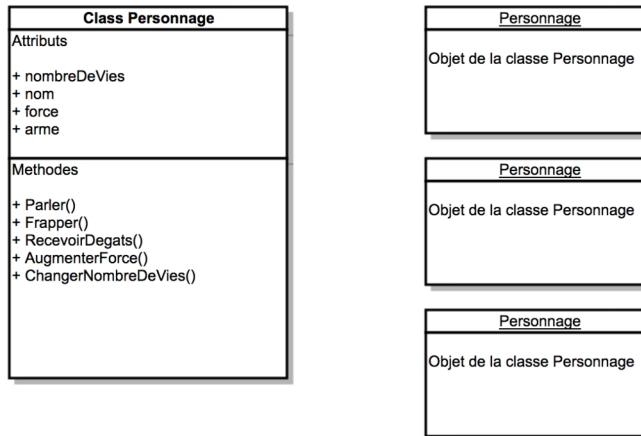
- ▶ A base d'objets qui contiennent les données.
- ▶ Code réutilisable.
- ▶ Code scindé en entités autonomes (classes).
- ▶ Les données sont protégées par encapsulation.

2.

La notion de Classes

Qu'est ce qu'une **classe**

Une classe est un plan qui nous permet de créer plusieurs objets à partir de ce plan. Exemple : Plusieurs personnages d'un jeux vidéo. On parle alors d'instances de la classe Personnage.



Comment créer une **classe**

- ▶ **Nom** : c'est quoi ?
Employé, compte bancaire, joueur, document, album...
- ▶ **Attributs** : ce qui la décrit.
Largeur, hauteur, couleur, type de fichier, score...
On les appelle aussi des **propriétés**.
- ▶ **Comportements** : que peut elle faire ?
Jouer, ouvrir, chercher, enregistrer, imprimer...
On les appelle le plus souvent des **méthodes**.

Classe / Objet

CompteBancaire
numCompte
solde
dteOuverture
typeDeCompte
ouvrir() fermer() depot() retrait()

Classe

A7652
500€
05/03/2000
courant
ouvrir() fermer() depot() retrait()

compteJean

B2311
-50€
01/02/2012
courant
ouvrir() fermer() depot() retrait()

compteFlorence

S2314
7 500€
01/02/1994
épargne
ouvrir() fermer() depot() retrait()

comptePierre

Objet (instance)

Les class native du langage

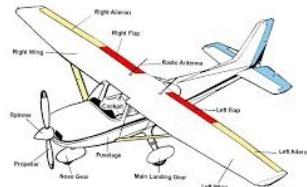
- ▶ La plupart des langages aujourd’hui proposent des classes pré-conçues prêtes à l’emploi.
- ▶ On peut utiliser ces classes et créer les nôtres.
- ▶ String(), Date(), Array()....

3.

Connaître les
objets en
programmation.

Qu'est ce qu'un **Objet**

- ▶ Chaque objet à une vie qui lui est propre (si je casse une tasse, l'autre reste intacte).
- ▶ Chaque objet peut avoir des propriétés différentes (une tasse peut être vide et l'autre à moitié pleine, une pomme peut avoir un nombre de feuilles différent ou une couleur différente).
- ▶ Chaque objet à ses propres comportements (un avion vole alors qu'un téléphone sonne).



Exemple d'objet informatique

- ▶ Un compte en banque est un objet informatique.
- ▶ Chaque objet est indépendant des autres.
- ▶ Chaque objet dispose de ses propres caractéristiques.
- ▶ Les objets peuvent interagir entre eux.

	solde : 500€ compte : A7652	nom : "Jean" sexe : homme	nom : "Florence" sexe : femme	
	depot() retrait()	marcher() courir()	marcher() courir()	

Objet compte bancaire Objet personne Objet personne

4.

Faire le point
sur l'abstraction.

4 notions fondamentales en POO

- ▶ L'Abstraction.
- ▶ Encapsulation.
- ▶ Héritage.
- ▶ Polymorphisme.

La notion d'abstraction fait référence à un objet que l'on visualise sans pour autant que celui-ci soit parfaitement décrit. Exemple : une table est un objet que l'on visualise, pourtant il reste abstrait car l'on ne connaît pas ses propriétés. Le principe de l'abstraction en programmation est de se focaliser sur l'essentiel en ignorant tous les détails.



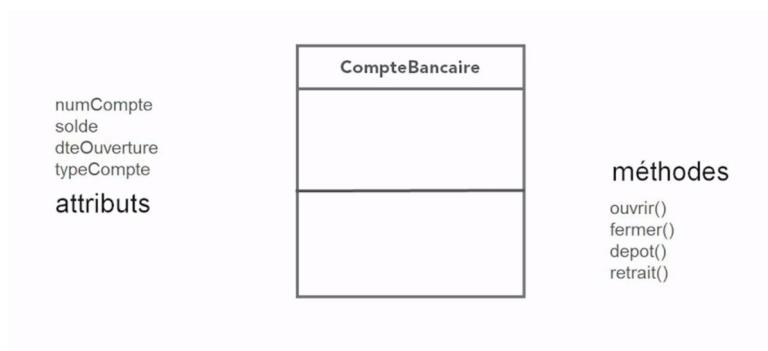
5.

L'encapsulation.

4 notions fondamentales en POO

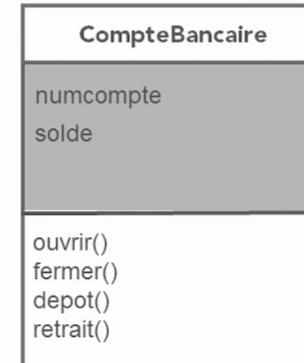
- ▶ L'Abstraction.
- ▶ Encapsulation.
- ▶ Héritage.
- ▶ Polymorphisme.

Consiste en premier lieu à encapsuler nos méthodes et nos attributs au sein d'une même unité : la classe.



Respect de l'encapsulation

- ▶ Un objet ne doit révéler de lui-même que le strictement nécessaire aux autres parties du programme. Exemple du compte bancaire : le solde n'est accessible que via une méthode de la classe. Sans utiliser cette méthode, on ne peut pas accéder au solde.
- ▶ On peut modifier les données d'une classe sans pour autant modifier les données de tous les objets.
- ▶ Les attributs sont privés.



6.

L'héritage.

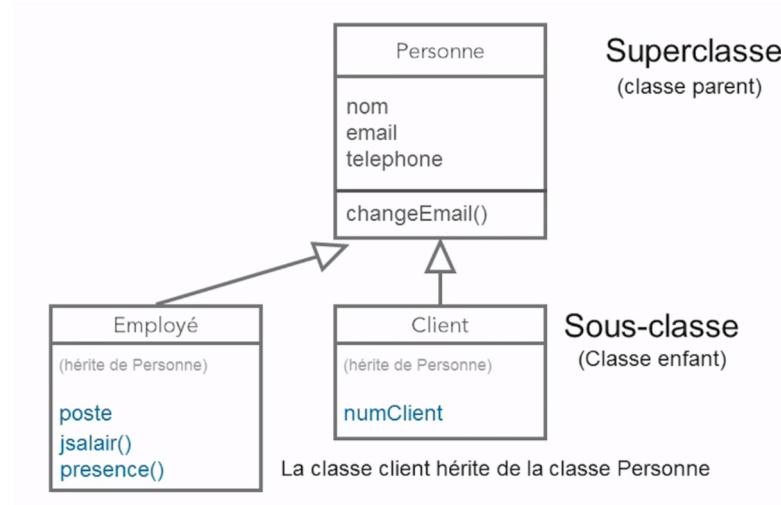
4 notions fondamentales en POO

- ▶ L'Abstraction.
- ▶ Encapsulation.
- ▶ Héritage.
- ▶ Polymorphisme.

L'héritage est une manière très pratique de réutiliser du code.

Les classes enfants héritent des attributs et méthodes de la classe parent.

Si on modifie la classe parent, toutes les classes enfants bénéficient de ces modifications.



7.

Le polymorphisme.

4 notions fondamentales en POO

- ▶ L'Abstraction.
- ▶ Encapsulation.
- ▶ Héritage.
- ▶ Polymorphisme.

Polymorphisme : qui peut prendre plusieurs formes.

$$a + b$$

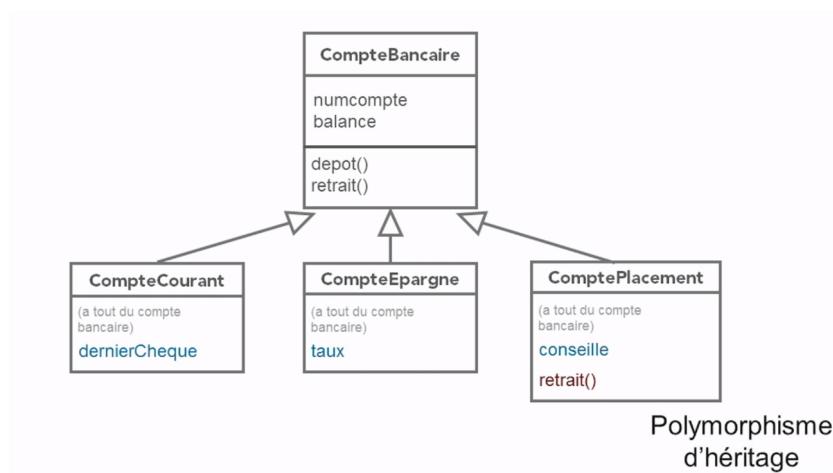
addition
5 7 12

Le + ne fait pas la
même chose.

$$a + b$$

concaténation
"Pierre" "Dubois" "PierreDubois"

Cas concret de Polymorphisme



- ▶ La méthode retrait() de la classe ComptePlacement se comporte différemment.
- ▶ Le polymorphisme est rarement utilisé.

III.

Analyser un problème en programmation.

Dans ce chapitre, nous verrons pourquoi et comment aborder l'analyse d'un problème en programmation.

8.

**Comprendre
l'analyse orientée
objet et son
processus.**

Prendre les bonnes Décisions ?

- ▶ Rassembler les besoins

Qu'est ce que l'application doit faire ? se concentrer sur le général et éviter le cas particulier.

- ▶ Décrire l'application

Décrire l'application simplement. Raconter l'histoire, les cas d'utilisations.

- ▶ Identifier les principaux objets

Essayer d'identifier les classes principales à partir des cas d'utilisations.

- ▶ Décrire les interactions

Exemple : Si objet client va ouvrir un compte en banque. Décrire les méthodes et comportements que les objets ont quand ils interagissent.
Diagramme de séquence.

- ▶ Créer un diagramme de classe

Représentation visuelle des classes de l'application en restant le plus proche possible des notions d'héritage, encapsulation et polymorphisme...

9.

Définir les besoins
en programmation.

Définir les **besoins**

- ▶ Les **besoins fonctionnels** : que font-ils ?
Caractéristiques, capacités.
- ▶ Les **besoins non fonctionnels** : quoi d'autres ?
Aide, législation, performance, support, sécurité

Exemple de **besoins fonctionnels**

- ▶ L'application doit continuer à fonctionner sans connexion réseau.
- ▶ L'application doit permettre à l'utilisateur de créer un message de 140 caractères.
- ▶ Le programme doit permettre de générer des reçus par email.
- ▶ L'application doit permettre à l'utilisateur d'effectuer une recherche par nom de client, numéro de téléphone ou numéro de client.

Les besoins exprimés doivent pouvoir être compris par l'ensemble des parties prenantes (maîtrise d'ouvrage, client, maîtrise d'oeuvre, ...)

Exemple de **besoins non fonctionnels**

- ▶ Le système doit répondre aux recherches dans les 2 secondes.
- ▶ Le support est disponible par téléphone du lundi au vendredi de 9h à 18h.
- ▶ Se conformer à tous les règlements pertinents de la sécurité sociale.

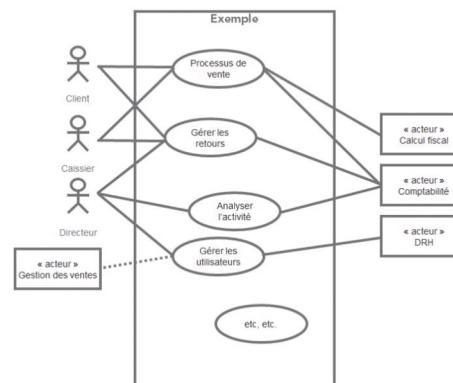
Sur de très grosses applications il est évident que d'autres techniques permettent de formaliser les besoins de manière plus adéquate.

