Bases de la conception orientée objet
Introduction

Petru Valicov
petru.valicov@univ-amu.fr

http://pageperso.lif.univ-mrs.fr/~petru.valicov/Teaching.html

2017-2018

Objectifs

- Analyse et conception d'une solution informatique
- Comprendre et modéliser une conception détaillée
- Utilisation d'une approche méthodologique
- Applications de notions en programmation objet
- Mise en œuvre à travers un langage de programmation orienté objet (C++ ou Java)
- gestion de version dans le développement + tests unitaires (si le temps le permet)

Lien étroit avec les notions vues dans le cours

Bases de la programmation orientée objet - M2103

Planning du cours

1. Introduction + besoins de l'utilisateur
2. Versionning
3. UML : Diagrammes de classes
4. UML : Aspects dynamiques des systèmes

Organisation

Volume horaire

- Cours : 6 séances de 1h30
- TD : 3 séances de 4h
- TP : 4 séances de 4h (par demi-groupe)
- Examen : 3 heures

Quatre enseignants interviennent dans les séances :

marc.laporte@univ-amu.fr - TD/TP
sophie.nabitz@univ-avignon.fr - TD/TP
petru.valicov@univ-amu.fr - CM/TD/TP
sebastien.nedjar@univ-amu.fr - un CM sur le versionning

Le travail en TD/TP est évalué. L'enseignant est là pour vous guider.

Contrôle des connaissances

Test

- 21.04.2018
- documents non-autorisés (ou presque)
- n'apprenez pas par cœur ! ! !
  préférez plutôt la compréhension des concepts

Contrôle continu - projet

- représente la partie *modélisation* de votre projet tuteuré
- collaboration avec les modules M2103, M2105 et M2107
- 12h de séances de travail prévues en salles
- plus de détails :
  https://github.com/IUTInfoAix/M2107_Projet

Les outils

- un crayon, une gomme et du papier
- un logiciel de modélisation : `StarUML`
- une IDE que vous préférez (`Eclipse`, `IntelliJ IDEA`, etc.)
- Versionning : `Git` et `GitHub`
- illustration des concepts avec des langages de programmation : `C++` et `Java`

Un peu de biblio

- **UML 2 par la pratique. Études de cas et exercices corrigés.** Pascal Roques, *7ème édition Eyrolles* : 2009.
- **UML 2.0 - Guide de référence**, *Editions Campus Press*, 2005.
- **UML 2 de l'apprentissage à la pratique. Cours et exercices.** *2ème édition*, Laurent Audibert, 2014.
- **La programmation orientée objet. Cours et exercices en UML 2 avec Java, C#, C++, Python, PHP et LINQ**, *6ème édition*, Hugues Bersini, 2014.

Génie Logiciel (Software Engineering) - motivation

- Systèmes informatiques :
  - **80 % de logiciel**
  - 20 % de matériel
- Depuis quelques années, la fabrication du matériel est assurée par quelques fabricants seulement.
  - Le matériel est relativement fiable.
  - Le marché est standardisé.

Constat : *les problèmes liés à l'informatique sont essentiellement des problèmes de logiciel*

Génie logiciel - pourquoi ?

- La "crise du logiciel" (années '60-'70)
- Les problèmes :
  - la fiabilité (les bugs ou cas non prévus)
  - manques de procédés de tests et vérification logicielle
  - le respect du cahier des charges du client (délais, coûts, spécifications etc.)
  - difficultés d'évolution
- Les causes :
  - une communication difficile
  - complexité croissante des logiciels
  - tests trop souvent insuffisants
  - trop de modifications
  - trop d'interrelations entre composants logiciels

Taux de réussite des projets

Étude sur 50000 applications (Standish Group, 2015) :

- Succès : 29 %
- Problématique : 52 % (budget ou délais non respectés, défaut de fonctionnalités)
- Échec : 19 % (abandonné)

MODERN RESOLUTION FOR ALL PROJECTS

	2011	2012	2013	2014	2015
SUCCESSFUL	29%	27%	31%	28%	29%
CHALLENGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

The Modern Resolution (OnTime, OnBudget, with a satisfactory result) of all software projects from FY2011-2015 within the new CHAOS database. Please note that for the rest of this report CHAOS Resolution will refer to the Modern Resolution definition not the Traditional Resolution definition.

CHAOS RESOLUTION BY PROJECT SIZE

	SUCCESSFUL	CHALLENGED	FAILED
Grand	2%	7%	17%
Large	6%	17%	24%
Medium	9%	26%	31%
Moderate	21%	32%	17%
Small	62%	16%	11%
TOTAL	100%	100%	100%

The resolution of all software projects by size from FY2011–2015 within the new CHAOS database.

Génie logiciel

- Comment faire des logiciels de qualité ?
- Qu'attend-on d'un logiciel ? Quels sont les critères de qualité ?

