UML / IHM

David Dupont

Rubika

2024/2025

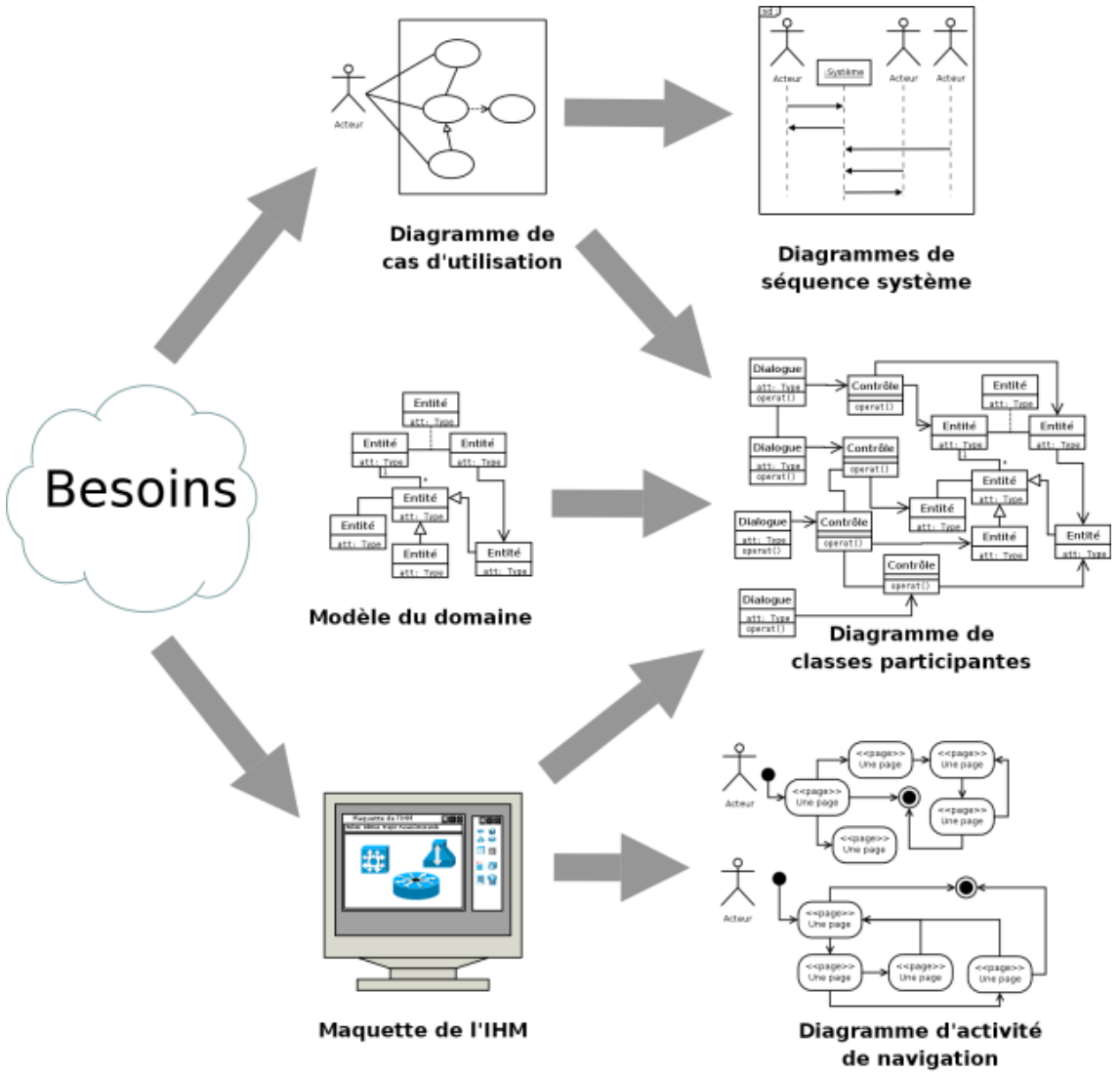
Organisation du module

- 7 journées ensemble
- Cours séparé en 3 parties
  - UML -> 1ere journée
    - Cours magistral le matin
    - Exercices l'après midi
  - HTML / CSS / Javascript -> journées 2, 3 et 4
    - Journée 2 : CM + exercices
    - Journées 3 & 4 : Projet
  - Frameworks -> journées 5, 6 et 7
    - Journée 5 : CM + exercices
    - Journée 6 & 7 : Projet

Plan du cours

- Intro
- Historique
- Pourquoi utiliser l'UML?
  - Conception
  - Rétro-ingénierie
- Avant propos : les objets
- Les types de diagrammes
  - Structuraux
  - Comportementaux
  - D'Interaction
- Exemple à travers des design pattern
- Comment on utilise l'UML dans un projet informatique?
- Bonnes pratiques de modélisation

UML

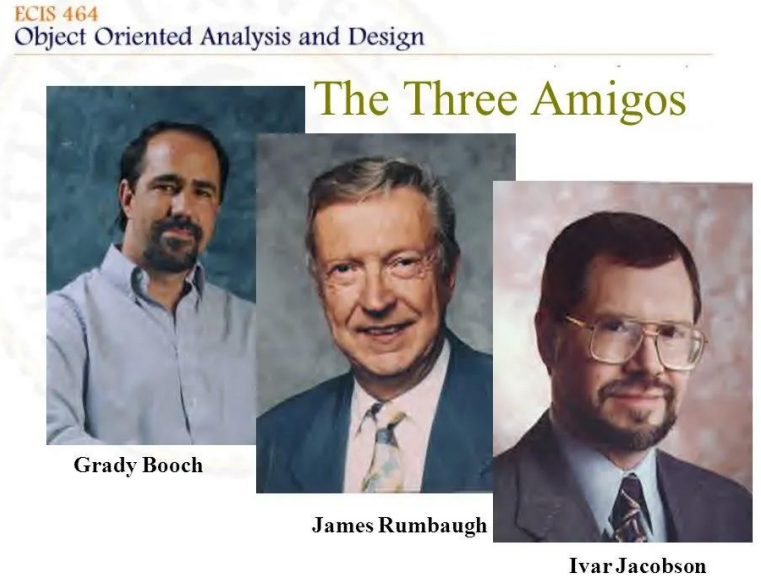


Intro : UML, C'est quoi?

- Unified Modelisation Language
- Représentation simplifiée de problèmes selon un point de vue
- Ensemble de diagrammes différents et complémentaires
- Destiné à décrire l'architecture, la conception et la mise en œuvre de systèmes logiciels complexes
- Principalement utilisé dans les applications orientées objets

Historique

- Créé par ces gars là ->



- En 1994, Association de :
  - The Booch Method
  - L'OMT (Object Modeling Technique)
  - OOSE (Object-Oriented Software Development)
- Présentation UML 1.0 en 1997 à l'OMG (object management groupe)
- UML 2.0 en 2005 avec de nouveaux diagrammes

Pourquoi utiliser l'UML?

- Prévoir avant de produire
- Réfléchir en amont pour anticiper des problèmes
- Gagner du temps au long terme
- Faire collaborer activement toutes les personnes du projet

Conception

- Une bonne architecture de base
- Réflexion autour des comportements des objets
- Anticiper les utilisations des clients finaux
- Documenter le projet

Rétro ingénierie

- Décomposer un code déjà existant
- Comprendre les relations et les dépendances des objets
- Documenter le projet
- Exemple:

```
1   public class Main {
2       public void main(String[] args) {
3           Animal habitant = new Animal ();
4           Zoo maubeuge = new Parc ();
5           maubeuge.ouvert = true;
6           maubeuge.mascoTe = habitant;
7           Zoo.capaciteMax = 76;
8           Zoo lille = new Musee ();
9           Parc.capaciteMax = 87;
10          lille.mascoTe = new Canin ( ) ;
11      }
12  }
13
14
```

Avant propos : La POO

- Programmation orienté objet
- Incontournable de nos jours
- Cf (https://www.wildcodeschool.com/fr-fr/blog/tout-ce-quil-faut-savoir-sur-les-10-langages-de-programmation-les-plus-utilisés) 4 des 6 langages les plus utilisés sont des langages orientés objets -> Python, Java, C#, C++
- Paradigme de développement qui permet de rassembler les éléments d'un même contexte dans un même endroit (un objet)

Procédural vs objet

- Procédural
  - Suite d'instructions à réaliser
  - Adapté à la programmation séquentielle car basé sur des fonctions
  - Utilisé par le langage C, COBOL, Pascal, PHP…

- Objet
  - Structure de l'application repose sur des objets
  - Objet est une représentation d'une entité (animal, livre, voiture…)
  - Objet = État + Comportement + Identité
  - Utilisé par le Java, Python, C++, C#…

Exemple Procédural

```c
int main(){
    int nombreViesMario = 2;
    int nombreViesLuigi = 3;
    char couleurMario = "rouge";
    char couleurLuigi = "vert";
    bool peutSauter = false;


    //si mario peut sauter, il esquive le trou, sinon il tombe
    printf("Un trou approche");
    if(peutSauter){
        nombreViesMario = 1;
    }
    return 0;
}
```

Même exemple avec l'objet

```java
public class Personnage {
    public int nombreVies;
    public String couleur;
    public boolean peutSauter;

    public Personnage(nombreVies, couleur, peutSauter){
        this.nombreVies = nombreVies;
        this.couleur = couleur;
        this.peutSauter = peutSauter;
    }
}

public static void main(String... args){
    Personnage mario = new Personnage(2, "rouge", false);
    Personnage luigi = new Personnage(3, "vert", true);

    System.out.println("un trou approche")
    if(mario.peutSauter){
        mario.nombreVies = 1;
    }
}
```

Les objets

- Un objet représente une entité
- Contient un ensemble d'éléments
  - Des attributs
  - Des constructeurs
  - Des méthodes
- Différence entre classe et objet:
  - Classe est le fichier descripteur
  - Objet est une instance de la classe -> possède une identité

Exemple de classe complet

État

Identité

Comportement

```java
public class Personnage {
    3 usages
    public String nom;
    3 usages
    public int nombreVies;
    3 usages
    public String couleur;
    4 usages
    public boolean peutSauter;

    public Personnage(String nom, int nombreVies, String couleur, boolean peutSauter){
        this.nom = nom;
        this.nombreVies = nombreVies;
        this.couleur = couleur;
        this.peutSauter = peutSauter;
    }
    public int getNombreVies() { return nombreVies; }
    public void setNombreVies(int nombreVies) { this.nombreVies = nombreVies; }
    public String getCouleur() { return couleur; }
    public void setCouleur(String couleur) { this.couleur = couleur; }
    public boolean isPeutSauter() { return peutSauter; }
    public void setPeutSauter(boolean peutSauter) { this.peutSauter = peutSauter; }

    public void saute(){
        if(this.peutSauter){
            System.out.println(this.nom + " saute, il esquive le trou");
        }else{
            System.out.println(this.nom + " ne sait pas sauter, il tombe");
        }
    }
}
```