10.

Introduction à UML.

Qu'est ce que UML ?

- ▶ UML (Unified Modeling Langage)
Langage de modélisation unifié.
- ▶ Utilisé pour modéliser des projets informatiques
Particulièrement utilisé en orienté objet.
- ▶ Exemple d'outil en ligne pour UML (et autres...)
<https://www.gliffy.com/>





Voir des cas d'utilisation de la conception orientée objet.

Ce chapitre est dédié à la compréhension des cas d'utilisation de la programmation orientée objet.

11.

Comprendre les cas d'utilisation.

Les cas d'utilisation

3 éléments sont à réunir :

- ▶ L'objectif : quel est le but ?
- ▶ L'acteur : qui en a besoin ?
- ▶ Le traitement : comment le problème est traité habituellement ?

Les cas d'utilisation : objectif

Utiliser des phrases courtes, verbes actifs

Exemples :

Enregistrer un nouveau membre.

Transférer des fonds.

Acheter des articles.

Créer une nouvelle page.

Traiter les retards de paiement.

Les cas d'utilisation : acteurs

Un acteur est une **personne** ou un **système** qui interagissent avec le programme.

Personnes :

- Employé
- Client
- Membre
- Administrateur...



Systèmes :

- Web services
- API
- Programme tiers...

Les cas d'utilisation : traitement

Scénario des actions à réaliser. Détailler l'ensemble des actions à mener pour atteindre l'objectif fixé.

- ▶ Rester bref.
- ▶ Utiliser un langage facilement compréhensible par n'importe quel utilisateur final.
- ▶ Eviter la documentation trop longue.
- ▶ Décrire le scénario du point de vue de l'utilisateur.

Les cas d'utilisation : exemple de scénario rédigé.

Titre : Acheter des articles

Acteur : Client.

Scénario :

Le client met les articles dans son panier. Le client fourni son adresse de livraison et effectue son paiement par cb. Le système valide le paiement et répond en confirmant la commande, puis il fourni un numéro de commande que le client peut utiliser pour vérifier l'état de sa commande. Le système envoie au client une copie du détail de sa commande par email.

Les cas d'utilisation : exemple de scénario avec listes à puces.

Titre : Acheter des articles

Acteur : Client.

Scénario :

1. Le client choisit d'entrer dans le processus de commande.
2. Le système présente au client une page de confirmation permettant de changer les quantités, supprimer ou annuler des éléments.
3. Le client entre son adresse de livraison.
4. Le système valide l'adresse du client.
5. Le client sélectionne un mode de paiement.
6. Le système valide les informations de paiement.
7. Le système génère un numéro de commande qui sera utilisé pour le suivi.
8. Le système affiche au client un écran de confirmation.
9. Le système envoie un mail au client avec le détail de sa commande.

Les cas d'utilisation : Scénarios alternatifs et détails.

Titre : Acheter des articles

Acteur : Client.

Acteur secondaire : ...

Scénario :

Portée : ...

Niveau : ...

Extensions : Décrire les mesures à prendre pour les situations hors stock.

Extensions : Décrire les mesures pour les commandes jamais finalisées.

Conditions préalables : Le client a ajouté au moins un article dans le panier...

Conditions finales : ...

Les intervenants :

Les technologies :

12.

Identifier les
acteurs.

Les cas d'utilisation : Identifier les acteurs

Acteurs : personnes



Responsable de la paie

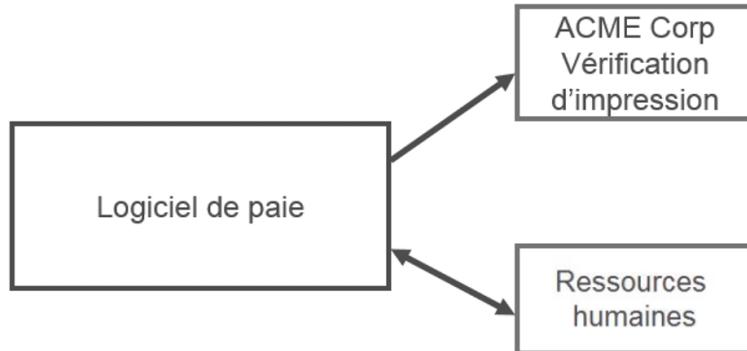


Employé



Directeur

Acteurs : systèmes



Les cas d'utilisation : Identifier les acteurs

Acteurs : systèmes externes, organisations

- ▶ Sources de données externes.
- ▶ Services web.
- ▶ Systèmes de sauvegarde...

Acteurs : personnes (identifier les rôles, la sécurité)

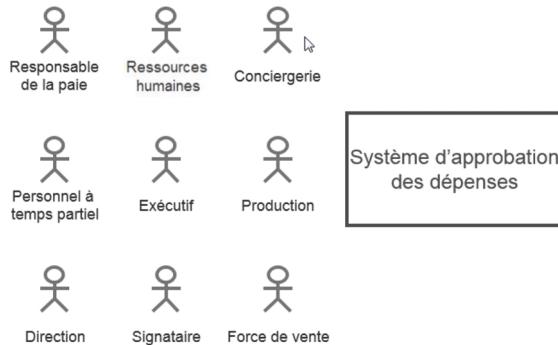
- ▶ Visiteurs.
- ▶ Membres.
- ▶ Administrateurs...

Acteurs : personnes (identifier les titres, postes, départements...)

- ▶ Directeurs.
- ▶ Responsable de la paie.
- ▶ Equipe de production...

Dans tous les cas, mettre l'accent sur l'objectif que l'acteur veut atteindre en fonction de son rôle, de son poste, en tenant compte du fait qu'il peut avoir plusieurs fonctions.

Les cas d'utilisation : regroupement d'acteurs



Principaux acteurs



Approbateur
Acteurs secondaires

13.

Identifier les
scénarios.

Identifier les scénarios

- ▶ Rester concentrer sur l'objectif à atteindre par l'acteur.

Acheter des articles



Se connecter à une application

>> Trop court

Créer un nouveau document

Écrire un livre

>> Trop large

Faire une balance comptable

Fusionner des traitements

>> Trop vague



Les scénarios alternatifs

Titre : Acheter des articles

Acteur : Client.

Scénario :

Le client met les articles dans son panier. Le client fournit son adresse de livraison et effectue son paiement par cb. Le système valide le paiement et répond en confirmant la commande, puis il fournit un numéro de commande que le client peut utiliser pour vérifier l'état de sa commande. Le système envoie au client une copie du détail de sa commande par email.

Extensions : Un ou plusieurs articles sont en rupture.

- (a) Le client retire les produits et continue sa commande.
- (b) Le client annule complètement sa commande.
- (c) ...

Extensions : Moyen de paiement rejeté.

- (a)

La rédaction du scénario

- ▶ Eviter les mots compliqués et inutiles.
- ▶ Le scénario doit être facilement compréhensible par l'utilisateur final et l'ensemble de l'équipe.
- ▶ Rédiger du point de vue de l'utilisateur et non du point de vue du programme. Ne pas expliquer ici le fonctionnement du programme mais se concentrer sur les actions et l'objectif de l'acteur.

Ne pas dire : le système se connecte au système de paiement externe via https et utilise JSON pour présenter les informations de paiement à la validation, puis attend une réponse.

Dire : le système valide les informations de paiement.

La rédaction du scénario

- ▶ Eviter de détailler les interfaces et fonctionnement, boutons, clics...

Titre : Acheter des articles

Acteur : Client.

Scénario :

1. Le client choisi d'entrer dans le processus de commande.
2. Le système présente au client une page de confirmation permettant de changer les quantités, supprimer ou annuler des éléments.
3.

A aucun moment l'on ne fait ici mention d'interfaces, de boutons ou de clics pour aller jusqu'au bout du processus de commande. Seules les intentions de l'utilisateur sont listées.

Après la rédaction : **questions à se poser**

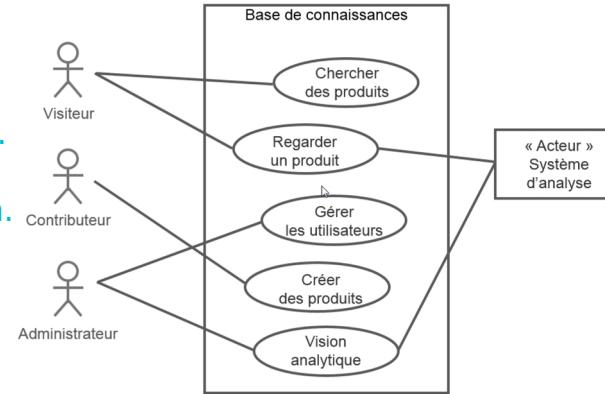
- ▶ Qui effectue les tâches d'administration du système ?
- ▶ Qui gère les utilisateurs et la sécurité ?
- ▶ Que se passe t-il si le système tombe en panne ?

14.

Réaliser un
diagramme de cas
d'utilisation.

Diagramme de cas d'utilisation : les normes UML

- ▶ Acteurs principaux à gauche.
- ▶ Acteurs secondaires à droite.
- ▶ <<Acteur>> pour identifier les systèmes.
- ▶ Cadre pour identifier système principal.
- ▶ Ellipses pour identifier les cas d'utilisation.
- ▶ Traits sans flèches pour relier les acteurs aux cas d'utilisation.



- ▶ L'utilisateur final peut rapidement identifier les éléments manquants.
- ▶ Simple et rapide à réaliser.
- ▶ Simple à comprendre par tous les monde.

15.

Découvrir la user
story.

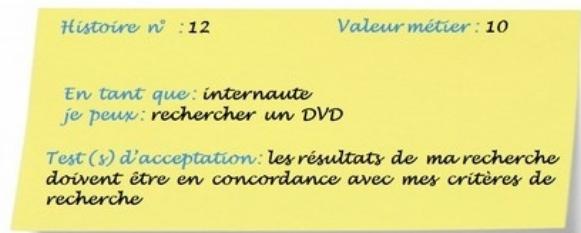
Description des besoins de l'utilisateur : **User Stories**.

- ▶ Plus simple et plus court qu'un cas d'utilisation.
- ▶ Décrit un petit scénario exclusivement du point de vue de l'utilisateur concentré sur l'objectif qu'il veut atteindre.

En tant que ... (type d'utilisateur)

Je veux ... (objectif à atteindre)

De sorte que ... (facultatif) (pourquoi l'utilisateur veut atteindre l'objectif)



Exemples : User Stories.

En tant que ... Client de ma banque.

Je veux ... Changer mon code PIN en ligne.

De sorte que ... (facultatif) Je n'ai plus à aller dans mon agence.

En tant que ... Utilisateur.

Je veux ... Effectuer une recherche par mot-clé.

De sorte que ... (facultatif) Je puisse trouver des articles rapidement.

En tant que ... Lecteur.

Je veux ... Changer de police et de couleur.

De sorte que ... (facultatif) Je puisse lire dans des conditions d'éclairage différentes.

L'objectif des User Stories est de faire le point sur tous les besoins spécifiques de l'utilisateur et pourquoi il en a besoin.

User stories vs cas d'utilisation.

Besoins des utilisateurs	Cas d'utilisation
concis : un index	long : un document
un objectif, pas de détails	objectifs multiples et détails
informel	langage de l'utilisateur final
« un endroit pour converser »	« l'enregistrement d'une conversation »

- ▶ On peut retrouver à la fois des US et des cas d'utilisation UML, mais le plus souvent il s'agit de l'un ou de l'autre.
- ▶ Dans un projet Agile (Xp, scrum) les US remplacent souvent les cas d'utilisation UML.
- ▶ Dans le cadre d'une méthodologie formelle avec un processus unifié, l'utilisation des cas d'utilisation UML remplace les US.

IV.