Définition

L'ensemble des méthodes, des techniques et outils concourant à la production de logiciel, au-delà de la seule activité de programmation.

L'art et la manière de créer un logiciel. Dépasse le cadre purement technique :

- développe les bonnes pratiques de conception, d'implémentation et de maintenance d'un logiciel
- permet d'obtenir une amélioration de la qualité du logiciel

Les qualités d'un logiciel

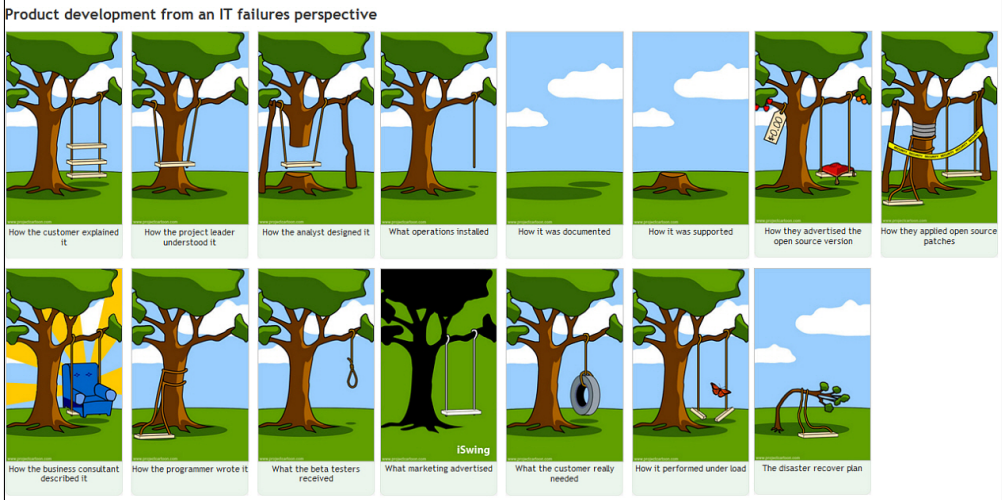
- Validité - correspond aux spécifications définies par le cahier des charges
- Fiabilité (robustesse) - gestion des conditions "anormales"
- Facilité d'utilisation (ergonomie)
- Extensibilité
- Réutilisabilité (en tout ou en partie)
- Compatibilité
- Efficacité (Performance) - utilisation optimale des ressources matérielles
- Portabilité - transfert sous différents environnement matériels et logiciels
- Intégrité - aptitude du logiciel à protéger son code et ses données contre des accès non autorisés
- Vérifiabilité - facilité de préparation des procédures de test

Processus de développement

Ce n'est pas simplement l'écriture du code ! ! !

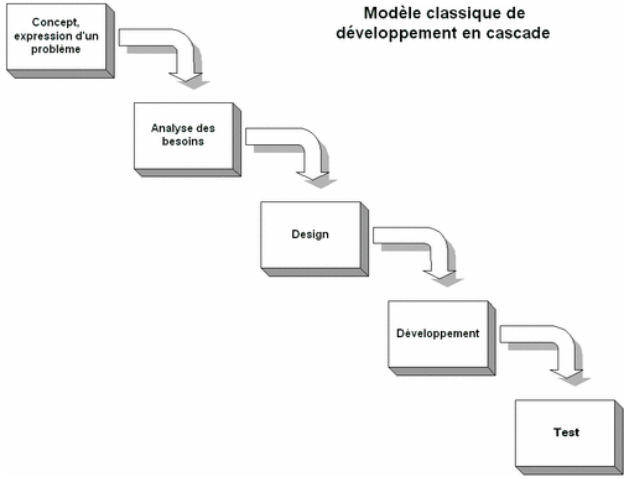
- Analyse des besoins (conceptuelle) – dans ce cours
- Modélisation (conception de l'architecture) – dans ce cours
- Implémentation – un peu ici, beaucoup en cours M2103
- Tests – dernière séance de TP + à fond dans M2103 et M2105
- Mise en exploitation (utilisation)
- Maintenance et évolution

Processus de développement



Processus de développement – en cascade

Chaque étape ne débute que lorsque la précédente est achevée.