Concepts applicables à l'UML

- L'encapsulation
- L'héritage
- Les interfaces / abstraction
- Polymorphisme

Encapsulation

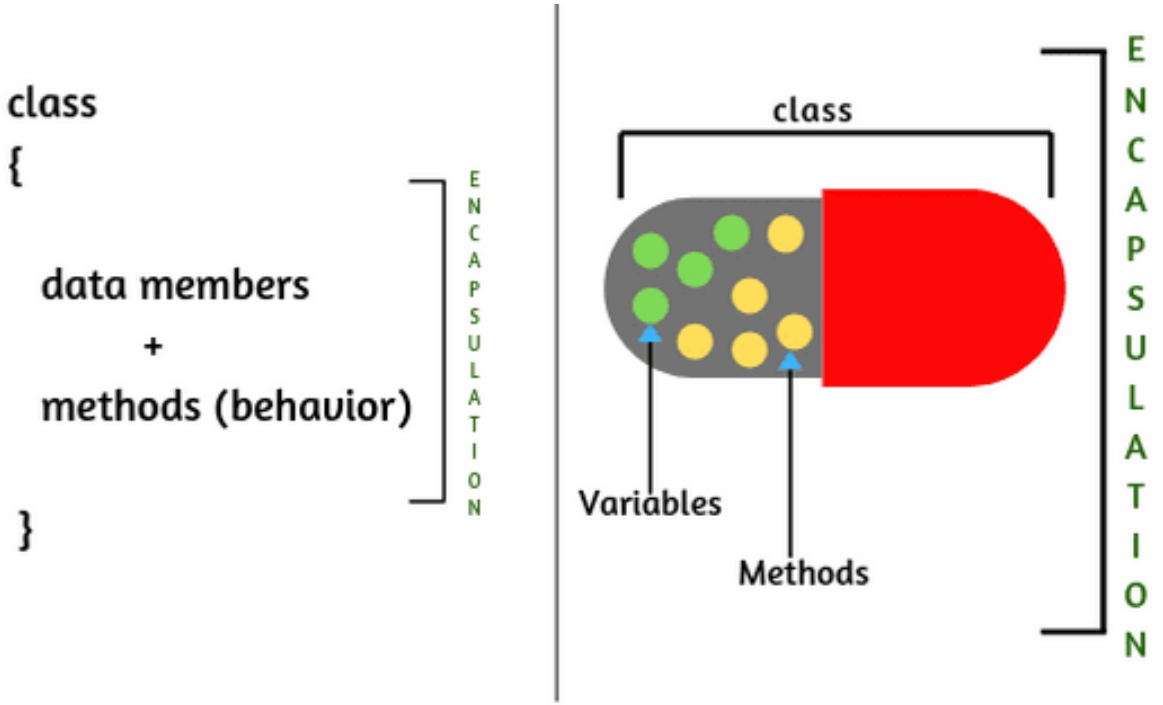


Fig: Encapsulation

- Cache les détails d'implémentation d'un objet
- Contrôle les accès aux données des objets

Héritage

- Principe permettant la création d'une classe (classe enfant) à partir d'une autre (classe parente)
- La classe enfant récupère automatiquement les attributs et comportement de la classe parente
- Permet de faire de
  - La généralisation : tout ce qui est commun se retrouve dans la classe parente
  - La spécification : chaque classe enfant peut avoir des attributs et méthodes spécifique

Interfaces / Abstraction

- « Boîte à outil » externe à une classe
- Rassemble un ensemble de constantes et de méthodes abstraites
- Abstraction: dit ce que la classe doit faire, mais ne dit pas comment
- Héritages et interfaces permettent le polymorphisme (bonne transition ☺)
- Exemple:

```
public interface Moteur {
    2 implementations
    void roule();
}
```

```
public class VoitureEssence implements Moteur {
    @Override
    public void roule() {
        System.out.println("Je roule à l'essence");
    }
}
```

```
public class VoitureElectrique implements Moteur {
    @Override
    public void roule() {
        System.out.println("Je roule avec une grosse batterie");
    }
}
```

Polymorphisme

- Principe permettant à un objet de prendre plusieurs forme
- Une classe héritée ou implémentée (via héritage ou interface) peut être manipulée par les classes utilisant cette dernière
- Elle aura la référence de la classe parente ou de l'interface mais aura le type de la classe enfant
- Ces types peuvent être interchangés si elle a la même référence de base

```java
public abstract class Animal {
    2 overrides
    public void cri(){
        System.out.println("l'animal cri");
    }
}
```

Classe parente (héritage)

```java
public class Chien extends Animal {
    @Override
    public void cri() { System.out.println("le chien fait wouf"); }
}
```

Classe enfant (le chien)

```java
public class Chat extends Animal {
    @Override
    public void cri() { System.out.println("le chat fait miaou"); }
}
```

Classe enfant (le chat)

```java
public class Renard {
    public void cri() { System.out.println("What's the fox say?");
}
```

Classe qui n'hérite pas de Animal

```java
public class Main {
    public static void main (String... args){
        Animal animalPolymorphe = new Chien();
        animalPolymorphe = new Chat();
        animalPolymorphe = new Renard();
    }
}
```

- Animal est la classe parente, et on déclare une nouvelle variable avec cette référence
- Comme Chien et Chat héritent de Animal, l'objet animalPolymorphe peut interchanger de type entre les deux
- Dans ce cas, le comportement de la méthode cri() changera avec celui du type associé
- Néanmoins, le renard n'héritant pas de Animal, le polymorphisme n'est pas possible avec cet classe

Retour à l'UML

Des questions?

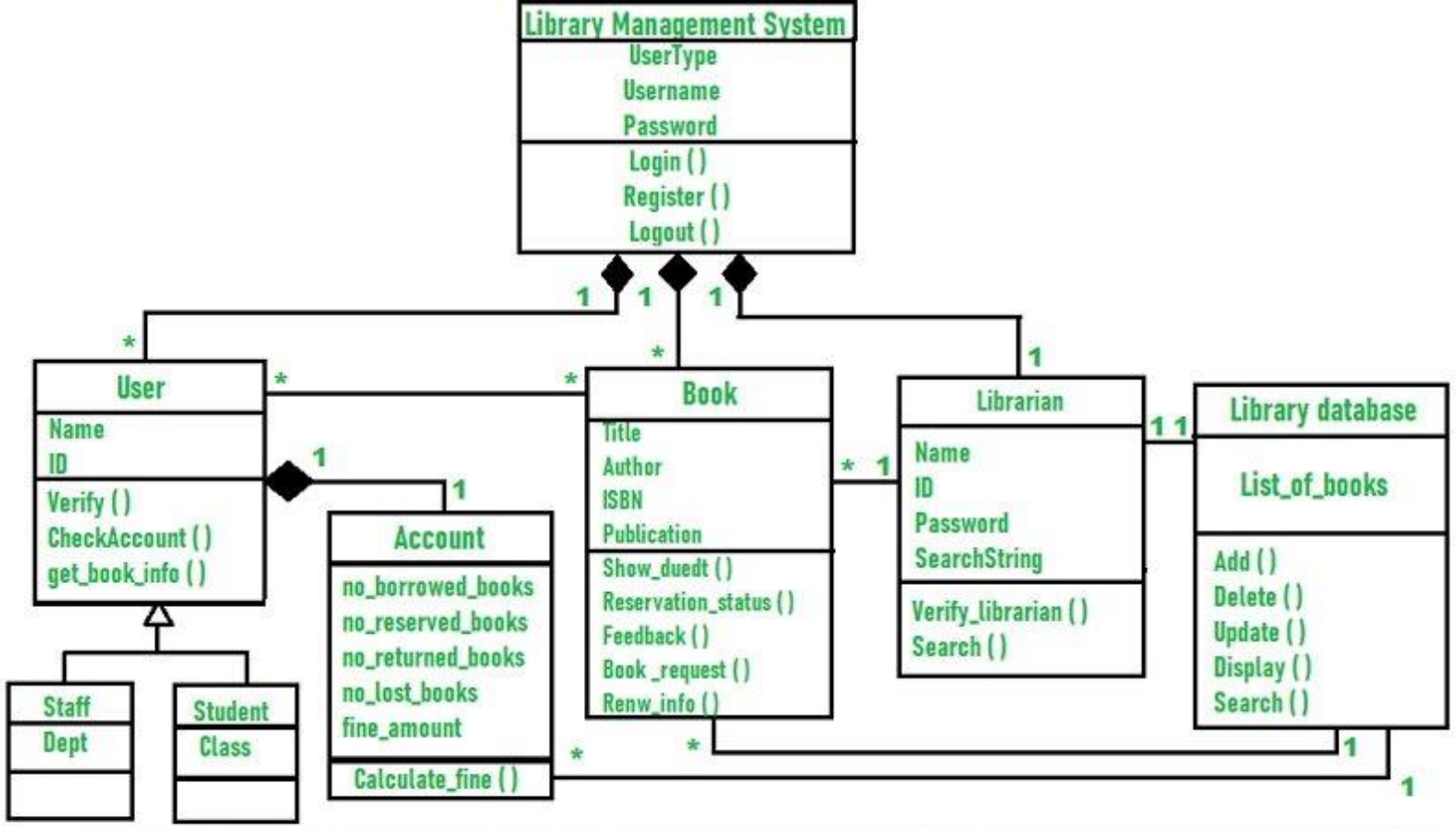
Les diagrammes

- Principaux outils de l'UML
- Servent à la modélisation de solutions apportées à certains problèmes complexes (ou non)
- Les grands intérêts des diagrammes: Spécification, visualisation, documentation et la communication
- 3 types de diagrammes:
  - Structuraux
  - Comportementaux
  - Interactions

Les diagrammes structuraux

- Représentation statique d'une architecture logicielle
- Liste les entités et leur liens entre eux
- Différents diagrammes: Classes, objets, composants, déploiement, composite, packages