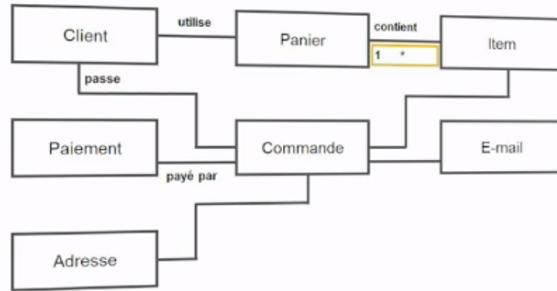
S'initier à la modélisation.

Dans ce chapitre, nous allons faire le point sur la modélisation en programmation.

16.

Créer un modèle
conceptuel.

Le modèle conceptuel



- ▶ Identifier rapidement les objets (au sens large du terme) les plus importants de l'application.
- ▶ Les mots-clé utilisés nous permettent d'identifier les futures classes.
- ▶ C'est un schéma simplifié nous permettant de comprendre les différentes interactions entre les objets.
- ▶ A ce stade, un oubli n'est pas important.

17.

Identifier les
classes.

Modèle conceptuel, identifier les objets

C'est la première étape qui permet de rassembler tous les cas d'utilisation et récits utilisateurs (scénarios)

- ▶ A partir des scénarios, repérer les noms qui semblent être de bons candidats pour créer nos futurs objets.
- ▶ Une première sélection se fait de manière complètement intuitive.

Scénario :

Le client met les articles dans son panier. Le client fourni son adresse de livraison et effectue son paiement. Le système valide le paiement et répond en confirmant la commande, puis il fourni un numéro de commande que le client peut utiliser pour vérifier l'état de sa commande. Le système envoie au client une copie du détail de sa commande par email.

Modèle conceptuel, identifier les objets

- ▶ Lister les noms soulignés.
- ▶ Repérer et supprimer les termes redondants.
- ▶ Regrouper les informations décomposées.

Client
Articles
Panier
Adresse
Paiement
Valider le paiement



Redondance

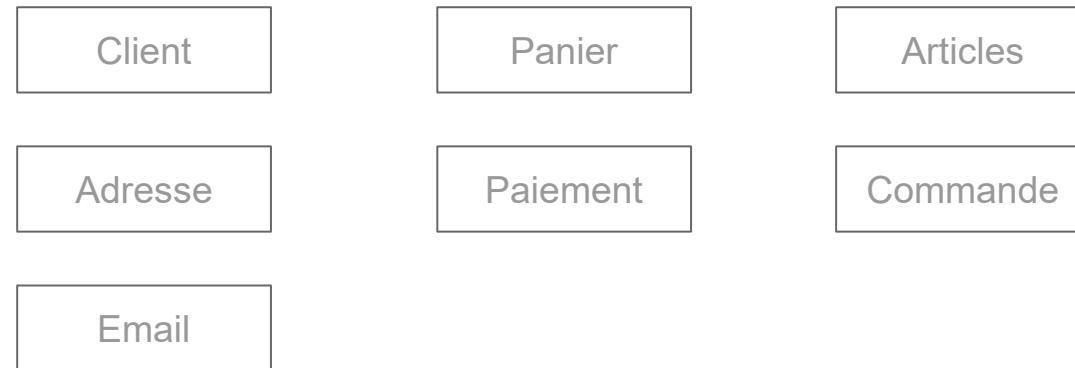
Numéro de commande
Etat de la commande
Détail de la commande
Commande
Email



Regroupement

Modèle conceptuel, identifier les objets

- ▶ Dessiner les noms restants dans des boîtes..



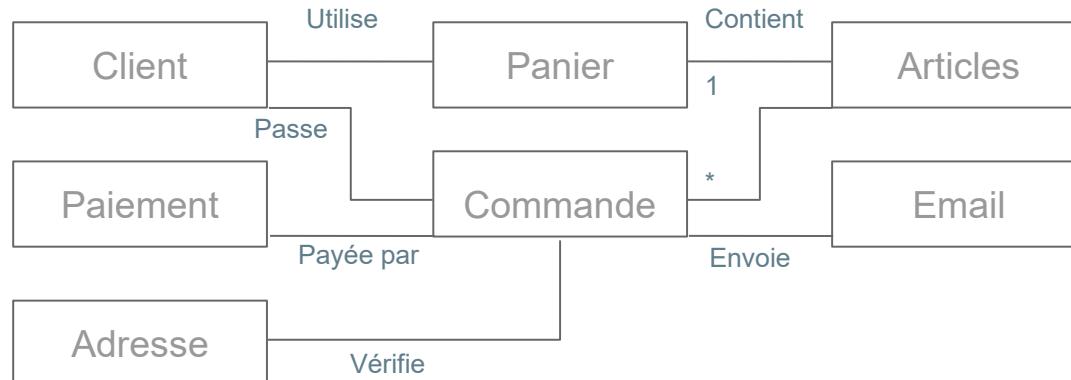
Ce diagramme va nous permettre de montrer facilement les responsabilités et les relations entre les différents objets que nous avons commencé à rassembler.

18.

Identifier les
relations.

Modèle conceptuel, identifier les relations

- ▶ Dessiner les relations entre ces objets.
- ▶ Si besoin se reporter aux cas d'utilisateurs et scénario.
- ▶ Exprimer la nature des liaisons entre objets.
- ▶ Diagramme doit rester compréhensible par des non informaticiens.



L'étape suivante consiste à identifier les responsabilités des différentes classes (les méthodes).

19.

Identifier les
responsabilités.

Modèle conceptuel, identifier les responsabilités

- ▶ Souligner les verbes, les actions, ce que font les acteurs dans le scénario.

Scénario :

Le client vérifie les articles dans son panier pour réaliser sa commande. Le client fourni son adresse de livraison et effectue son paiement. Le système valide le paiement et répond en confirmant la commande, puis il fourni un numéro de commande que le client peut utiliser pour vérifier l'état de sa commande. Le système envoie au client une copie du détail de sa commande par email.

Vérifie les articles

Paie et donne l'adresse

Valide de paiement

Réalise la commande

Confirmer la commande

Fourni un numéro de commande

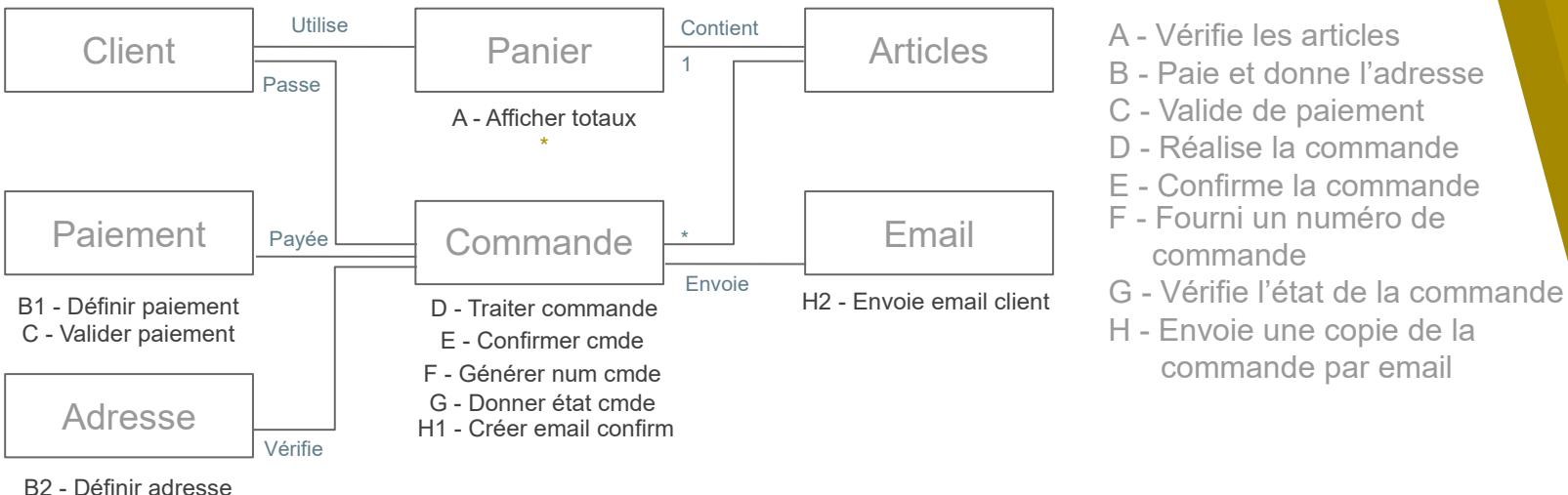
Vérifie l'état de la commande

Envoie une copie de la commande par email

- ▶ Supprimer les redondances et regrouper les actions si nécessaire.

Modèle conceptuel, identifier les responsabilités

- Parmi les actions, déterminer quel objet en a la responsabilité (qui va gérer cette tâche ?).



* Le panier est l'entité qui permet au client de vérifier ses articles et affiche les totaux.

Modèle conceptuel, identifier les responsabilités

A retenir :

- ▶ L'objet initiateur n'est pas l'objet responsable *.
- ▶ Il faut répartir les responsabilités le plus possible sur différents objets.

* L'objet client est à l'initiative de nombreuses actions, mais il n'est pas l'objet responsable du traitement de ces actions.

20.

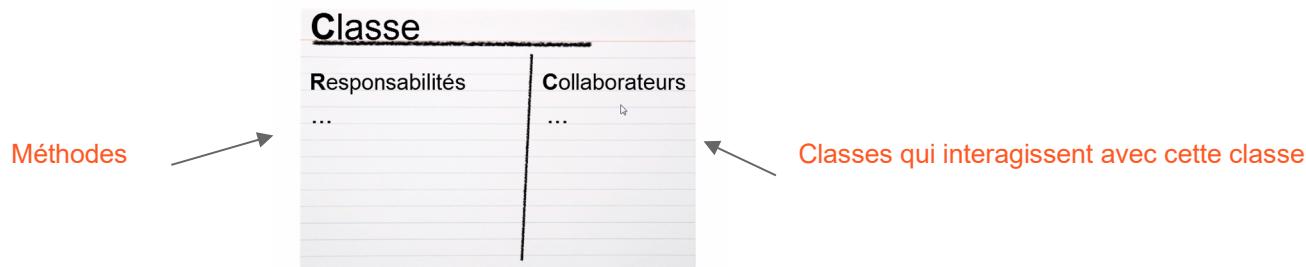
Les cartes CRC.
Classes
Responsabilités
Collaborateurs

Les cartes CRC

Il s'agit d'un autre outil très pratique permettant de définir nos classes, d'identifier leurs relations et leurs responsabilités.

Avantages :

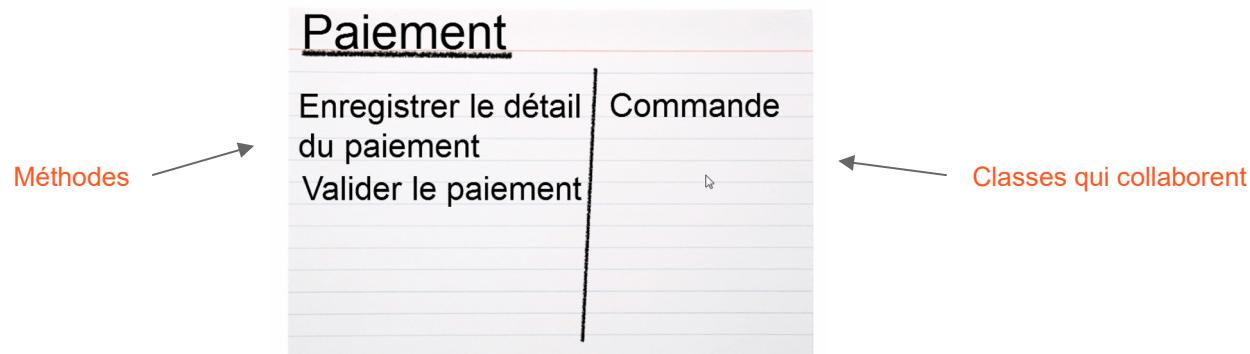
- ▶ Facile à créer.
- ▶ Facilite la discussion entre développeurs.
- ▶ Facile à détruire en cas de changement d'avis.