Processus de développement – en cascade

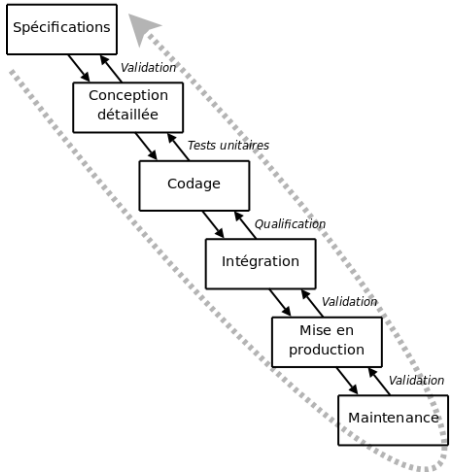
- Étapes successives
- Facile à mettre en place
- Planification simple

- Pas de communication entre les étapes
- Impossible de savoir quand une étape est "parfaite".
- Pas d'adaptabilité - effet tunnel

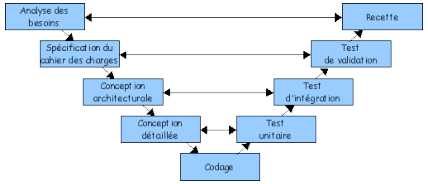
Solution limitée aux projets courts avec peu de participants

Modèle en cascade (amélioré)

- Étapes successives
- Facile à mettre en place
- Planification simple
- Toujours impossible de savoir quand une étape est "parfaite".
- La vérification du bon fonctionnement du système est réalisée trop tardivement : lors de la phase d'intégration, ou pire, lors de la mise en production.



Processus de développement – Modèle en V

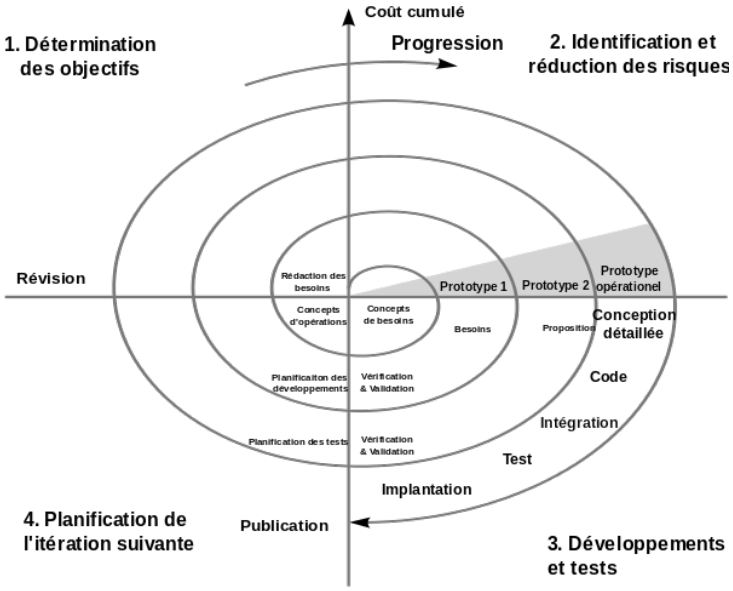


- Proposé comme solution pour le problème de réactivité du modèle en cascade
- Évite d'énoncer une propriété qu'il est impossible de vérifier objectivement après la réalisation
- Idéal quand les besoins sont bien connus, quand l'analyse et la conception sont claires

Inconvénients :

- Souffre toujours du problème de la vérification trop tardive du bon fonctionnement du système
- En pratique, le risque d'utilisation en cascade
- Comment séparer les étages ?

Processus de développement – Modèle en spirale (itératif)



Modèle en spirale – avantages

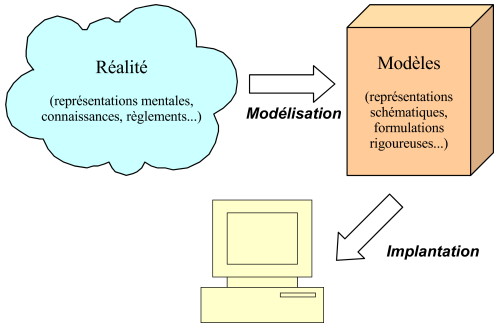
- Validation concrète et non sur documents
- Limitation du risque à chaque itération
- Client partenaire : retour rapide sur ses attentes
- Progressions : pas d'explosion des besoins à l'approche de la livraison : pas de ≪ *n'importe quoi pourvu que ça marche* ≫
- Flexibilité
- Planification renforcée
- Modèle objet : se prête parfaitement à une démarche incrémentale

Pourquoi modéliser ?

- les systèmes sont souvent trop complexes
- nous sommes incapables de les comprendre dans leur totalité
- le code ne permet pas de simplifier/abstraire la réalité

Modèle

- Une simplification de la réalité.
- Une *abstraction* centrée sur la représentation conceptuelle et physique d'un système.



Principes de la modélisation

Modéliser = abstraire la réalité pour mieux comprendre le système à réaliser.