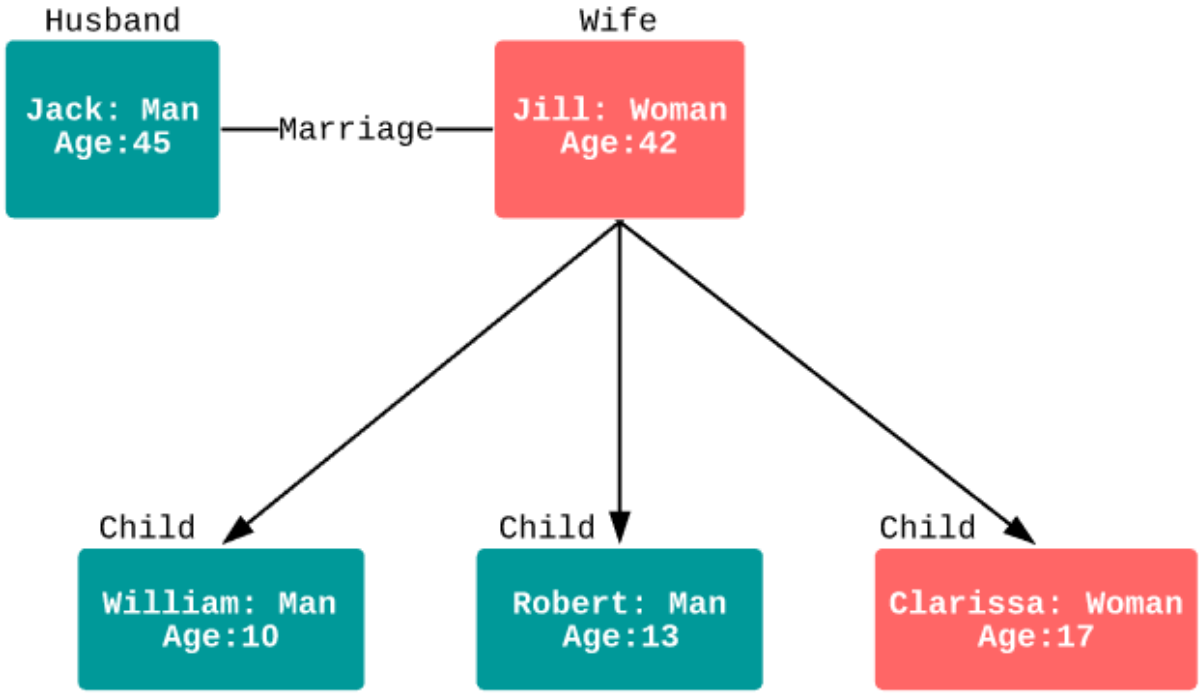
Diagramme de classes



CLASS DIAGRAM FOR LIBRARY MANAGEMENT SYSTEM

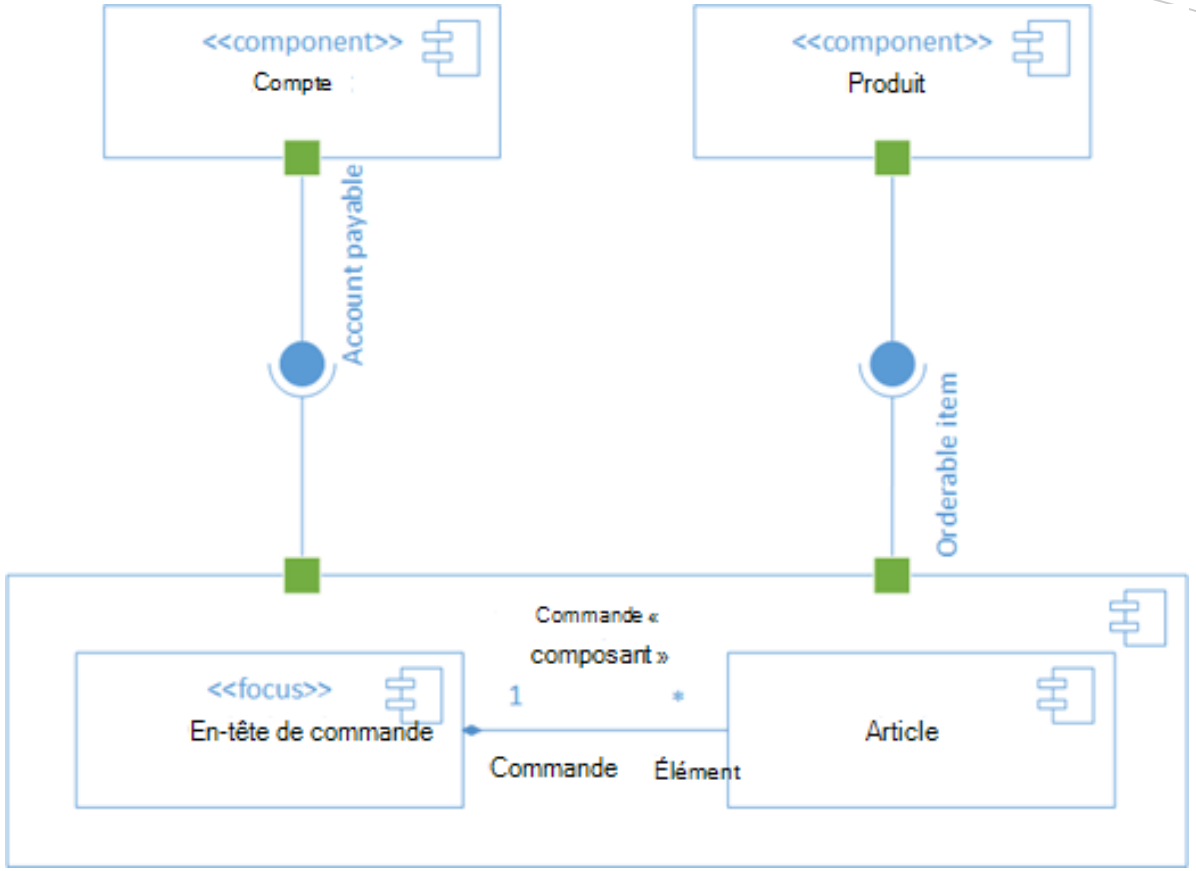
- Définition des entités :
  - Nom
  - Attributs
  - Opérations
- Et des relations :
  - Navigabilité (traits pleins avec losanges pleins)
  - Généralisation (flèches blanches)
  - Cardinalités (0,1, *)

Diagramme d'objets



- Ne se focalise pas sur la classe mais sur les instances
- Les entités sont toutes du même type, seules les attributs ont une valeur changeante
- On perd la notion de cardinalité et la généralisation, mais on peut garder la navigabilité
- Utilité:
  - Représenter un S.I. à un instant T
  - Schématiser un scénario à des fins de test ou de debug

Diagramme composants

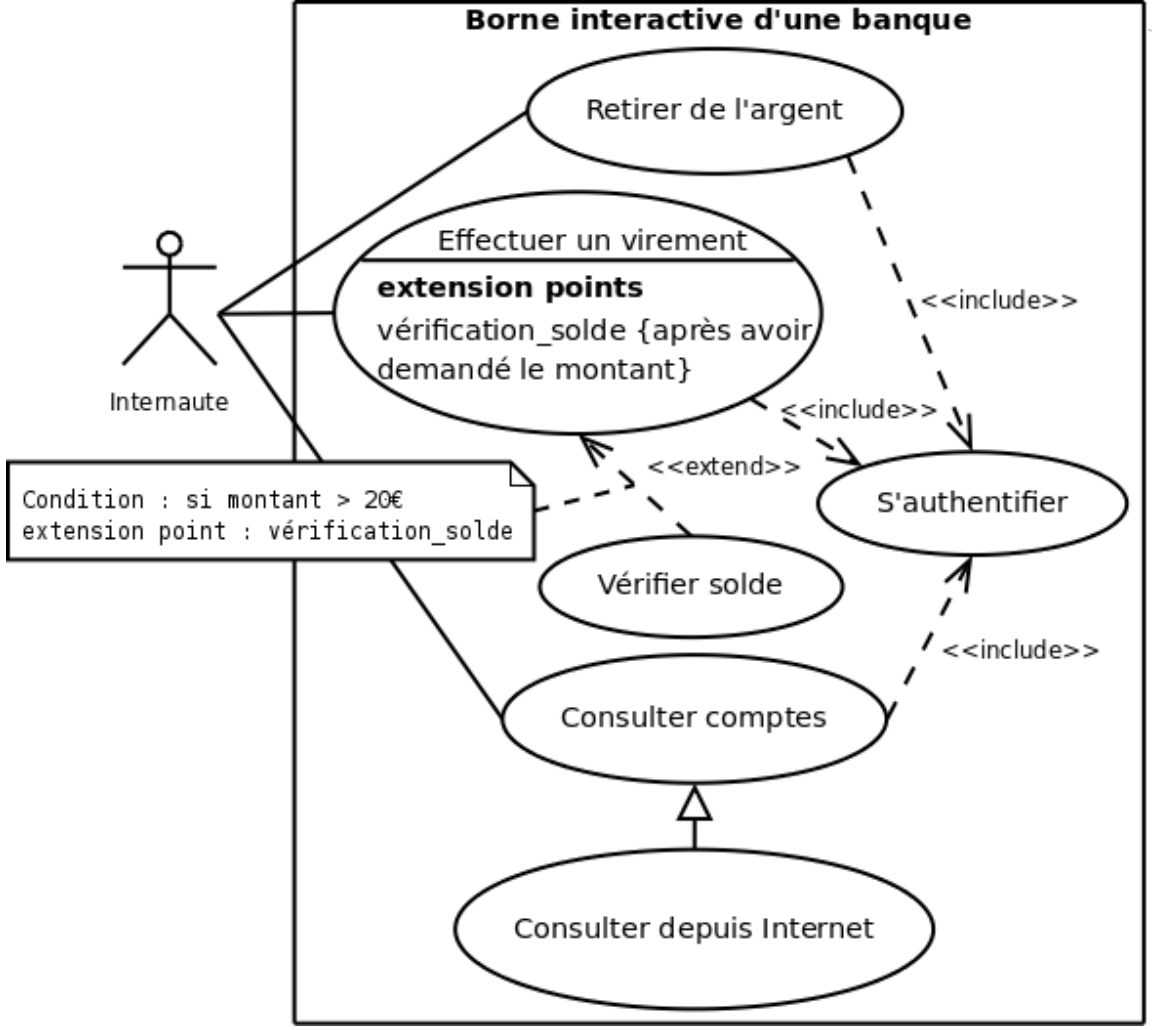


- Vue par composant (module / bibliothèque / fonctionnalité)
- Donne les interactions entre chaque composants à travers les connecteurs (interfaces requises / fournies) et les ports
- Utilité
  - Voir le S.I. sur un haut niveau
  - Prévoir une séquence de déploiement

Diagrammes comportementaux

- Modélise les fonctionnalités attendues par les entités du S.I
- Montre les interactions entre les objets et les acteurs
- Bien sûr, toujours des utilités de documentation et de communication
- Quelques diagrammes : activité, séquence, cas d'utilisation, état-transition, timing