Les cartes CRC

A partir des scénarios :

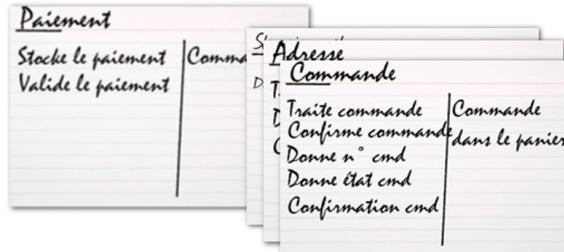
- ▶ Lister les actions à gauche.
- ▶ Lister les objets qui collaborent à droite.



Les cartes CRC

Autres avantages :

- ▶ Il est possible de déplacer les cartes pour mieux comprendre comment elles interagissent entre elles.
- ▶ Facile de déterminer ainsi les collaborateurs naturels.



Les cartes CRC

Conseil :

- ▶ Si la liste de responsabilité ne tient pas sur une seule carte, il faut repenser la classe en question.

Conclusion :

- ▶ Après cette étape (ou phase modèle de conception), vous devez avoir au moins le nom et les responsabilités de base du premier ensemble de classes à coder.



V.

Créer des classes orientées objet.

Ce chapitre permet d'aborder la programmation en commençant par la création des classes.

21.

Créer le
diagramme de
classe.

Diagramme de **classe**

C'est le diagramme le plus couramment rencontré.
Il permet de définir les classes avec leurs attributs et méthodes.

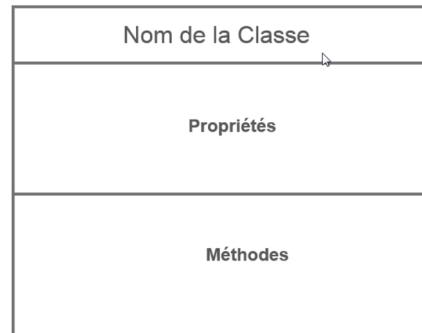


Diagramme de classe : le nom de la classe

- ▶ Respecter les conventions de nommage.
- ▶ Les noms de classes commencent par une majuscule.

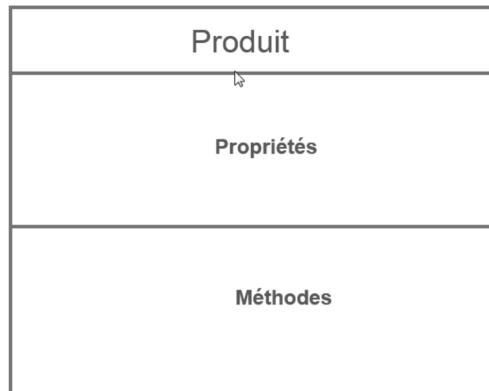


Diagramme de classe : les attributs

- ▶ Respecter les conventions de nommage.
- ▶ Commencent par une minuscule puis chaque mot par une majuscule, ni espace ni caractères spéciaux.
- ▶ On peut en désigner le type sans tenir compte du langage utilisé par la suite.
- ▶ On peut l'initialiser.
- ▶ Respecter les règles d'écriture en UML.

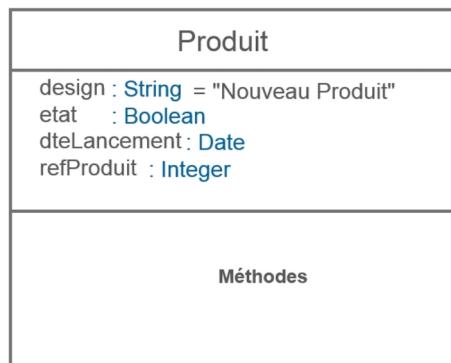


Diagramme de classe : les méthodes

- ▶ Respecter les conventions de nommage.
- ▶ Commencent par une minuscule puis chaque mot par une majuscule, ni espace ni caractères spéciaux.
- ▶ Faire suivre des parenthèses.
- ▶ On peut désigner les paramètres en réception par leur type.
- ▶ On peut désigner les types de valeurs retournées (type return).
- ▶ Autant que possible nommer les méthodes getters et setters (setNom, getDate).
- ▶ Désigner les visibilités (+ et -,...), ne rendre visible que le stricte minimum.
- ▶ Respecter les règles d'écriture en UML.

Produit
- design : String = "Nouveau Produit" - etat : Boolean - dteLancement : Date - refProduit : Integer
+ donneNom () : String + definirEtat (Boolean) + donneDetailProd() : String + afficheProd () - formatDetailProd() : String

Diagramme de classe : les méthodes et commentaires

- ▶ Représenté par un cadre dont l'angle supérieur droit est replié.
- ▶ Relié à la méthode par des pointillés.

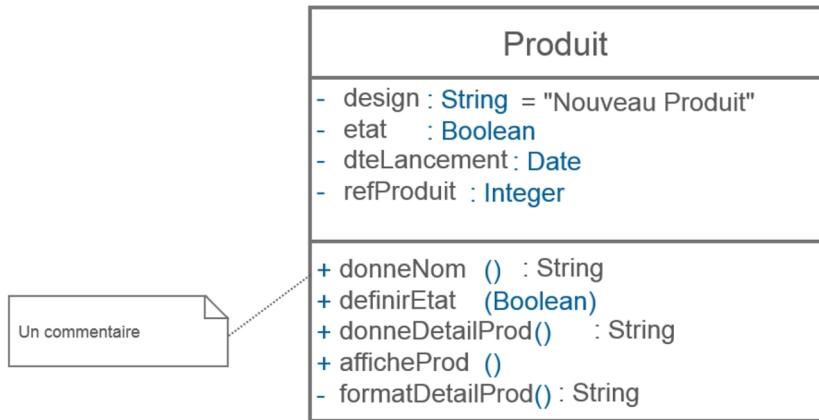
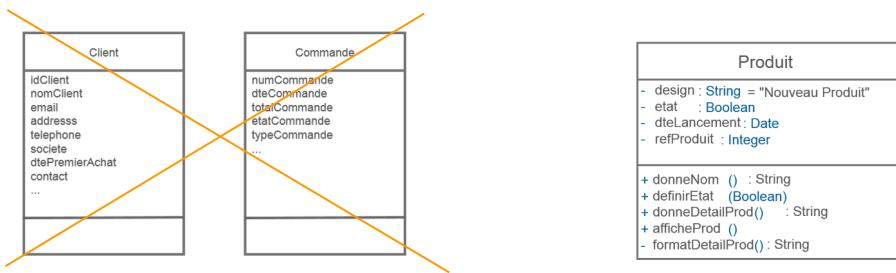


Diagramme de classe : ne sautez pas les étapes précédentes.

Conseils :

- ▶ Passer directement au diagramme de classe a pour résultat de se concentrer davantage sur les attributs en omettant les méthodes.
- ▶ Ce type d'approche qui consiste à se focaliser sur les données que l'entité doit contenir est dites de type relationnelle et davantage destinée aux bases de données par exemple.
- ▶ Il sera très difficile d'établir les relations entre classes avec cette approche.



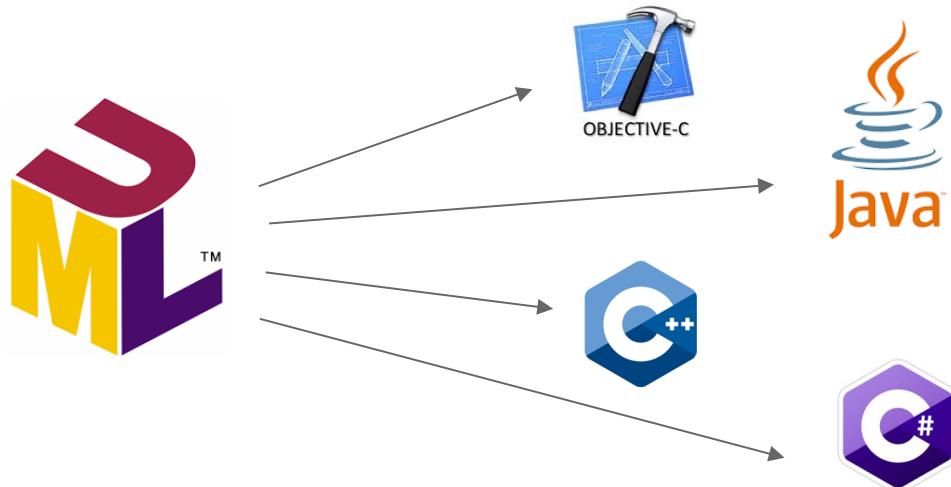
Nous intégrerons les relations entre classe sur le diagramme un peu plus tard.

22.

Convertir le
diagramme de
classe en code.

Convertir le diagramme de classe en code.

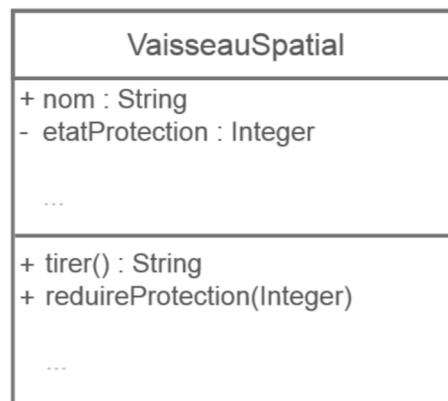
- ▶ UML : un langage universel.
- ▶ Utilisable avec n'importe quel langage de programmation.



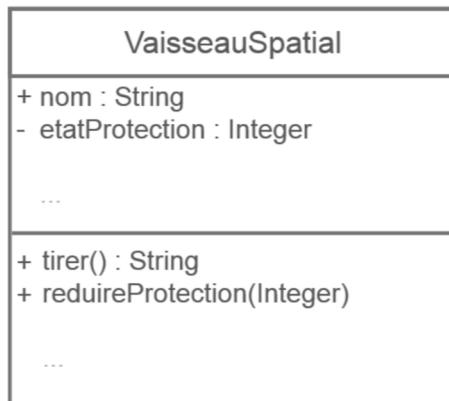
Convertir le diagramme de classe en code.

Ce diagramme permet :

- ▶ De comprendre tout de suite qu'il s'agit d'un jeux.
- ▶ D'en comprendre le fonctionnement rapidement (méthodes).
- ▶ De visualiser immédiatement les attributs.
- ▶ De connaître la visibilité des attributs et méthodes.

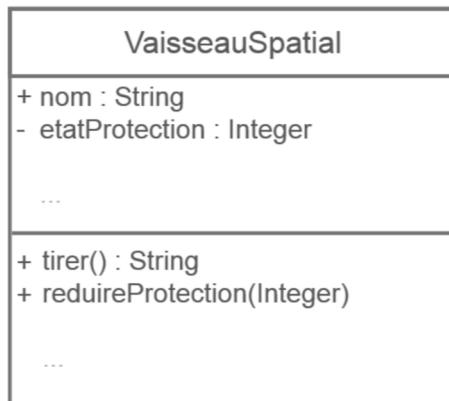


Convertir le diagramme de classe en code avec Java.



```
public class VaisseauSpatial {  
  
    // variables d'instance  
    public String nom;  
    private int etatProtection;  
  
    // méthodes  
    public String tirer() {  
        return "Booum!";  
    }  
      
    public void reduireProtection (int quantite) {  
        etatProtection -= quantite;  
    }  
}
```

Convertir le diagramme de classe en code avec C#.



```
public class VaisseauSpatial {  
  
    // variables d'instance  
    public String nom ;  
    private int etatProtection;  
  
    // méthodes  
    public String tirer () {  
        return "Booum!";  
    }  
  
    public void reduireProtection(int quantite) {  
        etatProtection -= quantite;  
    }  
}
```

Convertir le diagramme de classe en code avec VB.NET

VaisseauSpatial
+ nom : String
- etatProtection : Integer
...
+ tirer() : String
+ reduireProtection(Integer)
...