Modéliser le processus de développement pour :

- bien répartir les tâches et automatiser certaines d'entre elles
- réduire les coûts et les délais
- assurer un bon niveau de qualité et une maintenance efficace

Modéliser un système avant sa réalisation pour :

- comprendre le fonctionnement du système
- maîtriser sa complexité
- assurer sa cohérence
- pouvoir communiquer au sein de l'équipe de réalisation

Approches de la modélisation

Approche **fonctionnelle**

- Approche traditionnelle utilisant des procédures et des fonctions
- On identifie les fonctions nécessaires à l'obtention du résultat
- Les grands programmes sont décomposés en sous programmes
  - hiérarchie de fonctions et sous-fonctions
  - approche descendante (Top-Down)
- Bref : c'est ce que vous avez l'habitude de faire

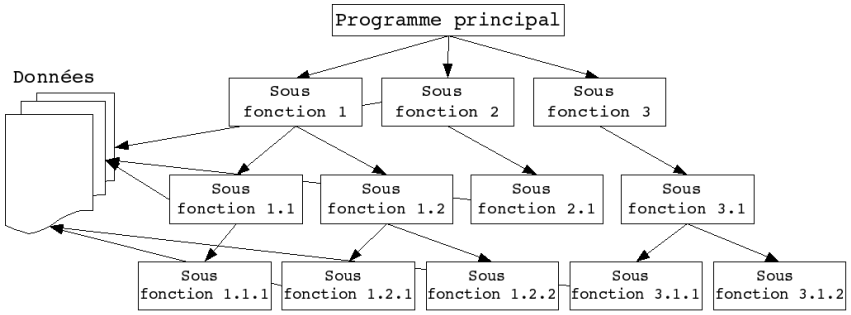
Approche **orientée objet**

1. On identifie les *entités* (objets) du système
2. On cherche à faire *collaborer* ces objets pour qu'ils accomplissent la tâche voulue.

Modélisation par décomposition fonctionnelle

Approche **descendante**

- Décomposer la fonction globale jusqu'à obtenir des fonctions simples à appréhender et donc à programmer.
- la **fonction** donne la **forme** du système.



Modélisation par décomposition fonctionnelle

Avantages

- Organisée, logique.
- Ordonnée, réduit la complexité.

Inconvénients

- Comment assurer l'évolution du logiciel ?
- Comment réutiliser les parties déjà développées ?
- Comment structurer les données ?

Modélisation orientée objets

La **Conception Orientée Objet** (COO)

- le système est découpé en "briques de base" – *les objets*
- la fonctionnalité du logiciel résulte de l'interaction des objets
- on regarde la réalisation des fonctionnalités à travers ces entités
- **approche ascendante**

La *structure* du problème conduit à l'architecture du logiciel.

Modélisation orientée objets

Avantages

- Simple : peu de concepts de base
- Raisonnement par abstraction sur les objets du domaine
- Souvent plus complète et plus proche et du monde réel
- Renforce l'extensibilité et la réutilisabilité

Inconvénients

- Parfois moins intuitive que l'approche fonctionnelle
- Pas de fils conducteur, nécessite une expérience pour être mise en place

La programmation orientée objets découle naturellement de la modélisation orientée objets

UML - Unified Modeling Language

Définition

UML est un **langage de modélisation** orienté objet standard qui permet de représenter (de manière graphique) et de communiquer les divers aspects d'un système informatique.

- Apparu au milieu des années '90 (G. Booch, I. Jacobson et J. Rumbaugh). La version actuelle - UML 2.2.
- UML n'est pas un langage de programmation et pas une méthode !
- C'est juste un ensemble de notations ayant comme base la notion d'objet.

UML - Motivation

- Modéliser un système des concepts à l'exécutable, en utilisant les techniques orientée objet
- Réduire la complexité de la modélisation
- Indépendant des langages de programmation
- Proche des humains...
- ... et des machines
  - Il existe des outils automatiques de génération de codes (Java, C++, C# etc.) à partir des diagrammes UML

Les outils de modélisation UML

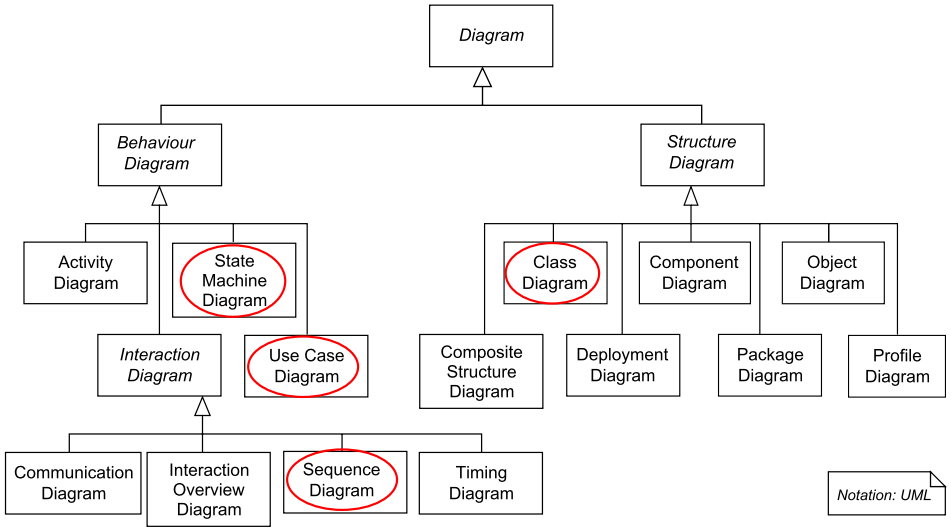
Les logiciels de modélisation UML sont nombreux.
Fonctionnalités principales :