Diagramme de cas d'utilisation



- Diagramme montrant les actions possibles d'un utilisateur dans un système
- On y voit les acteurs, les services avec leurs dépendances entre ces derniers
- Relation d'inclusion : un service en appelle un autre
- Relation d'extension : un service est conditionné par un autre

Diagramme d'activité

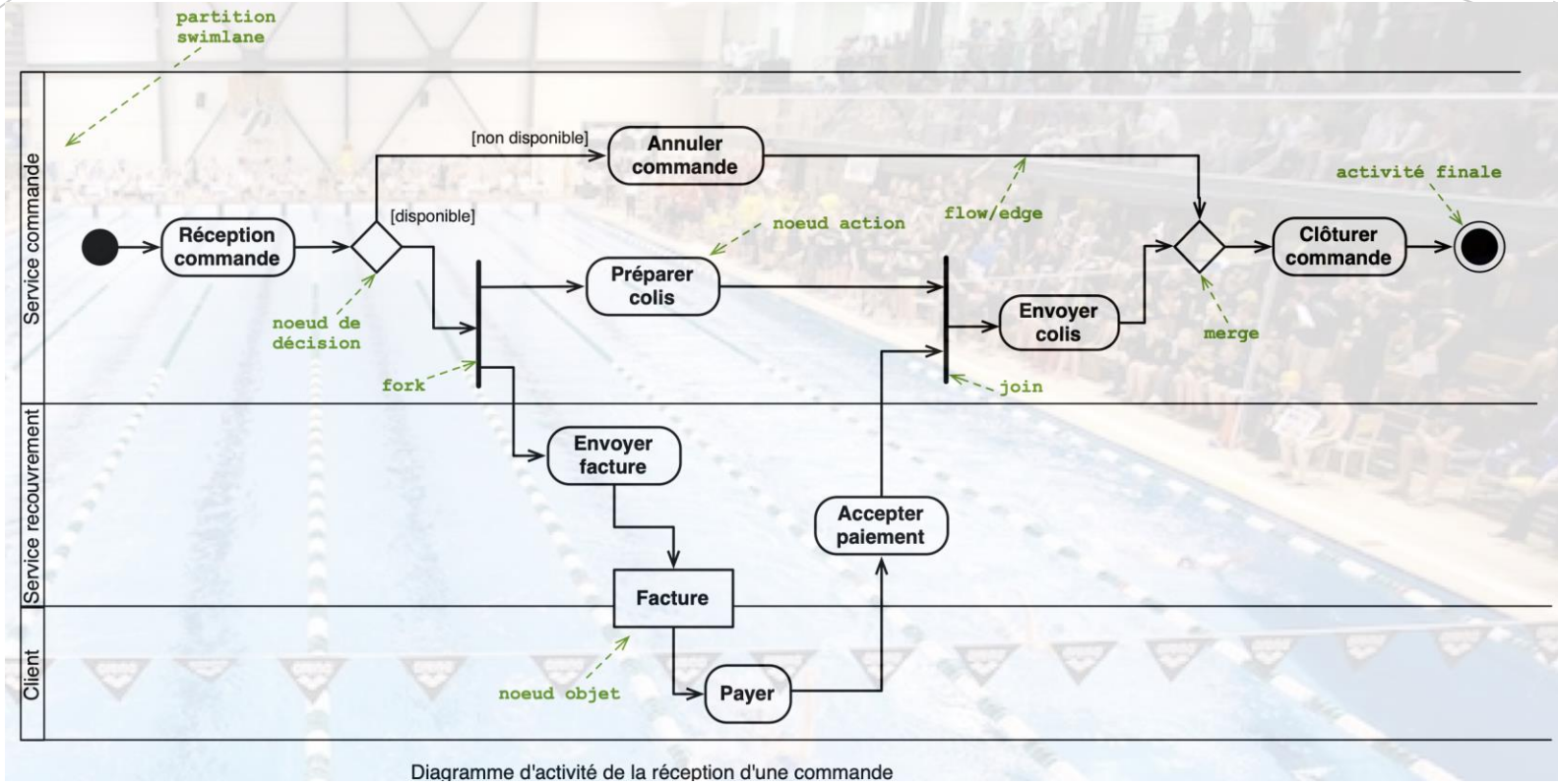
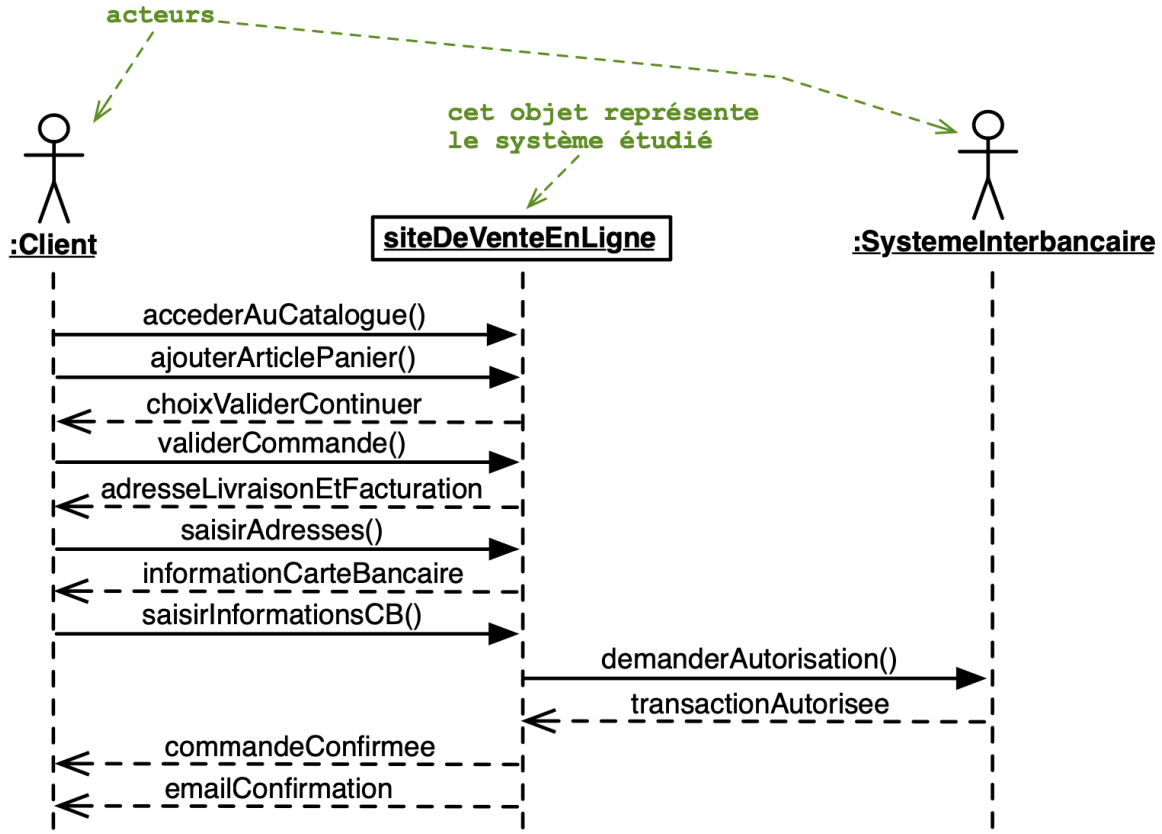


Diagramme d'activité de la réception d'une commande

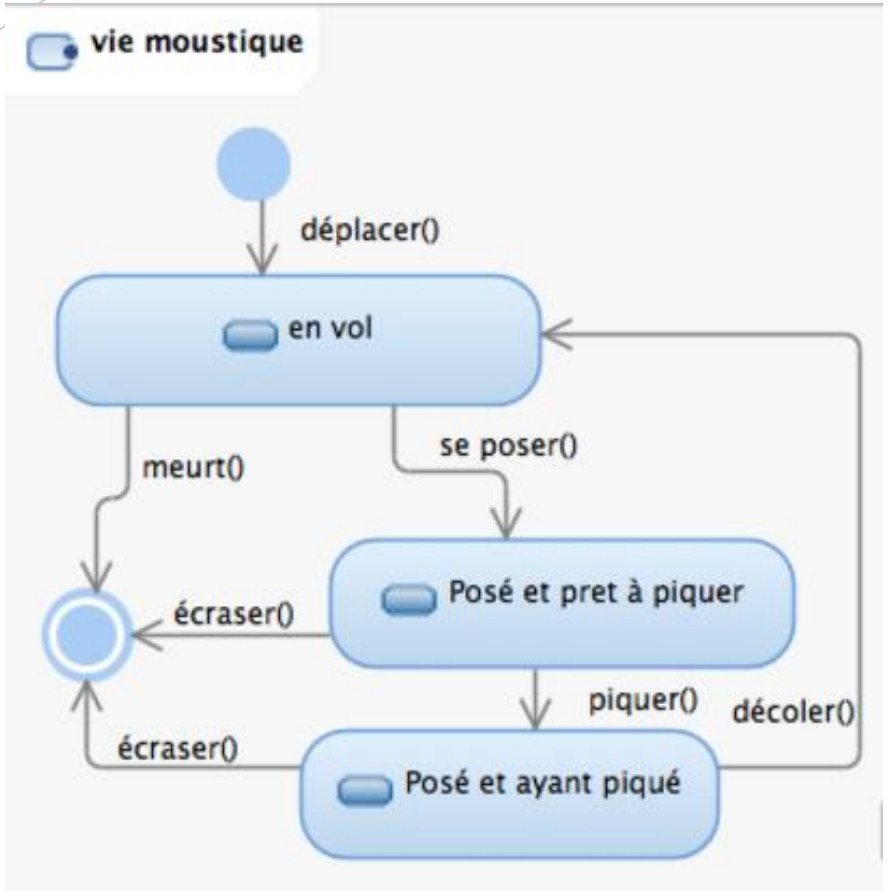
- Créé pour avoir une vision des séquences d'opérations
- On y voit les comportements de chaque acteurs pour réaliser une tâche
- On suit l'évolution de la tâches à travers les symboles de début, fin, des activités, des transitions des nœuds de décision et des barres de synchronisation