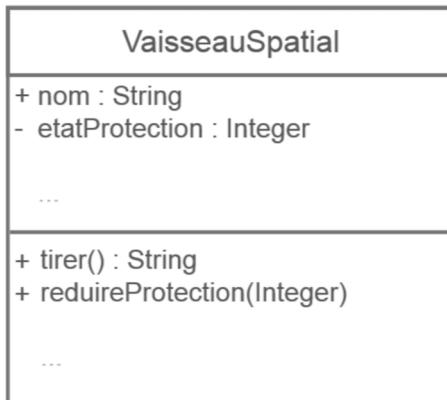
```
Public Class VaisseauSpatial
    ' variables d'instance
    Public nom As String
    Private etatProtection As Integer

    ' méthodes
    Public Function tirer () As String
        Return "Booom!"
    End Function

    Public Function reduireProtection (quantite as Integer)
        etatProtection -= quantite
    End Function

End Class
```

Convertir le diagramme de classe en code avec Ruby



```
class VaisseauSpatial

# variables d'instance
@nom
@etat_protection

# méthodes
def tirer
    return "Booum!"
end

def reduire_protection (quantite)
    etat_protection-= quantite
end

end
```

Convertir le diagramme de classe en code avec Objective-C

VaisseauSpatial
+ nom : String
- etatProtection : Integer
...
+ tirer() : String
+ reduireProtection(Integer)
...

```
@interface VaisseauSpatial: NSObject {  
    @public  
    NSString *nom ;  
    @private  
    int etatProtection;  
}
```

```
// déclaration des méthodes  
-(NSString *) tirer;  
-(void) reduireProtection:(int)quantite;  
  
@end
```

interface

```
@implementation vaisseauSpatial  
  
-(NSString *) tirer {  
    return @"Booom!";  
}  
  
-(void) reduireProtection: (int)quantite{  
    etatProtection -= quantite;  
}  
@end
```

implémentation

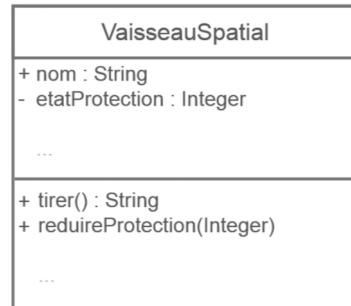
23.

Découvrir la durée
de vie d'un objet.

Durée de vie d'un objet.

Instancier un objet et lui attribuer des valeurs avec des paramètres est possible grâce à la notion de constructeur.

- ▶ Que se passe t-il quand on créé un objet (le constructeur) ?
- ▶ Que se passe t-il quand on a plus besoin de l'objet (le destructeur) ?



Instanciation d'un objet.

Dans la plupart des langages on utilise le mot réservé “New”.

Que se passe t-il à la création d'un objet ?

- ▶ Allocation mémoire pour l'objet
- ▶ Initialisation des variables de l'instance.
- ▶ Renvoi une référence vers cet objet.
- ▶ Dès lors on peut commencer à utiliser l'objet.

Java Client `jean` = new Client();

C# Client `jean` = new Client();

VB.NET Dim `jean` As New Client

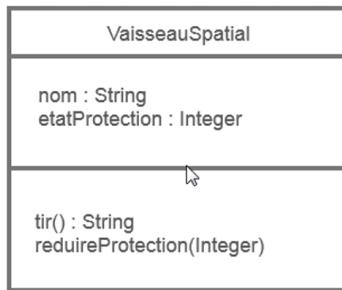
Ruby `jean` = Client.new

C++ Client *`jean` = new Client();

Objective-C Client *`jean`= [[Client alloc] init];

Le constructeur.

Une méthode qui est utilisée pour construire l'objet.



`VaisseauSpatial ariane= new VaisseauSpatial();`

`object : ariane`

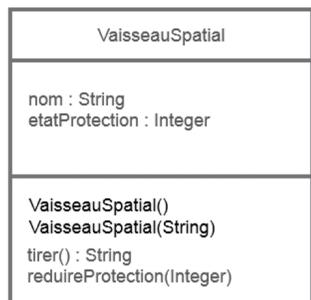
`nom : null`
`etatProtection : 0`



Résultat obtenu.

Le constructeur.

Créer un instance avec des paramètres pour le constructeur.



Représentation UML

```
public class VaisseauSpatial {  
  
    // variables d'instance  
    public String nom;  
    private int etatProtection;  
  
    // constructeur de méthode  
    public VaisseauSpatial() {  
        nom = "Vaisseau sans nom";  
        etatProtection = 100;  
    }  
    // surcharge de méthode  
    public VaisseauSpatial(String n) {  
        nom = n;  
        etatProtection = 200;  
    }  
    // autres méthodes  
}
```

```
VaisseauSpatial ariane =  
new VaisseauSpatial("ariane 2");
```

object : ariane

```
nom : ariane 2  
etatProtection : 200
```

Résultat obtenu.



Le destructeur.

Permet de libérer de la place en mémoire.

- ▶ Il faut que plus rien dans le code ne pointe vers l'objet pour le détruire car s'il est utilisé dans le code, nous ne pourrons pas le détruire.

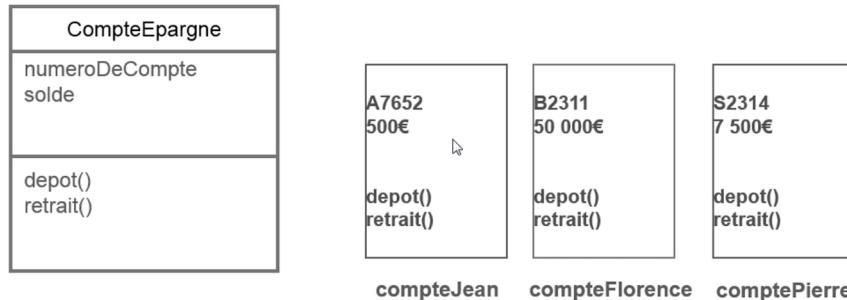
24.

Utiliser les
membres statiques
/ partagés.

Utiliser les membres **statiques** / partagés.

Dans l'exemple suivant, nous créons plusieurs objets de la classe CompteEpargne.

- ▶ On utilise autant de fois le mot “New” qu'il est nécessaire.
- ▶ Dans chaque objet les variables sont créées et ont des valeurs différentes..

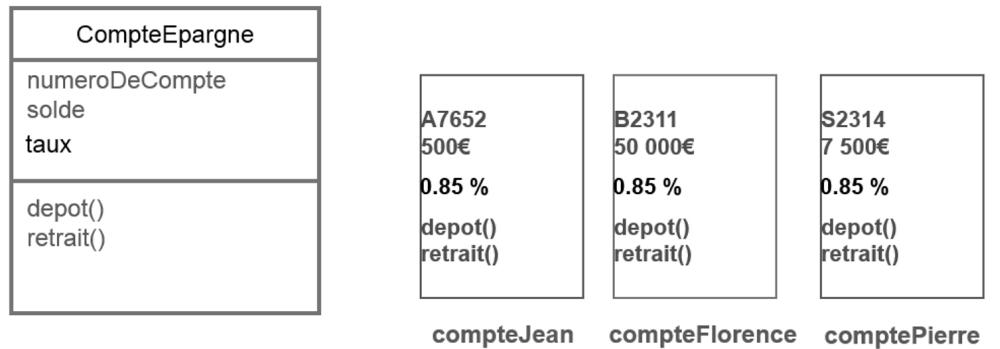


- ▶ A présent, on souhaite ajouter un taux fixe pour tous les objets....

Utiliser les membres **statiques** / partagés.

Que se passe t-il si l'on ajoute un attribut dans la classe ?

- ▶ Pour chaque instance, la variable est créée.
- ▶ Il y a alors redondance d'information.

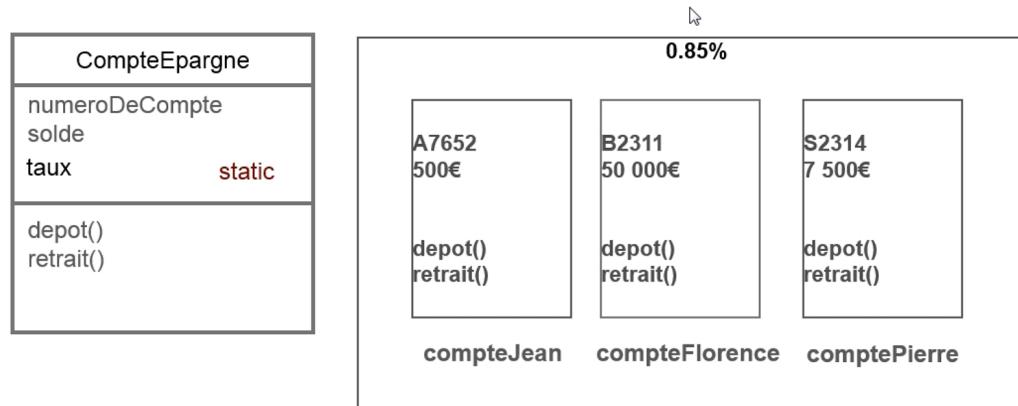


- ▶ En procédurale, on créerait une variable globale accessible partout dans le code. C'est différent en Objet, arrive alors la variable static.

Utiliser les membres **statiques** / partagés.

Résultat obtenu avec un attribut en static dans la classe ?

- ▶ La variable ne s'est pas créée dans chaque objet.
- ▶ Elle s'est créée une seule fois dans la classe.
- ▶ Elle existe et a le même valeur dans tous les objets.



- ▶ On appelle cela **un membre partagé** avec tous les objets issus de la même classe.

Création des variables **statiques** / partagés.

```
public class CompteEpargne{  
  
    // variables d'instance  
    public String numeroDeCompte;  
    private Money solde;  
  
    // variables statiques  
    public static float taux;  
  
    // le reste du code  
}
```

' VB.NET – variables partagées
Public Shared taux As Float

Ruby – variables de classe
@@taux

Création de méthodes statiques

De la même manière il est possible de créer des méthodes partagées.

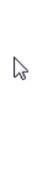
```
public class CompteEpargne {  
  
    // variables d'instance  
  
    // variable publique (accessible) static  
    public static float taux;  
  
    // méthodes publiques statiques  
    public static definirTaux(float r) {  
        // change le taux  
        taux = r;  
    }  
  
    public static donneTaux() {  
        return taux;  
    }  
  
    // le reste de votre code  
}
```

- ▶ La méthode est créée une seule fois dans la classe pour tous les objets.
- ▶ Par contre la variable taux doit passer en private (encapsulation).
- ▶ Et l'on utilise les getters et setters definirTaux() et donneTaux() ...

Création de méthodes statiques

```
public class CompteEpargne {  
  
    // variables d'instance  
  
    // changer en privé  
    private static float taux;  
  
    // méthodes publiques statiques  
    public static definirTaux(float r) {  
        // change le taux  
        taux = r;  
    }  
    public static donneTaux() {  
        return taux;  
    }  
  
    // le reste de votre code  
}
```

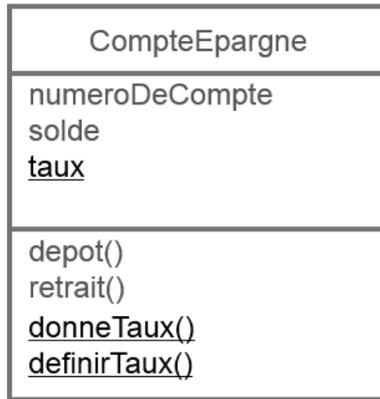
CompteEpargne.definirTaux(0.9);



Pour y accéder dans le code.

Attention, indiquer qu'une méthode est statique, c'est dévoiler un détail d'implémentation inutile au client, c'est violer le principe de l'encapsulation, de l'abstraction. En programmation objet, utiliser et s'appuyer sur des objets est bien plus prudent.

Montrer les membres statiques en UML



En UML on souligne les membres statiques.

VI.

Découvrir l'héritage, l'interface, l'agrégation et la composition.

Dans ce chapitre, nous verrons l'importance de connaître les relations entre les différents objets utilisés pour la programmation orientée objet..

25.

Identifier les
situations
d'héritage.

L'héritage décrit (est une) **relation**

Une voiture **est un** véhicule.

Un bus **est un** véhicule.

Une voiture est un bus.

Un employé **est une** personne.

Un client **est une** personne.

Un client est un parieur.

← Héritage

Un compte courant **est un type de** compte en banque.

Un compte d'épargne **est un type de** compte en banque.