- de modéliser tous les diagrammes UML, avec tous les composants.
- de naviguer facilement et naturellement entre ces diagrammes (organisation arborescente en paquetages)
- d'exporter les diagrammes pour les intégrer dans les documents de conception.
- production automatique de code, de documents, ...

Exemples : Umbrello, StarUML, Visual Paradigm, Rational Architect, etc.

En TP vous allez utiliser StarUML

Les diagrammes UML



Les diagrammes UML

Besoins des utilisateurs

- **diagramme des cas d'utilisation**

Aspect statique (vue structurelle) – représentation des données

- *diagramme objet*
- **diagramme de classes**

Aspect dynamique des objets – cycle de vie

- **diagramme État/Transition**
- *diagramme d'activités*

Interaction entre les objets (vue fonctionnelle)

- **diagramme de séquence**
- *diagramme de collaboration*

Besoins utilisateurs

Première étape UML d'analyse d'un système :

À quoi le système va servir ?

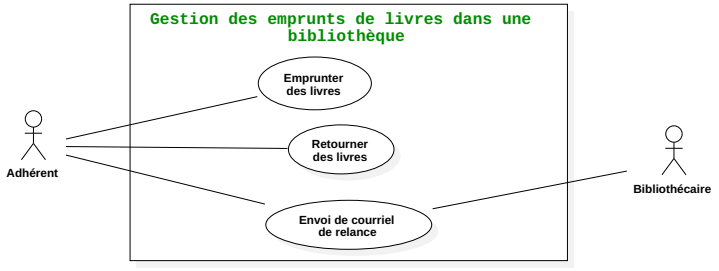
Souvent le maître d'ouvrage et les utilisateurs ne sont pas des informaticiens. Il leur faut un moyen simple d'exprimer leurs besoins.

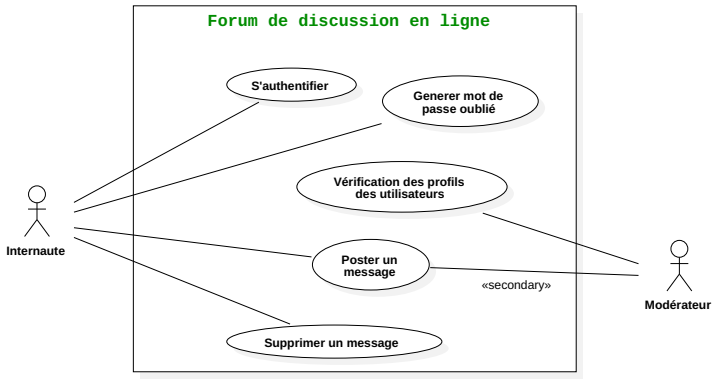
Solution : On fait des dessins !

Diagramme de cas d'utilisation (*Use Case*)

- Le système modélisé est (plus ou moins) une boîte noire
- On s'intéresse aux interactions entre le système et l'extérieur
- Les diagrammes des cas d'utilisation permettent :
  - de recueillir,
  - d'analyser,
  - d'organiser les besoins, et
  - de recenser les grandes fonctionnalités d'un système.

Use Case - exemples jouet





Les acteurs

Un **acteur** représente un *rôle* joué par une entité extérieure au système étudié et qui interagit avec celui-ci.

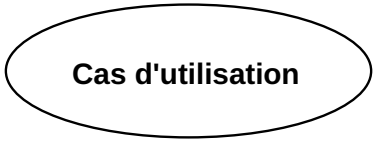
- Plusieurs entités peuvent être représentées par le même acteur
- Une même entité ayant plusieurs rôles peut être représentée par plusieurs acteurs

3 types :

- Les êtres humains
- Le matériel externe (les capteurs, imprimantes, ...)
- Les autres applications informatiques

Les cas d'utilisation

Un **cas d'utilisation** est une unité cohérente représentant une fonctionnalité visible de l'extérieur. Cette fonctionnalité souvent nécessite une série d'actions plus élémentaires.