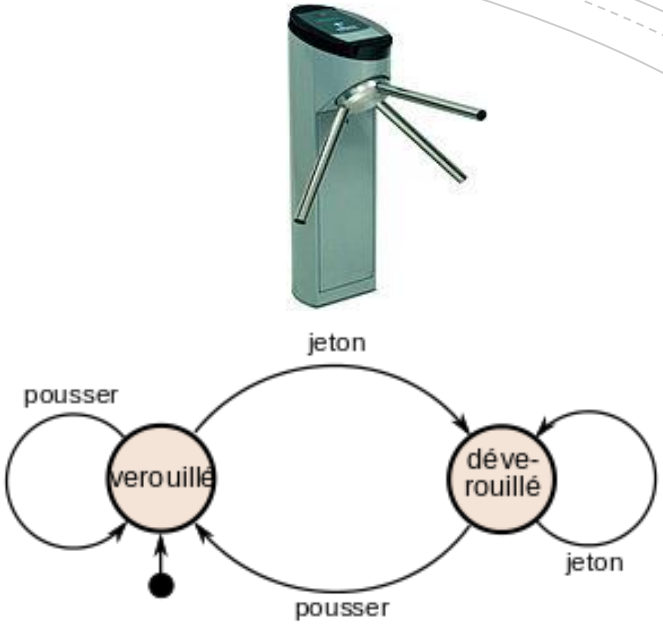
Diagramme de séquence



- Met l'accent sur la chronologie des étapes à réaliser et sur les interactions entre les acteurs
- Plus on va vers le bas du diagramme, plus on avance dans le traitement
- Chaque acteur est représenté par une colonne et on suit le traitement avec les flèches pleines ou en pointillés.
- Possibilité de faire des boucles ou des if/else avec des encadrés

Diagramme État-Transition





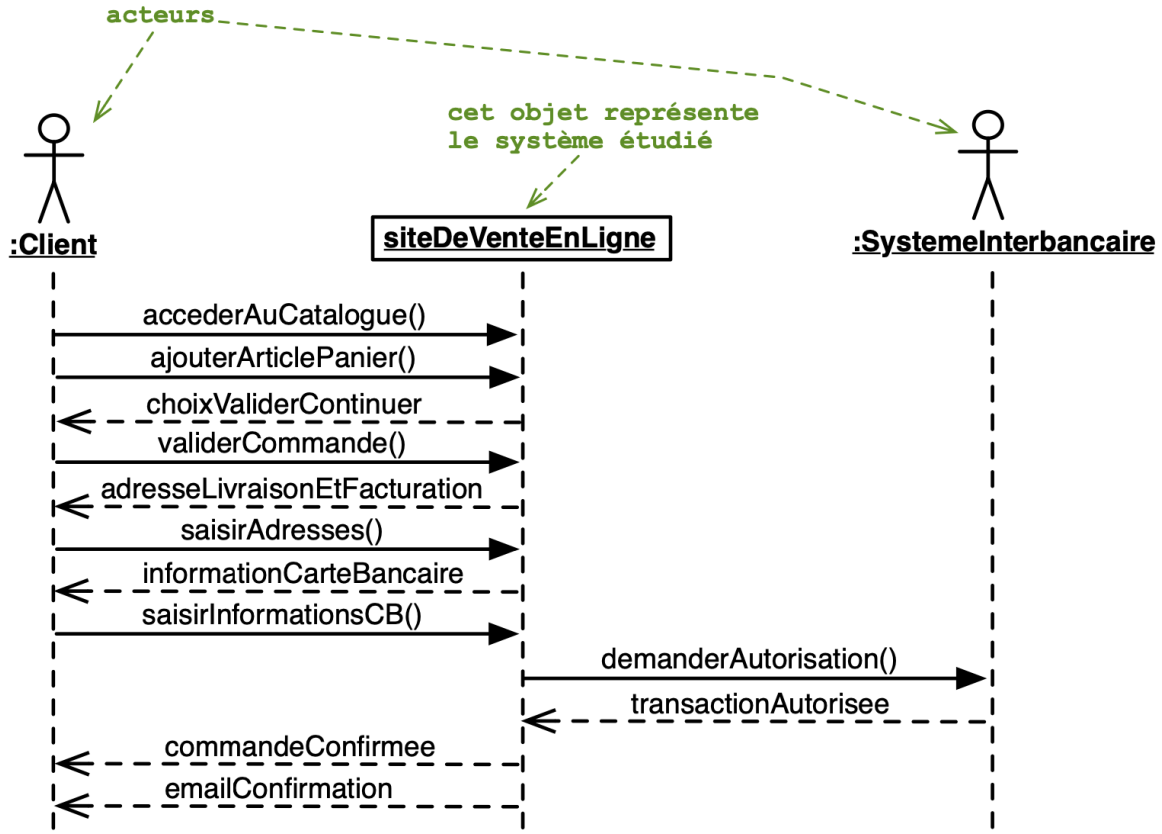
Automate

- Représentation de systèmes/objets qui ont un comportement différent en fonction de leur état en cours
- Comme dans d'autres diagrammes, on retrouve les points de commencement et de fin de processus, les transitions et ici les rectangles représentent les états du système décrit.

Diagrammes d'interaction

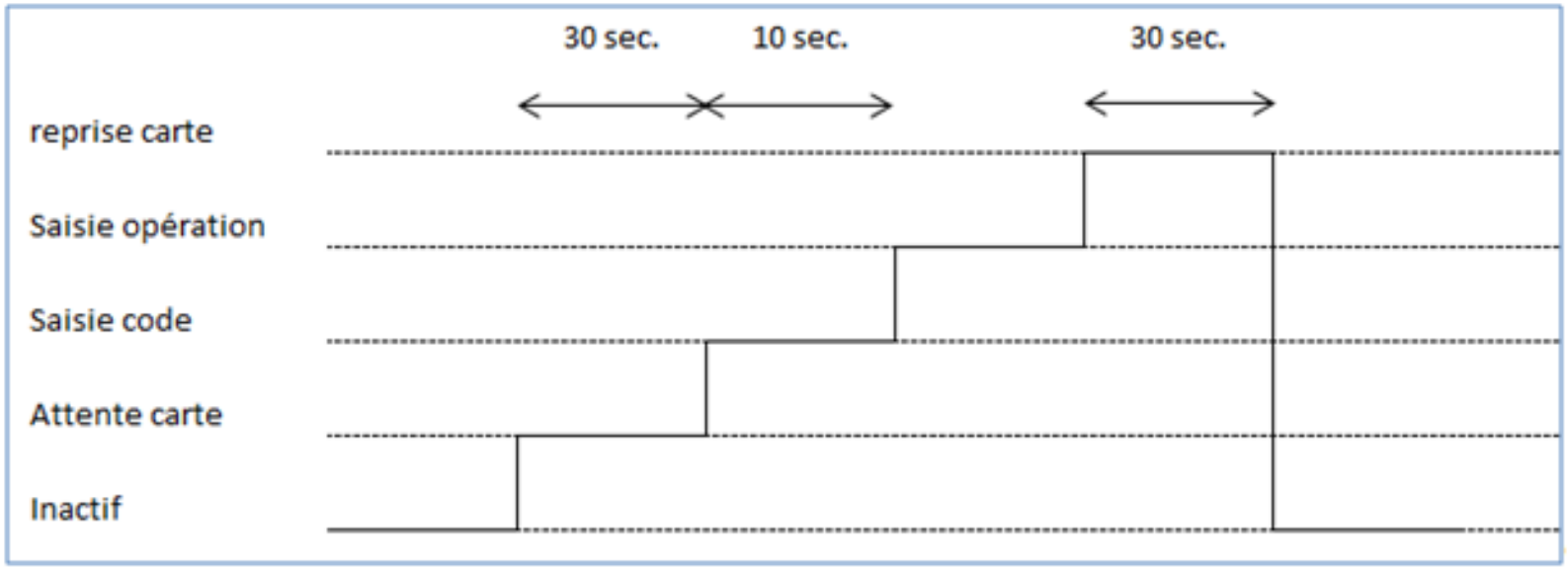
- Sous ensemble de diagrammes montrant les interactions entre les systèmes/fonctionnalités
- Mettent l'accent messages échangés et la temporalité de la séquence
- Les diagrammes : séquence (comme le comportemental), communication, de temps, diagramme global d'interaction

Diagramme de séquence (qui suit?)



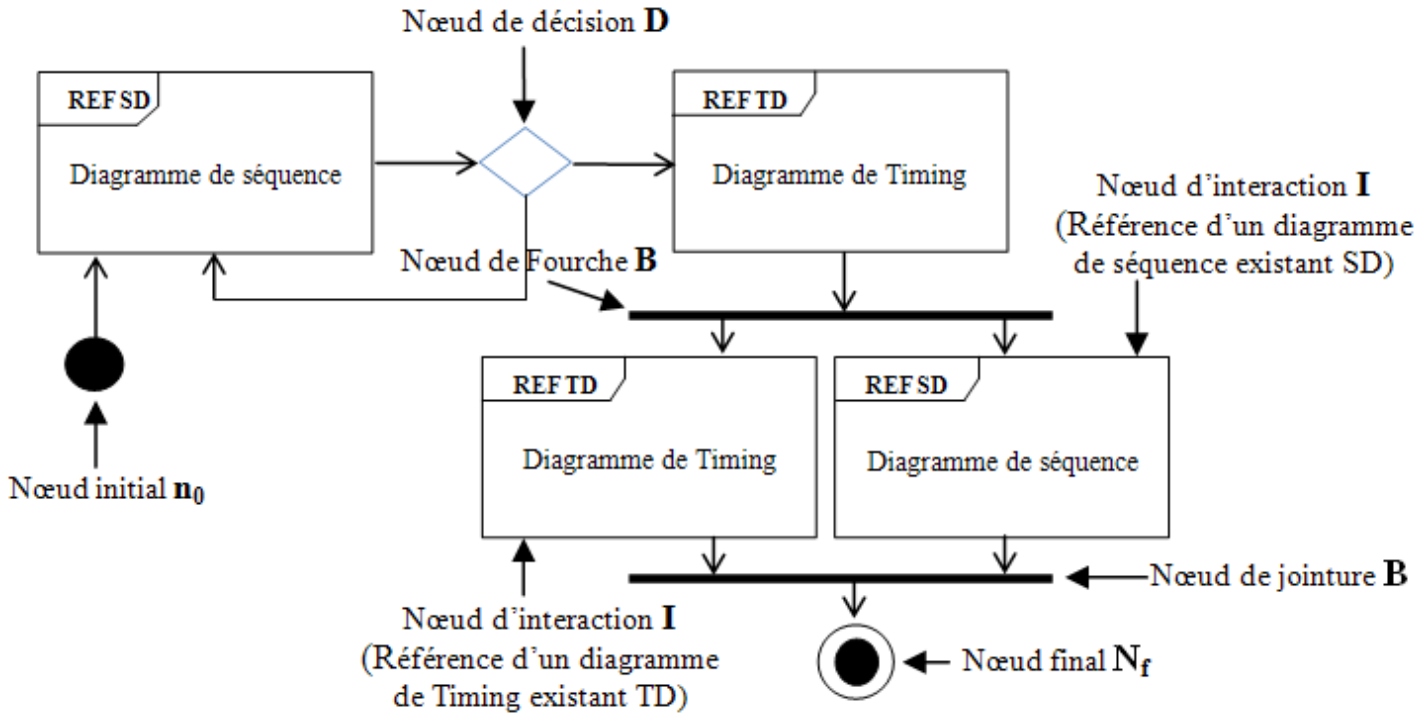
- Met l'accent sur la chronologie des étapes à réaliser et sur les interactions entre les acteurs
- Plus on va vers le bas du diagramme, plus on avance dans le traitement
- Chaque acteur est représenté par une colonne et on suit le traitement avec les flèches pleines ou en pointillés.
- Possibilité de faire des boucles ou des if/else avec des encadrés