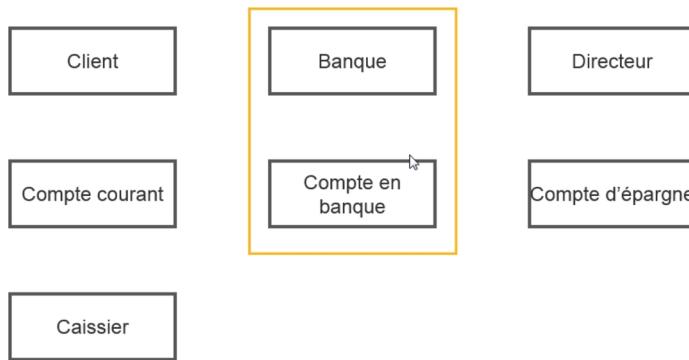
Une Peugeot 108 **est une** voiture **et un** véhicule.

Un caniche **est un** chien **et un** mammifère **et un** animal.

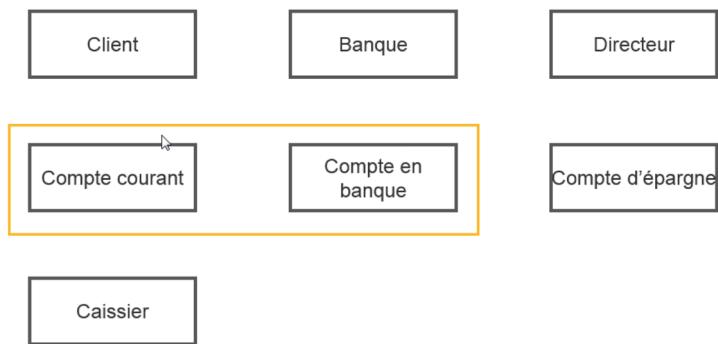
← Double Héritage

← Triple Héritage

Identification de l'héritage

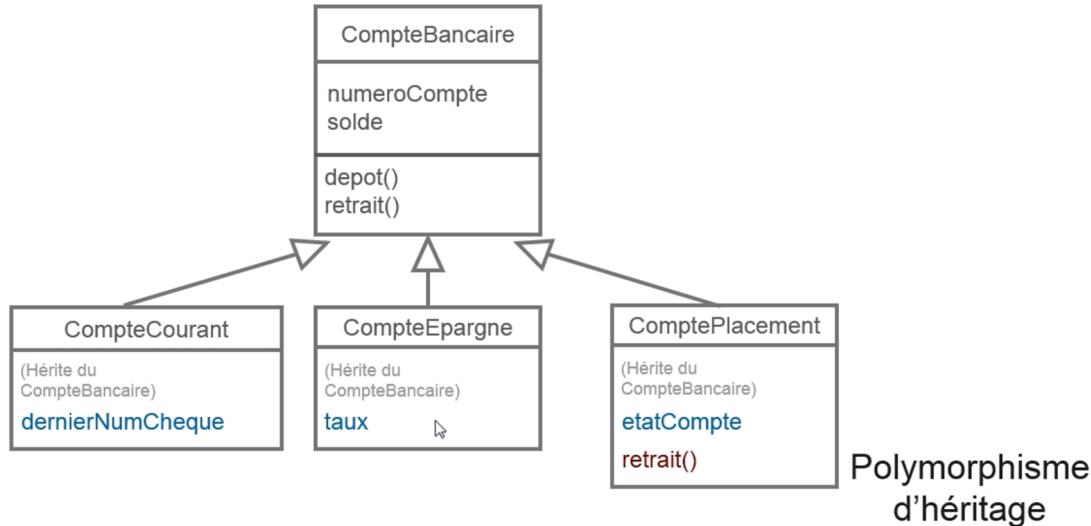


Un compte en banque **est** une banque ?
Non, pas d'héritage.

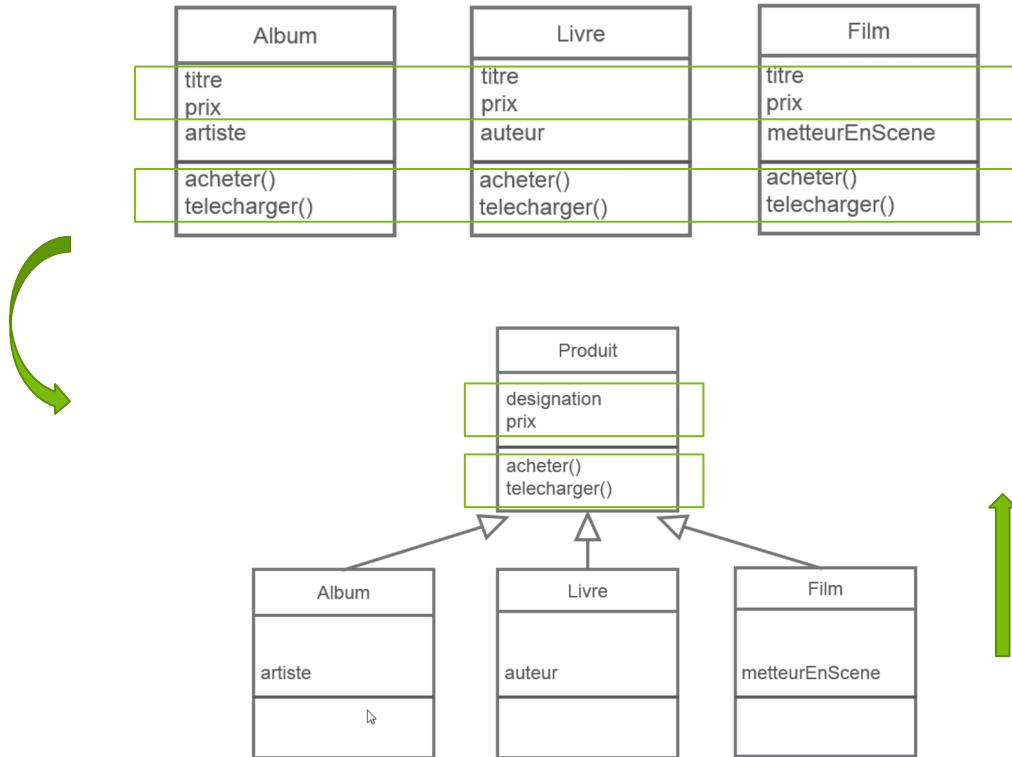


Un compte courant **est** un compte en banque ?
Oui, héritage.

Représentation UML de l'héritage

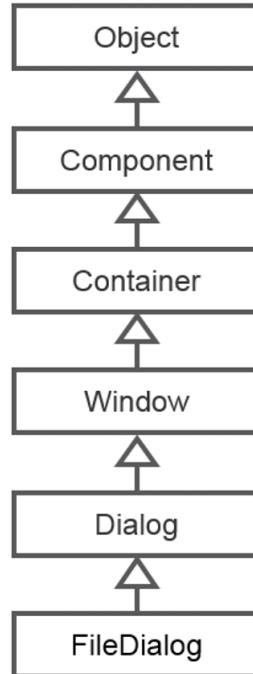


Comment déterminer qui hérite de qui ?



- En partant du bas vers le haut.
- Je constate des points communs.
- Je créé une classe parent.

Utilisé tous les jours dans les langages de programmations



- ▶ Exemple avec Java.
- ▶ Les classes fournies du langages héritent très souvent d'autres classes.

26.

Utiliser l'héritage.

L'héritage dans le code

L'héritage existe dans tous les langages de programmation orientée objet.

Java `public class Album extends Produit { ...`

C# `public class Album : Produit { ...`

VB.NET `Public Class Album
 Inherits Produit ...`

Ruby `class Album < Produit ...`

C++ `class Album : public Produit { ...`

Objective-C `@interface Album : Produit { ...`

- ▶ Le symbole le plus couramment utilisé est le symbole ":"

Polymorphisme dans une sous classe

Le mot-clé “super” est le plus couramment utilisé mais la façon d'y arriver est différente dans chaque langage et renvoie à la documentation de chaque langage.

```
class Capitale extends Ville {  
    private String monument;  
  
    public Capitale(){  
        //Ce mot clé appelle le constructeur de la classe mère  
        super();  
        monument = "aucun";  
    }  
  
    public String decrisToi(){  
        String str = super.decrisToi() + "\n \t ==>" + this.monument+ " en est un  
monument";  
        System.out.println("Invocation de super.decrisToi()");  
  
        return str;  
    }  
}
```

- ▶ Exemple en Java.

Appeler une méthode **dans une superclasse / parent**

Java `super.methodeParent();`

C# `base.methodeParent();`

VB.NET `MyBase.methodeParent()`

Ruby `super methode_parent`

Objective-C `[super methodeParent];`

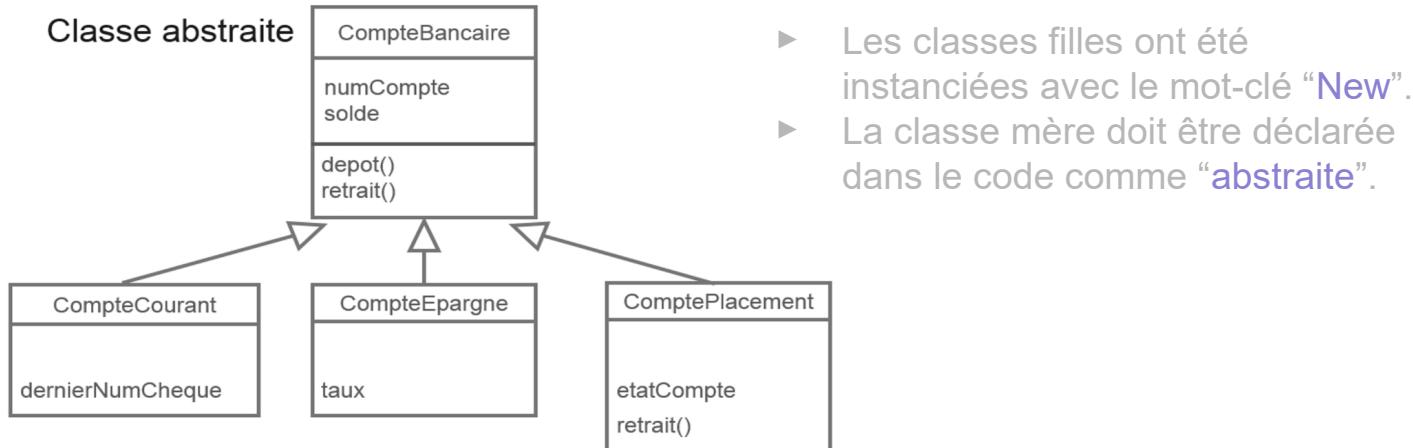
C++ `NomDeLaClasseDeBase::methodeParent();`

27.

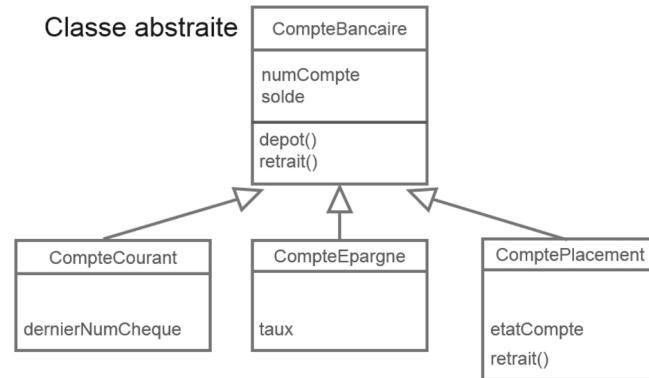
Utiliser les classes
abstraites.

Les classes abstraites

Si une classe parent n'est jamais instanciée, on parle alors de classe abstraite.



Les classes abstraites écrite dans le code



- ▶ En Java : **abstract** class CompteBancaire {}
- ▶ En VB.NET : Public **MustInherit** class CompteBancaire ...
- ▶ Dans d'autres langages, le simple fait de ne jamais l'instancier fera de la classe une classe abstraite.
- ▶ En Ruby l'abstraction n'existe pas.
- ▶ L'abstraction est très répandue en Java, C#, C++ et VB.NET

28.

Utiliser les interfaces.

Utiliser les interfaces

Une notion très voisine de l'héritage.