C'est aussi un service rendu à un acteur. Donc :

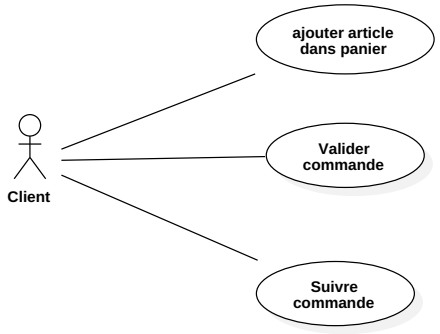
Un **cas d'utilisation** est l'expression d'un service réalisé de bout en bout, avec un déclenchement, un déroulement et une fin, pour l'acteur qui l'initie.

Trouver les cas d'utilisation

- *Comment* et *pourquoi* chaque acteur se sert du système ?
- un cas d'utilisation = des actions s'exécutant à un même moment, à l'aide des mêmes ressources, de la même façon
- *Séparer* en différents cas d'utilisation des actions déclenchées à des moments différents, des actions exigeant des ressources différentes pour s'exécuter, et des traitements de natures différentes (unitaire ou par lots)
- Pas de redondance
- Pas "trop" de cas d'utilisation – bon niveau d'abstraction
- Le temps ne doit pas être modélisé dans le diagramme

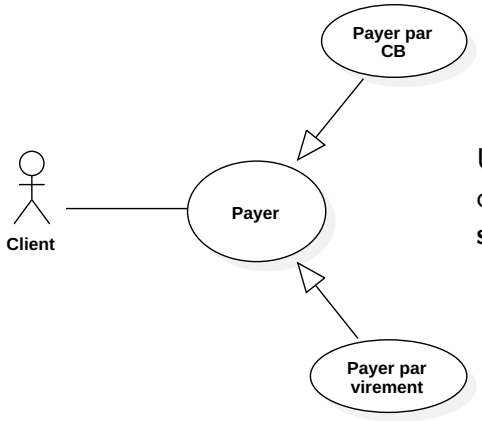
Relations

- Les acteurs sont reliés aux cas d'utilisation qu'ils peuvent déclencher
- Le trait représente des données qui sont échangées entre l'utilisateur et le système



- Pas d'échange entre deux acteurs ! ! ! **À votre avis, pourquoi ?**

Généralisation

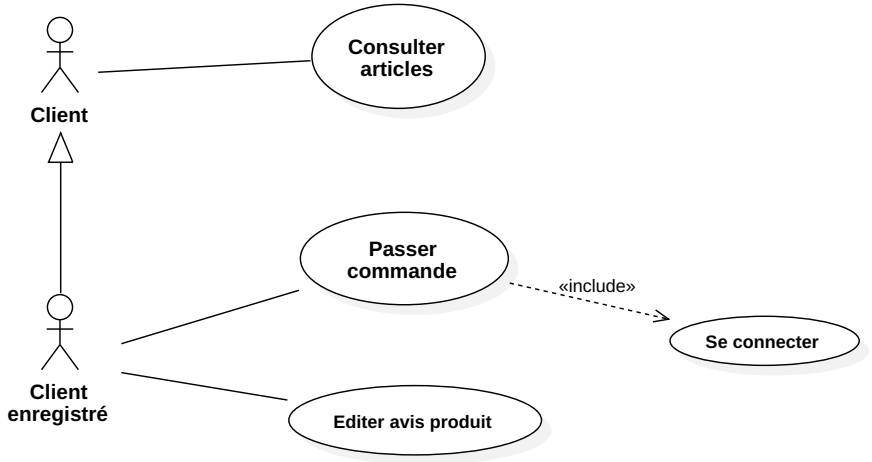


Un virement est un cas particulier de paiement. C'est une **sorte de** paiement.

- La flèche pointe vers l'élément général.
- Relation présente dans la plupart des diagrammes `UML`, se traduit par le concept d'**héritage** dans les langages orientés objet.

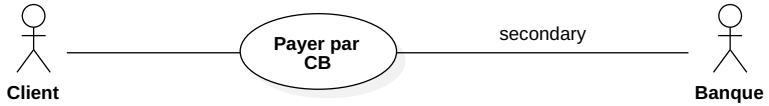
Relations entre acteurs

Une seule relation possible : la **généralisation**



Acteurs principaux et secondaires

- L'acteur est dit **principal** pour un cas d'utilisation lorsque l'acteur est à l'initiative des échanges nécessaires pour réaliser le cas d'utilisation.

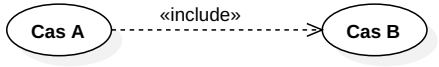


- Les acteurs **secondaires** sont sollicités par le système

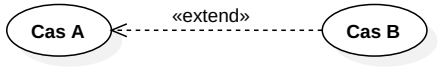
  Le plus souvent, ce sont d'autres systèmes informatiques avec lesquels le système développé est inter-connecté.