Diagramme de temps



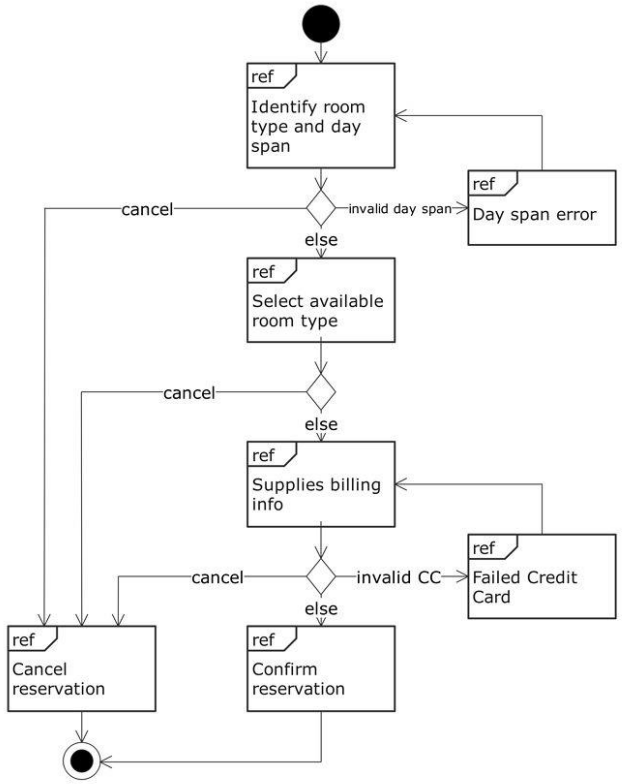
- Visualisation sur le temps passé par chaque tâche
- Souvent utilisé dans les systèmes temps réel ou dans les systèmes embarqués (IoT)
- Chaque composant/acteur (sur la verticale) possède sa ligne de vie en fonction du temps (en horizontal)
- Le trait plein représente la tâche exécutée et effectue des transitions entre les composants (barres verticales) un fois que la durée de traitement est passée

Diagramme d'interaction global



- Très ressemblant au diagramme d'activité mais ne fait pas apparaitre les acteurs
- Approche hiérarchique car composé de plusieurs sous-diagrammes

Exemple sur une réservation de chambre d'hôtel



- Chaque transition porte un action pour la décrire
- Chaque carré (activité) peut être un sous diagramme, ex : diagramme de séquence pour la sélection d'une chambre, ou de cas d'utilisation pour le paiement

Instant exercices

Exercice

- Modéliser un jeu d'échec ! ♟
- Faites un diagramme de structure et un diagramme comportemental

Règles

- Un jeu d'échec possède un plateau avec des cases noires et blanches
- 2 joueurs s'affrontent pour essayer d'éliminer le roi
- Chaque joueur possède un temps limité pour jouer, le premier joueur à dépasser 5 minutes de jeu perd
- Chaque fois qu'un joueur joue, son chronomètre se met en pause et celui de l'adversaire commence le décompte
- Chaque joueur possèdent 8 pions, 2 tours, 2 fous, 2 cavalier, un roi et une dame
- Chaque pièces peut se déplacer sur le plateau et de sa propre façon (pion vont tout droit, les fous en diagonal etc…)
- Chaque pièce peut éliminer une pièce adverse si son déplacement le permet (attention au pion qui élimine en diagonal)
- Si le roi est dans la ligne de mire d'une pièce adverse, il est en échec et si aucun coup ne peut le sauver, il s'agit d'un échec et mat et la partie se termine
- Un joueur peut abandonner la partie et celle ci se termine également
- Si au bout de 6 tours, les mêmes mouvements se répètent, la partie se termine en match nul
- D'autres règles?

L’UML à travers des design pattern

En français : des patrons de conception

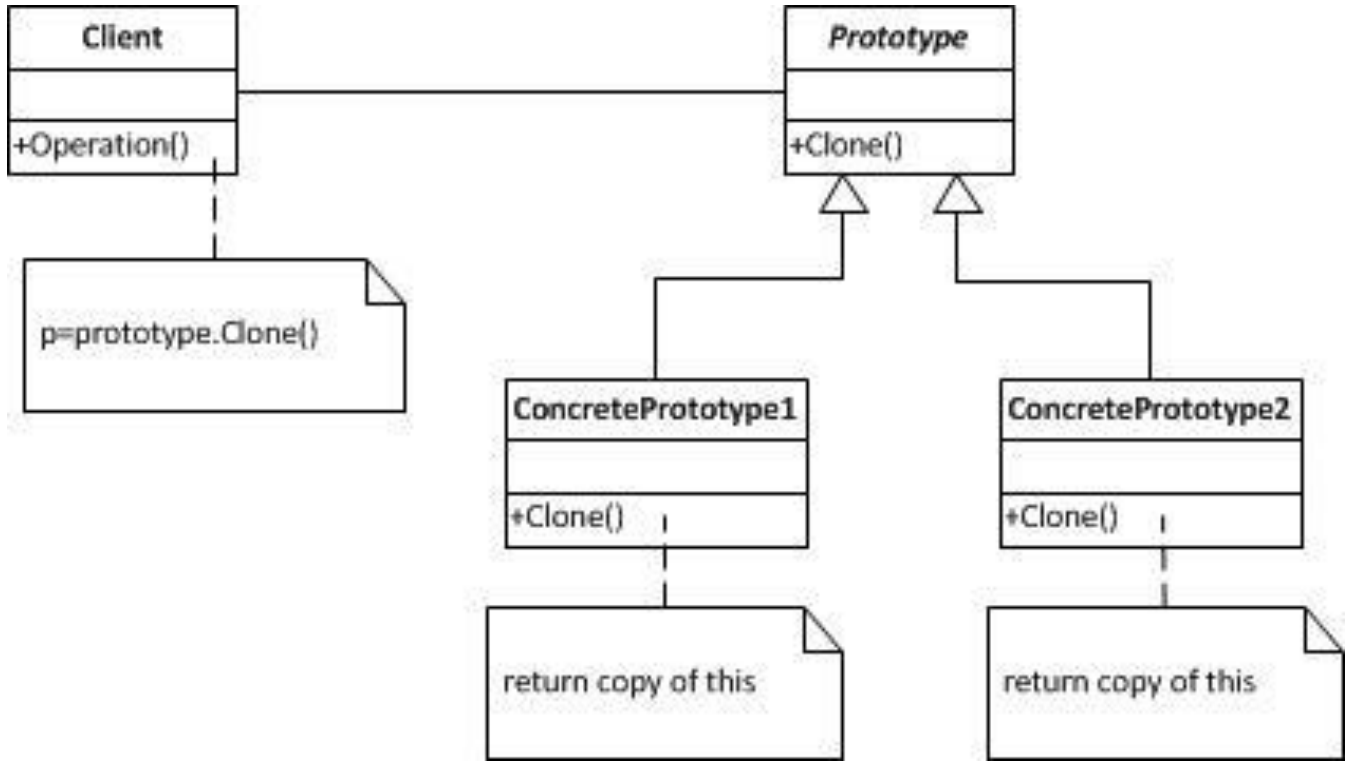
C’est quoi un design pattern?

- Solutions apportées à des problèmes de conception dans le développement logiciel
- Approche de réflexion sur des modèles ou structures
- Avantages:
  - Solutions standardisées
  - Maintenance améliorée
  - Évolution du code facile
  - Documentation/communication
- Différents types de design patterns
  - Les créationnels
  - Les structuraux
  - Les comportementaux

Les créationnels

- Permettent de donner des méthodes facilitant la création des objets dans une structure
- Découple l'instanciation des objets de leur utilisation
- Quelques exemple de design patterns:
  - Singleton
  - Factory
  - Builder
  - Prototype

Prototype

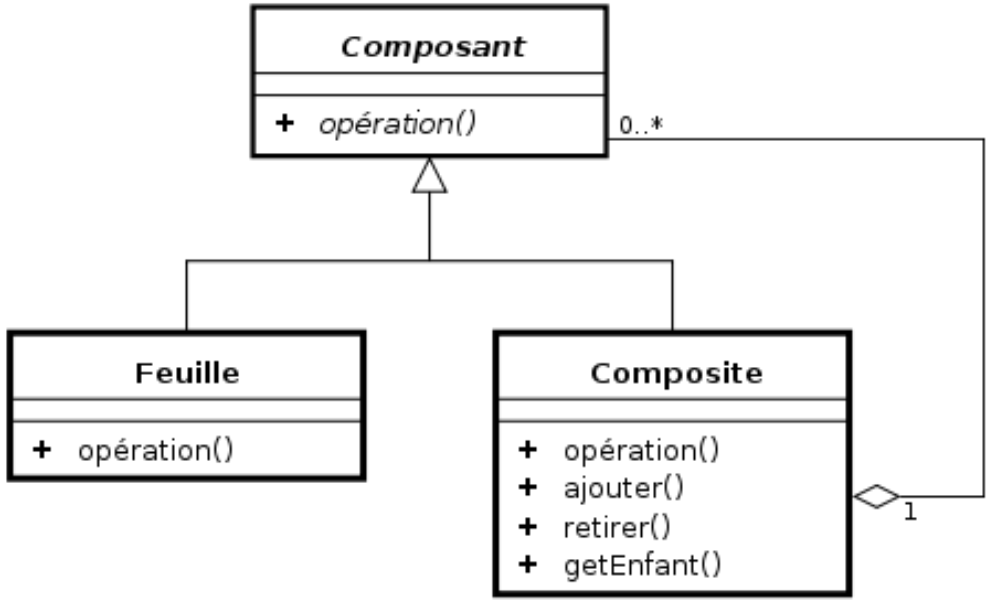


- Problématique: On veut copier des entités
- Solution naïve? On prend une entité du même type, et on copie toutes ses caractéristiques
- Malheureusement toutes les informations ne sont pas tout le temps accessibles (données cachées)
- Objectif du design pattern : demander à l'objet lui-même de se dupliquer car il connaît toutes ses informations