- ▶ Existe dans de nombreux langages de POO.
- ▶ Communément appelée interface mais n'a rien à voir avec les fenêtres ni les IHM.
- ▶ C'est un peu comme une classe sans fonctionnalités ni code réel à l'intérieur.
- ▶ Ne contient que des signatures de méthodes que je veux rendre disponibles pour plusieurs classes.

```
Interface Impression {  
    // signatures des méthodes  
    void imprime();  
    void imprimePDF(String nomFichier);  
}
```

Utiliser les interfaces, implémentation

- ▶ Créer des classes qui vont implémenter les signatures d'une interface.
- ▶ Utilisation du mot-clé "implements" pour créer une classe qui va détailler le fonctionnement des méthodes.

```
Interface Impression {  
    // signatures des méthodes  
    void imprime();  
    void imprimePDF(String nomFichier);  
}
```

```
class UneClasse implements Impression{  
    // méthodes de l'interface  
    public void imprime() {  
        // la suite du code  
    }  
    public void imprimePDF(String nomFichier){  
        // la suite du code  
    }  
    // autres fonctionnalités  
}
```

Utiliser les interfaces dans le code

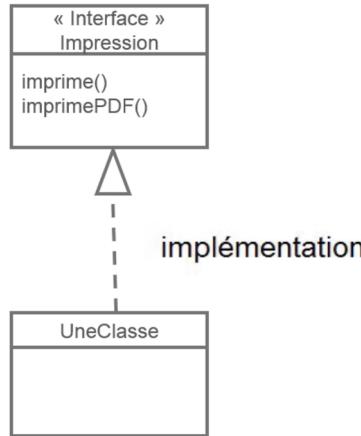
- ▶ Dans ma boucle while, si l'objetGenerique est une instance d'impression, j'ai le droit d'utiliser sa méthode "imprime()".

```
Interface Impression {  
    // signatures des méthodes  
    void imprime();  
    void imprimePDF(String nomFichier);  
}  
  
// plus loin dans le code  
while (objetGenerique in ListeObjets) {  
    if (objetGenerique instanceof Impression){  
        //  
        objetGenerique.imprime();  
    }  
}
```

Intérêts :

- ▶ Nous pouvons utiliser des méthodes appartenant à une classe sans en connaître l'organisation du code.
- ▶ J'ai simplement une interface qui met à ma disposition des méthodes que je peux utiliser à n'importe quel moment.

Représentation UML des interfaces



- ▶ <<interface>> pour désigner une classe “interface”.
- ▶ Utilisation en UML d’une flèche en pointillé.

Conclusion sur les interfaces

- ▶ N'existe pas dans tous les langages mais un grand nombre l'accepte.
- ▶ S'appelle parfois autrement, en Objectiv-C on parle de protocole.
- ▶ Approche beaucoup plus souple que l'héritage.
- ▶ Notamment lorsqu'il s'agit de faire évoluer dans le temps une application.

29.

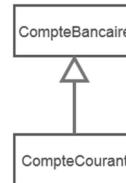
Utiliser les
agrégations et la
composition.

Les associations

- Le modèle conceptuel permet d'établir les interactions entre objets.



- L'héritage est symbolisé par une flèche de la classe fille vers la classe mère.



- Deux autres termes nous permettent d'aborder la notion de liaison : l'**agrégation** et la **composition**.

L'agrégation

- ▶ Exprime une liaison un peu comme l'héritage.

Un client **a une** adresse.

Une voiture **a un** moteur.

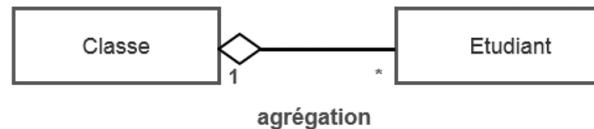
Une banque **a beaucoup** de comptes courants.

Une université **a beaucoup** d'étudiants.

- ▶ Quand on peut dire qu'une classe a un ou plusieurs objets d'une autre classe, on dit qu'il y a **agrégation** entre les deux classes.

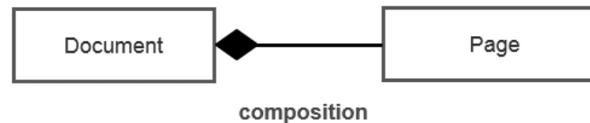
L'agrégation et la composition

- ▶ Une classe a plusieurs étudiants (**agrégation** entre deux classes).
- ▶ La classe contient au moins une instance Etudiant.



▶ UML : losange vide.

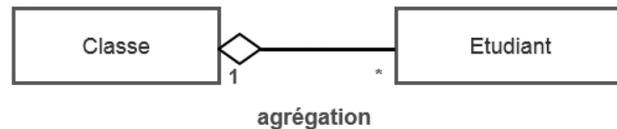
- ▶ Un document a plusieurs pages (c'est une **composition**).
- ▶ La classe Document contient au moins une instance Page.
- ▶ La destruction du document détruit toutes les pages.



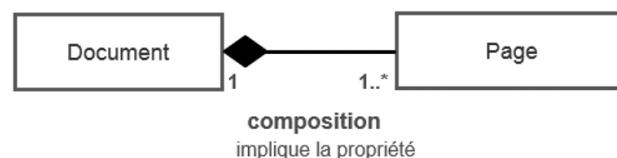
▶ UML : losange plein.

Différence entre agrégation et composition ?

- ▶ Si je supprime l'élément 1 (Objet Classe), les éléments multiples (Objets Etudiant) continuent d'exister.



- ▶ Si je supprime l'élément 1 (Objet Document), les éléments multiples (Objets Page) sont supprimés.



- ▶ La composition implique la notion de propriété alors que l'agrégation non.

VIII.

Connaître les concepts avancés de la programmation orientée objet.

Dans ce chapitre, nous allons approfondir nos connaissances dans la conception orientée objet avec UML.

30.

Créer des
diagrammes de
séquence.

Diagramme de séquence

- ▶ Permet de représenter une partie d'un processus.
- ▶ Permet de mieux comprendre ce qui doit se passer à l'instant T à l'intérieur d'un processus.

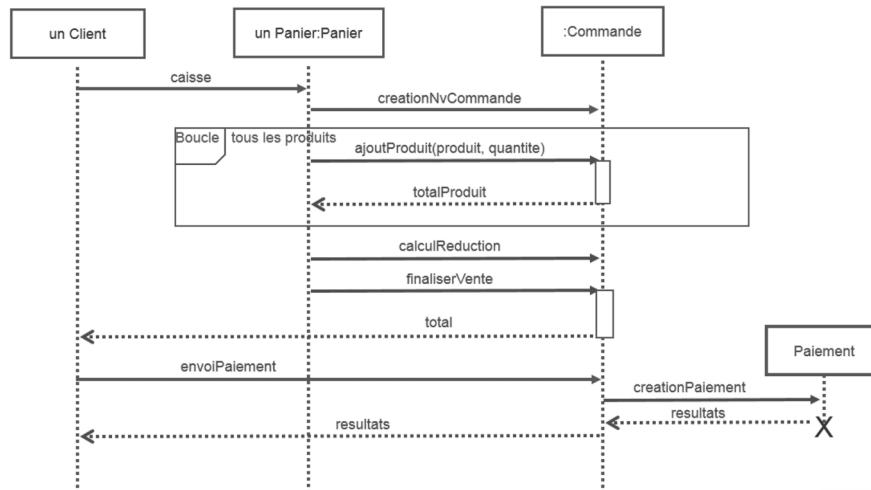


Diagramme de séquence

- Commencer par créer des boîtes représentant les objets et les classes qui interagissent.

un Client

un Panier:Panier

:Commande



Désigne un objet

Désigne un objet de
la classe Panier

Désigne la classe
Commande

Diagramme de séquence

- ▶ Ensuite l'on représente par des lignes pointillées verticales symbolisant le temps qui passe.

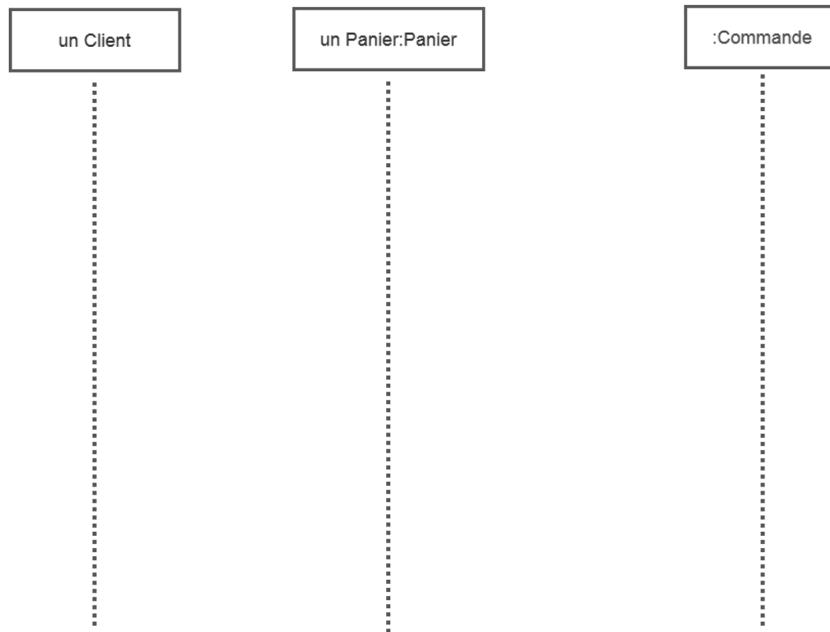
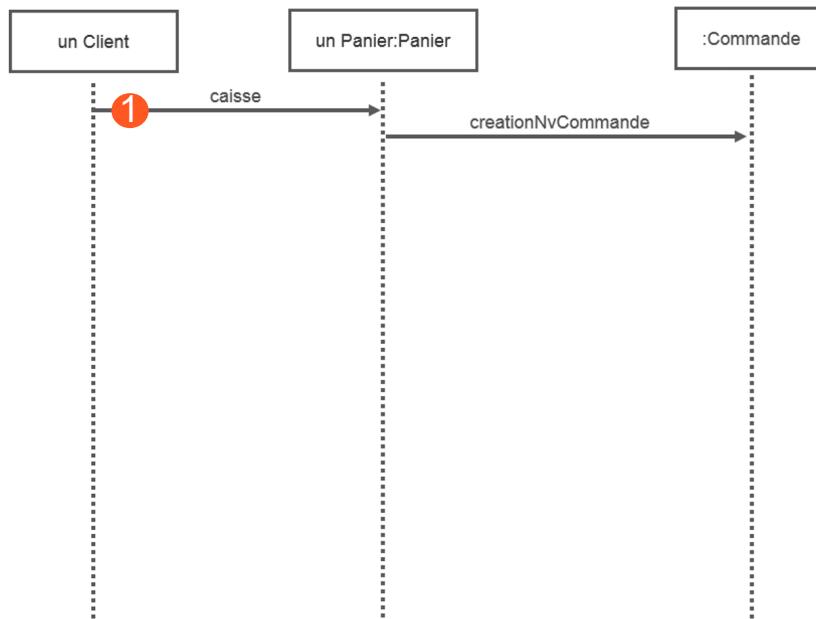


Diagramme de séquence

- ▶ Lister les séquences d'interactions entre entités (du haut vers le bas).

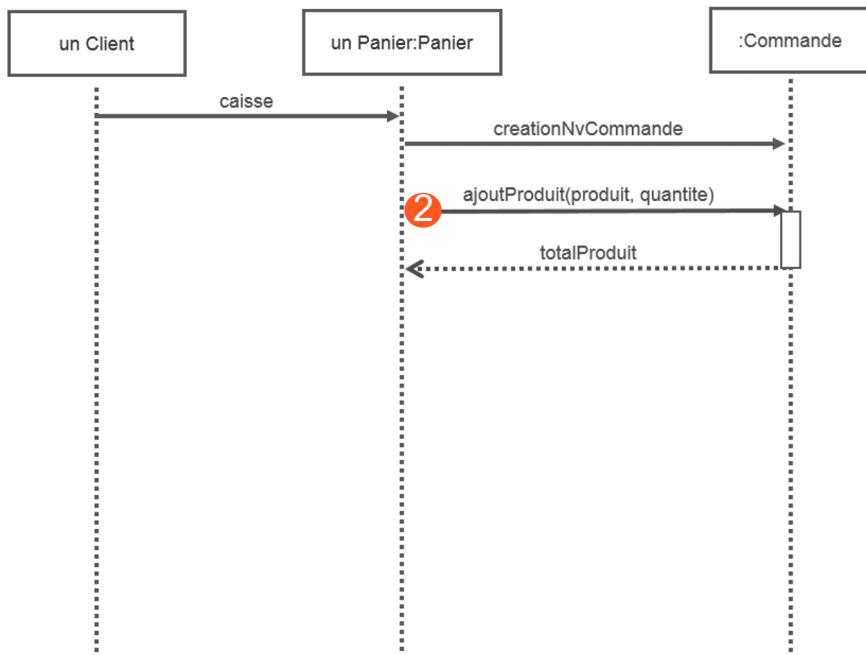


- ▶ Le client veut terminer sa commande.
- ▶ L'objet Panier doit lancer la création d'une nouvelle commande.