Relations entre cas d'utilisation

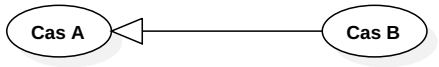
- **Inclusion** : le cas A inclut le cas B. Implique le déclenchement automatique de B sans intervention externe



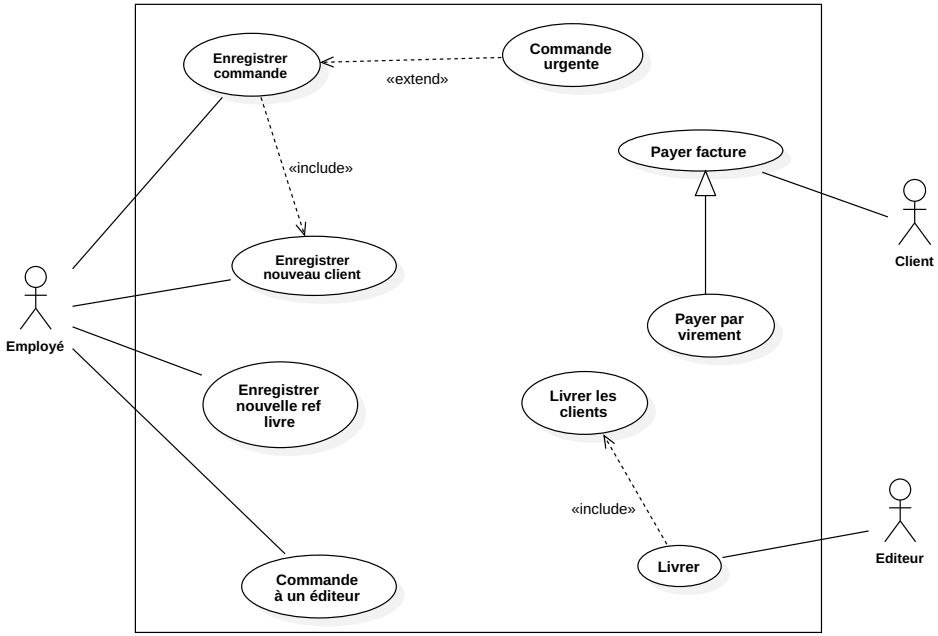
- **Extension** : le cas B étend le cas A *i.e.* B est un "cas spécial" de A



- **Généralisation** : le cas A est une généralisation du cas B (B est une sorte de A).