Les pattern structuraux

- Répond à des problématiques d'organisation et d'assemblage d'entités
- Gère les interactions entre les objets pour augmenter la flexibilité et la réutilisabilité des fonctionnalités
- Quelques exemples:
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Proxy
  - Strategy

Pattern Composite

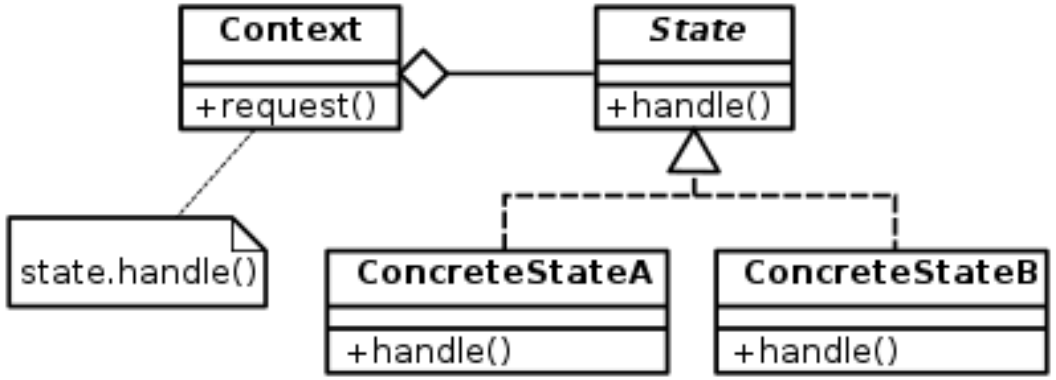


- Problématique : On veut parcourir une structure arborescente.
- Exemple: calculer le prix de tous les produits contenus dans une boîte. Dans cette, on peut avoir des produits et d'autres boîtes, comment faire?
- But de Composite : On fait un organisation permettant de traiter tous les objets de la structure de la même manière
- Dans l'exemple: on déballe le carton, si c'est un produit, on récupère son coût, si c'est une boîte on recommence l'opération

Les pattern comportementaux

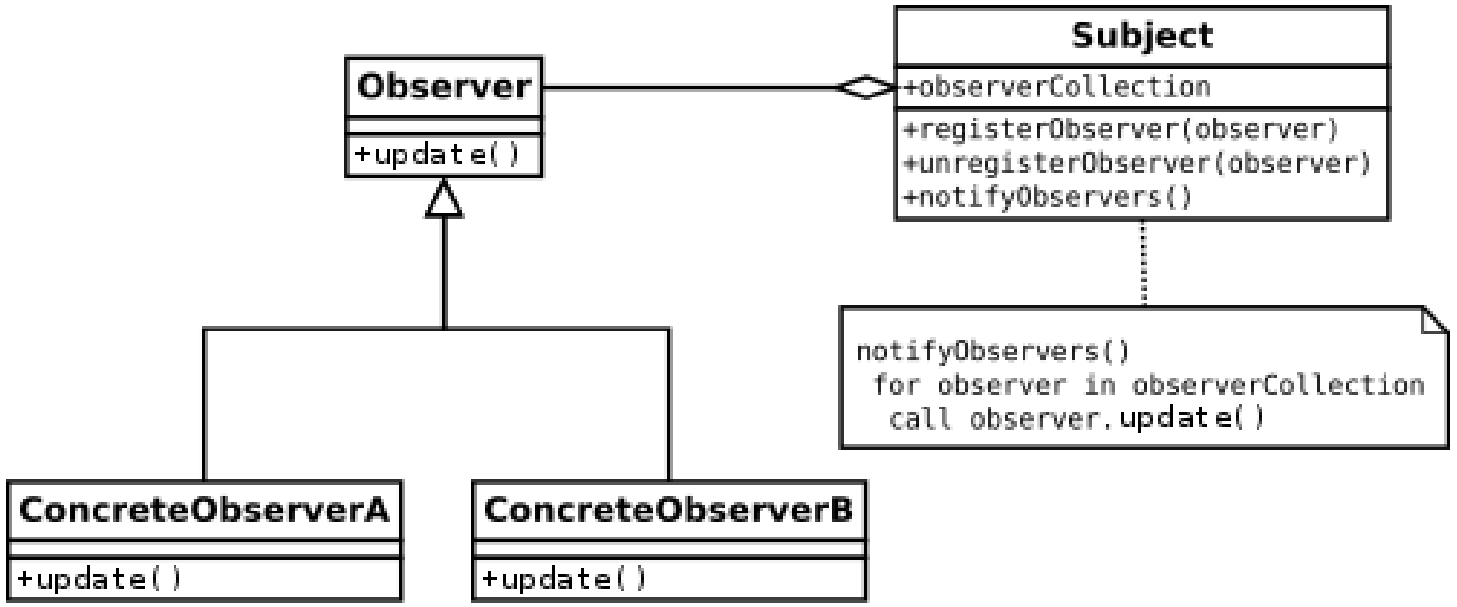
- Méthodes pour améliorer la communication entre les fonctionnalités
- Défini le flux de travail et les responsabilités de chaque entités
- Quelques exemples:
  - Chain of responsability
  - Interpreter
  - Observer
  - State

Pattern State



- Problématique : Une entité change de comportement en fonction de son état (déjà expliqué slide 32)
- Solution: On garde son état en mémoire dans le contexte. En fonction de cet état, ses fonctionnalités changent
- Exemple: Une machine à laver en attente de remplissage, en lavage, en rinçage, etc..

Pattern Observer



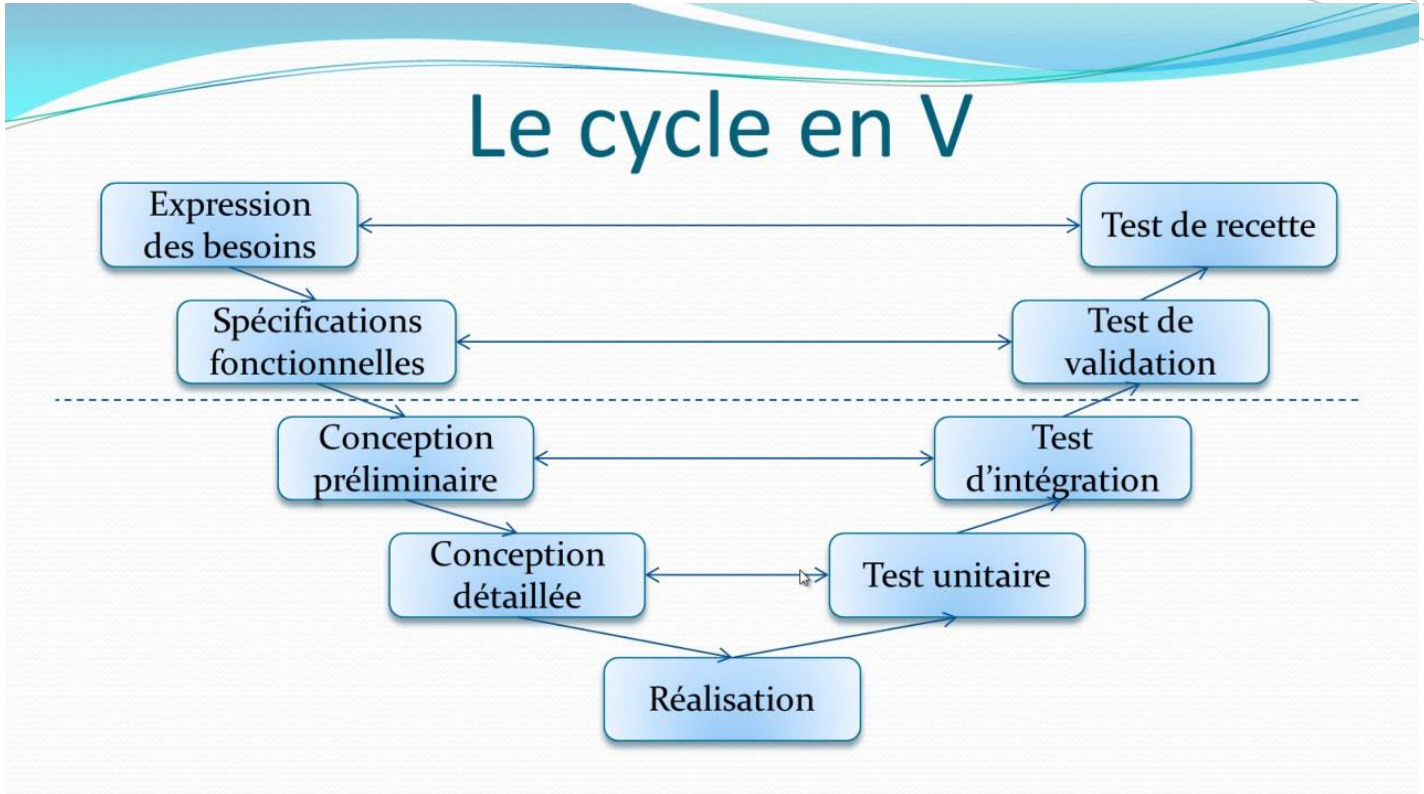
- Problématique : traitements asynchrones, en attente de la réponse
- Solution: au lieu de demander à chaque fois si la demande est traitée, être prévenu dès qu'elle est finie
- Exemple : Newsletters, demande de permis moto à la préfecture…

Comment on utilise l'UML à travers des projets?

Les projets informatiques

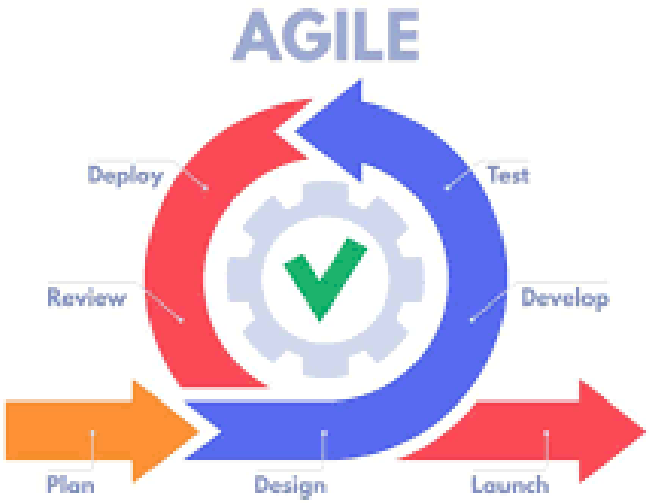
- Comment naissent ils?
- Les clients qui ont un besoin
- Concevoir ou faire évoluer un jeu/ un design/ une application
- Quelles sont les méthodes pour prendre ces besoins en compte?