- 1 Flèche continue et pointe pleine indique une action.
Au dessus du trait, on indique l'action qui se passe pour passer d'une entité à l'autre.

Diagramme de séquence

- Une séquence faisant appel à une méthode.



- L'objet Panier utilise une méthode pour ajouter un produit à la commande.
- La commande renvoie un message à l'objet Panier.

② Flèche continue et pointe pleine indique une action.

Au dessus du trait, on indique l'action qui se passe pour passer d'une entité à l'autre.

Cadre blanc indique l'utilisation d'une méthode.

Flèche pointillée ouverte indique un message de retour.

Diagramme de séquence

- Représenter une boucle.

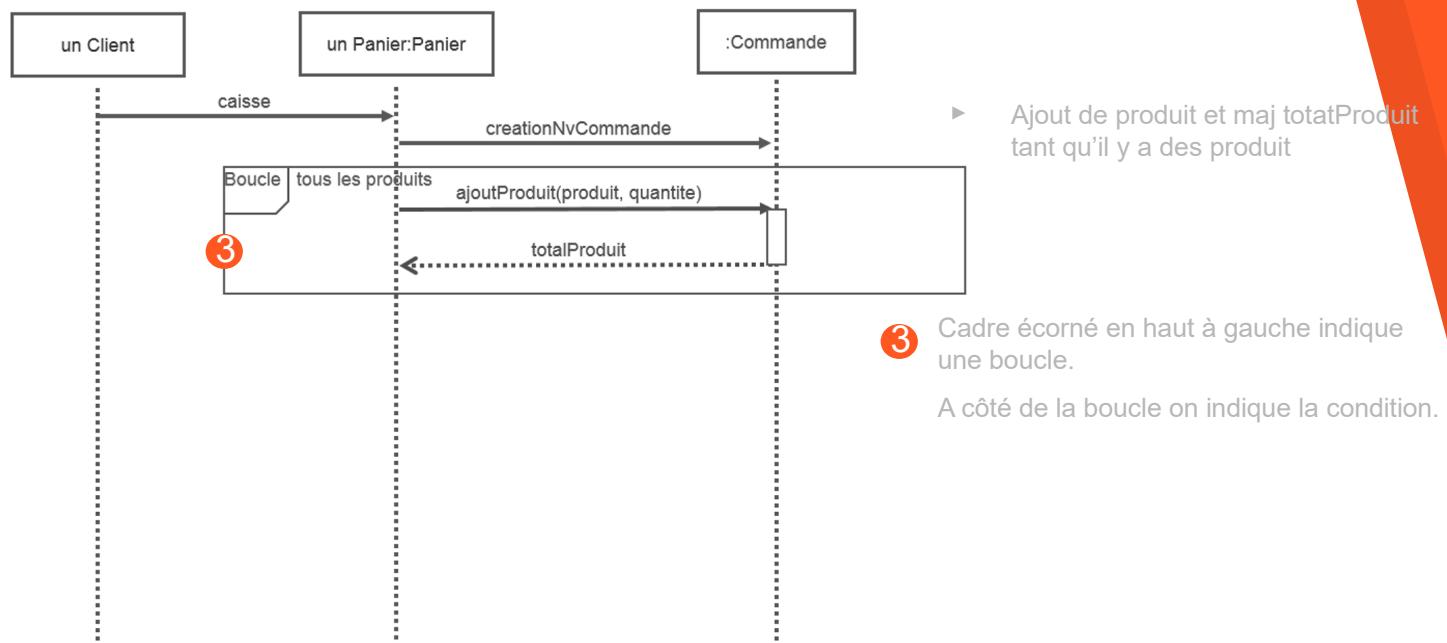
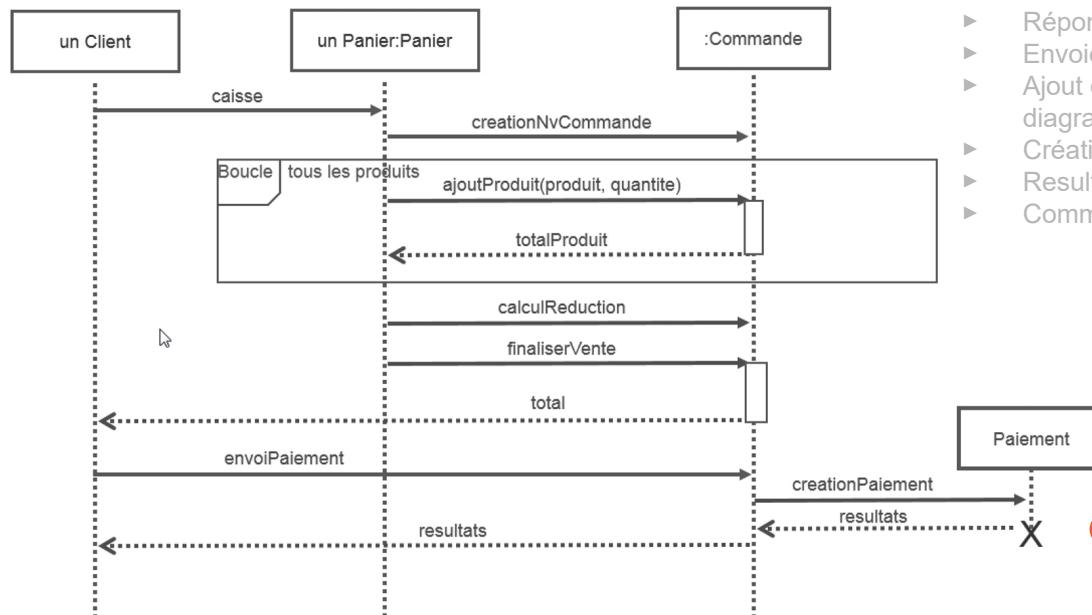


Diagramme de séquence

- Ajout d'un objet et indication de fin de vie.



- Envoi d'une réduction à Commande.
- Méthode utilisée pour finaliserVente.
- Réponse au Client.
- Envoie du paiement.
- Ajout de l'objet Paiement au diagramme.
- Création du paiement.
- Réponse retournée à Commande.
- Commande répond au client.

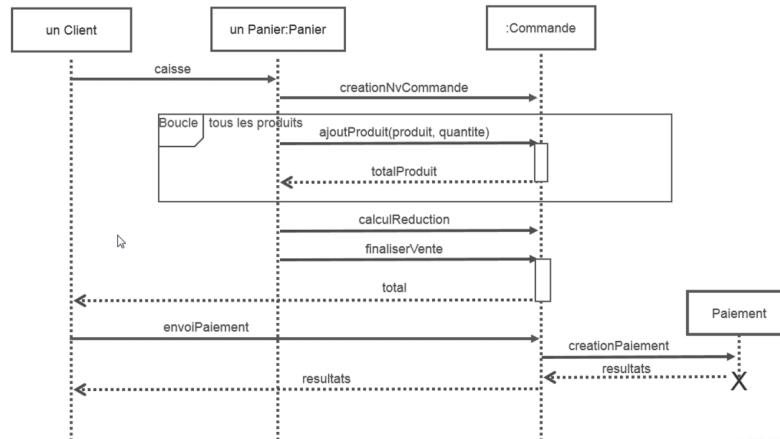
4

X indique la fin de vie de l'objet.

Diagramme de séquence

A retenir :

- ▶ Fait pour représenter les points-clés des traitements.
- ▶ Ne pas chercher à représenter l'ensemble des traitements dans le détail.
- ▶ Permet de voir rapidement si l'on a oublié une classe.
- ▶ Offre un autre outil pour échanger sur les processus avec les utilisateurs.



31.

Travailler avec les
diagrammes UML
avancés.

Diagramme UML avancés

- ▶ 14 diagrammes UML sont disponibles.

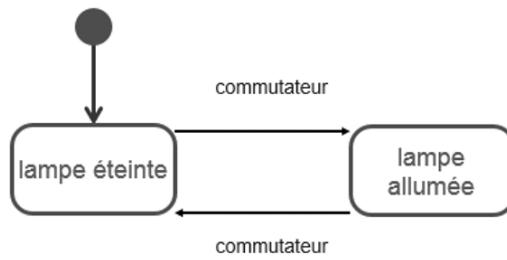
- Diagramme de classe
- Diagramme de cas d'utilisation
- Diagramme d'objet
- Diagramme de séquence
- Diagramme état-transition
- Diagramme d'activité
- Diagramme de déploiement
- Diagramme des paquetages
- Diagramme de composant
- Diagramme de profil
- Diagramme de communication
- Diagramme de temps
- Diagramme de structure composite
- Diagramme d'interaction



Les indispensables en
POO

Deux complémentaires

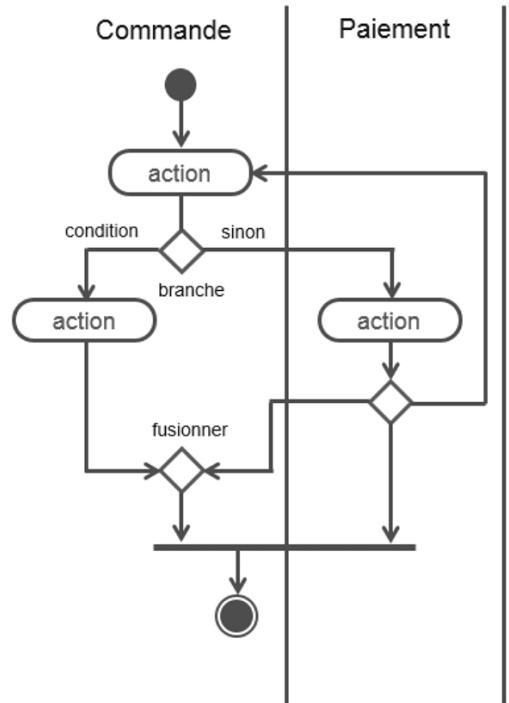
Diagramme Etat transition



Angles arrondis pour ne pas confondre avec les classes.

- ▶ Sert à montrer dans le détail les différents états qu'un objet peut avoir durant sa vie.
- ▶ Montre en détail ce qui se passe au moment d'un état à l'autre.
- ▶ Utile quand on a des objets qui existent dans des configurations extrêmement différentes.

Diagramme d'activité



Permet de se déplacer au sein d'un processus.

- ▶ Explique les scénarios alternatifs.
- ▶ Possible de le scinder pour visualiser les interactions entre objets dans le processus.

Diagramme UML conclusion

- Diagramme de classe
- Diagramme de cas d'utilisation
- Diagramme d'objet
- Diagramme de séquence
- Diagramme état-transition
- Diagramme d'activité
- Diagramme de déploiement
- Diagramme des paquetages
- Diagramme de composant
- Diagramme de profil
- Diagramme de communication
- Diagramme de temps
- Diagramme de structure composite
- Diagramme d'interaction



Suffisent dans 90% des projets.

Eventuellement.

Souvent spécifiques au langage.

32.

Utiliser les outils UML.

Outils UML et Aide au développement

- ▶ 14 diagrammes UML sont disponibles.

Outils pour réaliser des diagrammes

Visio, OmniGraffle

Outils en ligne pour réaliser des diagrammes

gliffy.com, creately.com, lucidchart.com

Outils de programmation : environnement de développement

Visual Studio, Eclipse avec UMLTools

Produits commerciaux

Altova UModel, Sparx Enterprise Architect, Visual Paradigm

Open source

ArgoUML, Dia

MÉTIER