Exemple - système de vente de livres

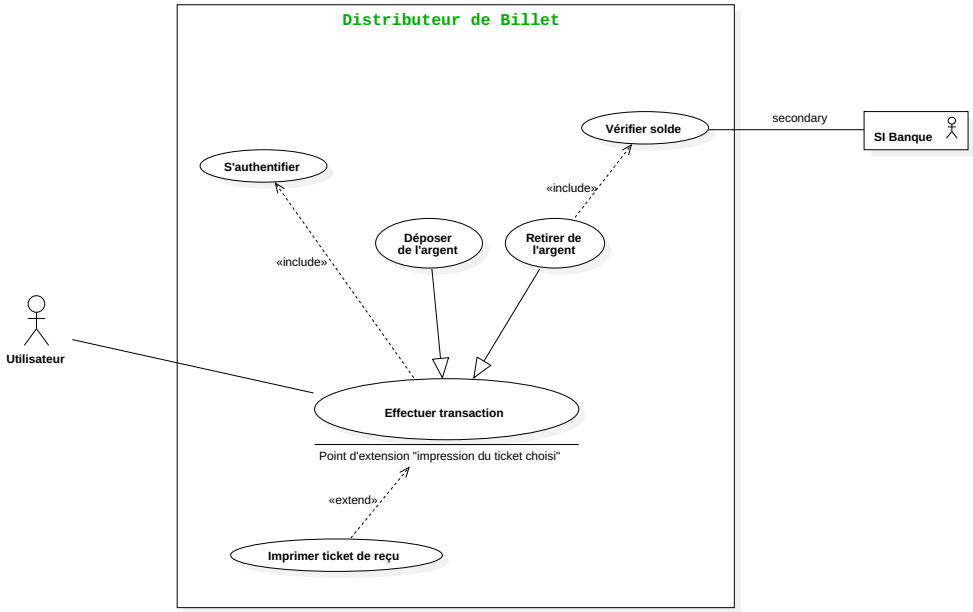


Dépendances d'inclusion et d'extension

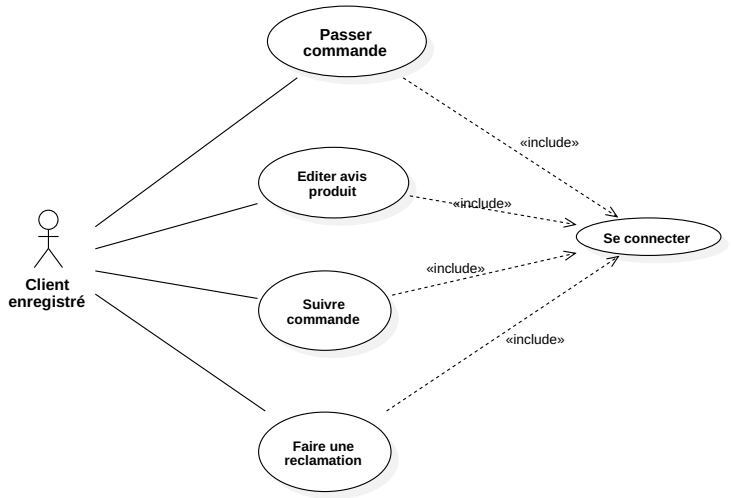
- Les inclusions et les extensions sont des relations de **dépendances**
  - Lorsqu'un cas A inclut un cas B, A dépend de B.
  - Lorsqu'un cas A étend un cas B, A dépend aussi de B.
- Lorsqu'un élément A dépend d'un élément B, toute modification de B sera susceptible d'avoir un impact sur A.
- Les termes ≪ *include* ≫ et ≪ *extend* ≫ sont des **stéréotypes**

Le sens des flèches pointillées indique une dépendance, pas le sens de la relation d'inclusion.

Les dépendances - exemple



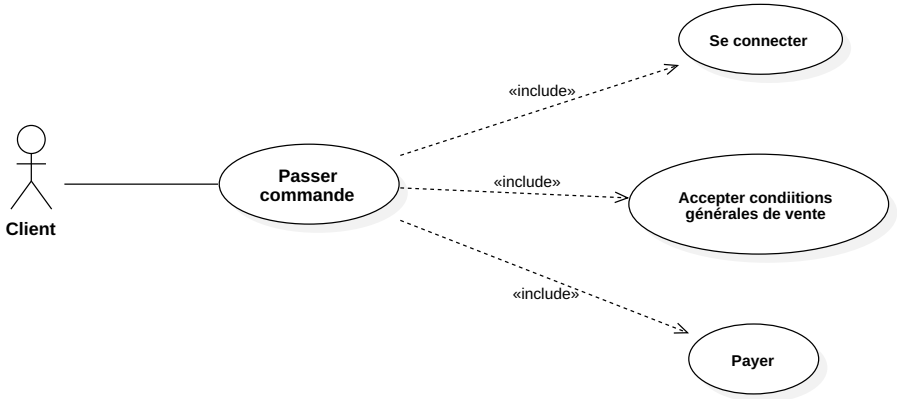
Réutilisabilité avec les dépendances



Inutile de développer plusieurs fois un module de connexion.

Décomposition grâce aux dépendances

Quand un cas d'utilisation fait intervenir un trop grand nombre d'actions élémentaires, on peut procéder à sa **décomposition** en cas plus simples.



Description des cas d'utilisation

- Le diagramme *Use Case* décrit les grandes fonctions du système du point de vue des acteurs
- Le diagramme n'expose pas de façon détaillée le dialogue entre les acteurs et les cas d'utilisation.
- Une simple expression est insuffisant pour décrire un cas d'utilisation : ambiguïtés, détails importants etc.

Chaque cas d'utilisation doit être documenté !

Description textuelle

1. Identification :
   - **Nom du cas** : Consulter le compte depuis internet
   - **Objectif** : Détailler les étapes permettant à un client à consulter son compte bancaire
   - **Acteurs** : Client, Banque (secondaire)
   - **Date** : 21/02/2018
   - **Responsables** : Marc Laporte
   - **Version** : 1.0
2. Séquencements :
   - Le cas d'utilisation commence lorsqu'un client demande la consultation de son compte
   - **Pré-conditions**

     Le client existe et son compte a été créé
   - **Enchaînement nominal**

     2.1 Le client s'authentifie
     2.2 Le système vérifie le login et le mdp ont été saisis
     2.3 Le système vérifie que le login et le mdp sont valides
     2.4 Le système affiche la page web du compte pour le client

Description textuelle - suite

- **Enchaînements alternatifs**

  a En (2.2) : si le login ou le mdp sont incorrectement saisis, le client est averti de l'erreur, et invité à recommencer
  b En (2.3) : si les informations sont erronées, elles sont re-demandées au client

3 Séquencements (suite) :
- **Post-conditions**
  - Le système stocke dans l'historique la date d'accès au compte du client

4 Rubriques optionnelles
- **Contraintes non fonctionnelles :**
  - Fiabilité : les accès doivent être sécurisés
  - Confidentialité : les informations concernant le client ne doivent pas être divulgués
- **Contraintes liées à l'interface homme-machine :**
  - Toujours demander la validation de la part du client

Diagramme des cas d'utilisation - rappel