Le cycle en V



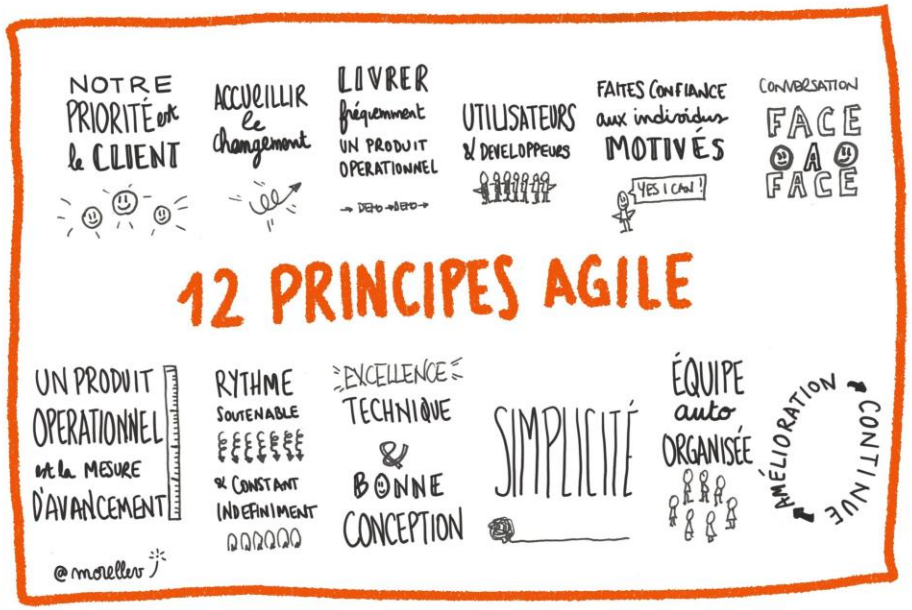
- Méthodologie de prise en compte d'un besoin dans un projet informatique
- Une phase de conception (descente du V) et une phase de réalisation (la remontée)
- Problématique : longue phase avant la démonstration au client
  - Peut être les besoins ont été mal compris
  - Ou ils ont changés entre temps…
  - Donc grosse perte de temps

L’agilité



- Pour limiter ce risque de perte de temps, les méthodes agiles ont été créées sur une base d’itérations à intervalle régulier
- Premières méthodes inventées dans les années 40 et utilisées pour le première fois à grande échelle dans le programme Mercury en 1950

Le manifest agile

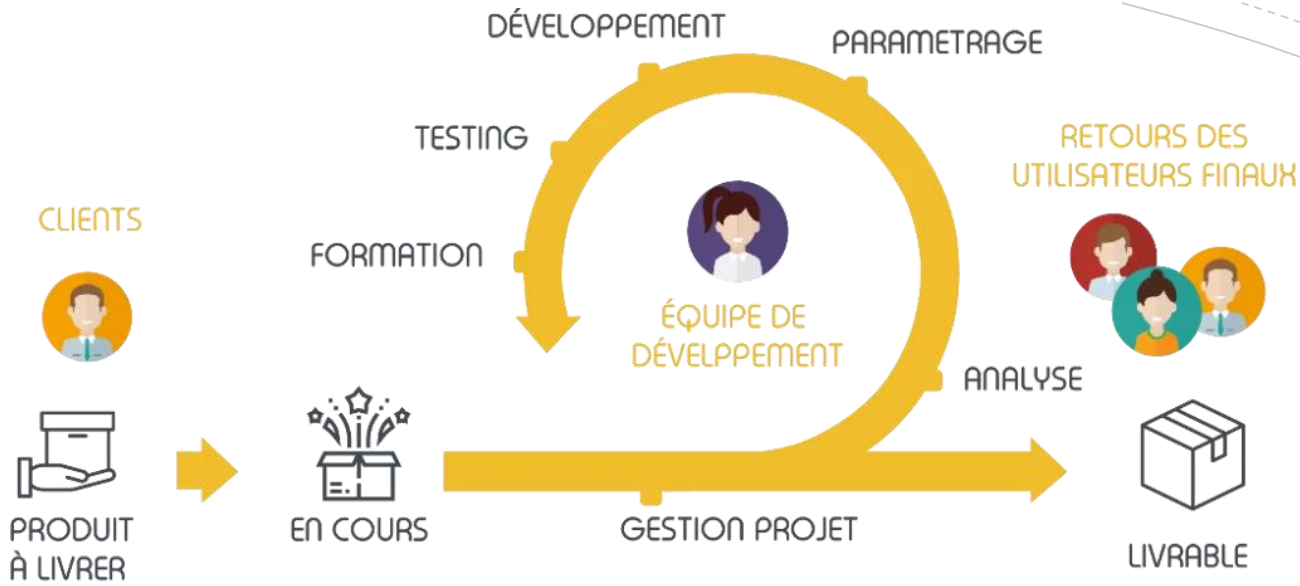


- Créé en 2001 par un ensemble d'ingénieur qui ont regroupé leur méthodologies de travail
- Basé sur 12 concepts autour de la satisfaction du client en le faisant participer activement au projet, pour avoir des retours au plus vite
- Principales méthodologies du manifest agile:
  - Extrem Programming
  - Scrum
  - Kanban
  - Safe

Ok, mais comment on pratique l'agilité dans les faits ?

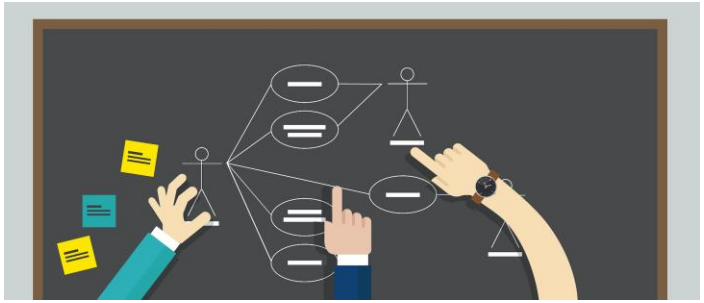
- L'agilité est basé sur des itérations de temps (des sprints)
- Le but est de pouvoir livrer une application fonctionnelle à chaque fin de sprint
- Une démonstration est faite alors au client pour avoir ses retours
- Cela permet créer un produit final petit à petit sans s'éloigner du besoin de base
- Avantages de l'agilité:
  - Adaptabilité / Réactivité
  - Focus sur les fonctionnalités essentielles et leur qualités
  - Amélioration continue
  - Réduction des risques

Et l'UML dans tout ça?



- L'UML fait partie intégrante des méthodologies agiles
- Toute la phase de conception est basée dessus
- Sur le schéma ci-dessus, L'UML se trouve dans :
  - Dans l'analyse du besoin, on peut trouver des diagrammes de haut niveaux comme le cas d'utilisation
  - Dans le paramétrage avec des diagrammes composants qui peuvent servir aussi pour la livraison de l'application
  - Dans la phase de développement, où les diagrammes sont plus dans le détail (classe, séquence…)
  - Plus globalement dans la communication avec le client où les utilisateurs à travers des diagrammes d'activités
  - Etc…

Bonnes pratiques dans la modélisation



Ne pas hésiter à vérifier le modèle par le code: cela prouve si les idées sont bonnes

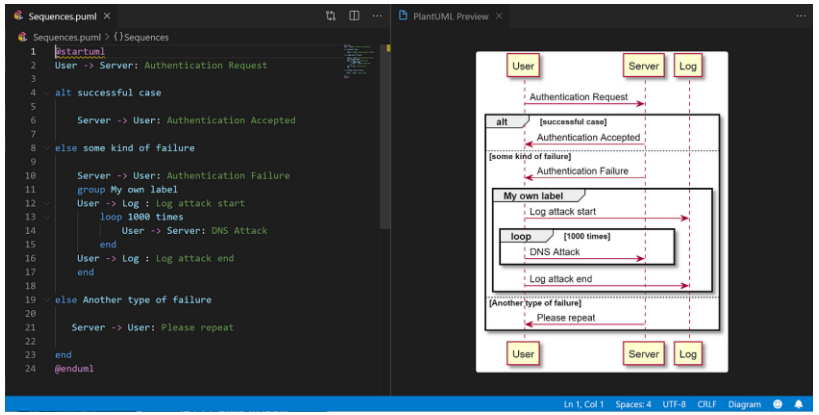
K.I.S.S

KEEP.IT.SIMPLE, STUPID!

Ne modélisez pas seul!
Cela doit être collaboratif, « Seul on va plus vite, ensemble on va plus loin »

Privilégiez des outils simples: des feutres et un tableau aident à être plus créatifs qu'un logiciel lourd

De la même manière, ne vous contentez pas d'un seul diagramme, passez sur un autre diagramme si vous êtes bloqués



Restez simple dans la conception: plus le modèle est simple et clair, plus il est facile à valider

Pas la peine de tout détailler !

Merci pour votre écoute

Vous avez des questions?

Pour me contacter

d.dupont@rubika-edu.com

david.dupont@imt-nord-europe.fr

Instant Exercice

Exercice

Modéliser un système de commande de repas qui permet aux clients de se faire livrer son repas à domicile par des livreurs indépendants et fait par des cuisiniers indépendants

Comment fonctionne le système?

Le système est décrit à travers les fonctionnalités suivantes de haut niveau:

- Inscription d'un utilisateur en tant que client, cuisinier ou livreur
- Pour s'inscrire, les utilisateurs doivent fournir un nom, prénom, adresse, numéro de téléphone
- Un cuisinier doit faire la liste des plats qu'il peut préparer
- Un cuisinier doit pouvoir indiquer pour chaque plat s'il est disponible ou non.
- Un cuisinier doit dire s'il est actuellement ouvert ou fermé
- Un repas a un nom, une liste d'ingrédients et un prix
- Un client peut commander à tous les cuisiniers au statut ouvert
- Un livreur peut accepter une livraison et reçoit de l'argent une fois fait
- les paiements sont gérés par Paypal -> une entité extérieure
- les clients donnent un avis sur le travail du cuisinier et du livreur
- Un administrateur peut supprimer des utilisateurs, des plats ou des commentaires inappropriés

Par groupe de 2, réaliser un diagramme structural et un ou plusieurs diagrammes comportementaux permettant de modéliser votre application « Rubi'Food »