

# Maven

- ❖ Basée sur un système de plugins
  - Un plugin ↔ un script Java
    - i.e. une classe avec une méthode particulière
    - Paramétrable via un morceau de XML
  - Une exécution de Maven ↔ suite d'exécution de plugins

| Phase                  | Tâche(s)                |
|------------------------|-------------------------|
| process-resources      | resources:resources     |
| compile                | compiler:compile        |
| process-test-resources | resources:testResources |
| test-compile           | compiler:testCompile    |
| test                   | surefire:test           |
| package                | jar:jar                 |
| install                | install:install         |
| deploy                 | deploy:deploy           |

- ❖ pom.xml ← config. du projet
- ❖ src/ ← sources
  - main/ ← à distribuer
    - java/ ← code Java
    - resources/ ← fichiers à distribuer (config appli, images, etc)
    - webapp/ ← ressources web (pour les war: html, jsp, js, images)
    - javacc/ ← grammaire pour générer les parsers
  - test/ ← uniquement pour les tests
    - java/, resources/, javacc/, etc
- ❖ target/ ← tout ce qui est généré, il est supprimé par clean  
Il ne faut **pas** le versionner (utiliser .gitignore)

# Git

## Commandes de base :

- ❖ Création
  - init, clone
- ❖ Fichiers
  - add, remove, mv, status
- ❖ Versions
  - commit, checkout
- ❖ Branches
  - branch, merge, rebase
- ❖ Synchronisation de dépôts
  - pull, fetch, push

# User Story

Les caractéristiques d'une bonne User Story : critères **INVEST**

- **Indépendante** : assure l'indépendance d'une User Story vis-à-vis des autres ;
- **Négociable** : une User Story doit être un support de discussion en vue d'une amélioration du besoin initial ;
- **Valorisable / Verticale** : la réalisation d'une User Story doit en soi rendre un service à l'utilisateur ;
- **Estimable** : une User Story doit être bien définie pour être facilement chiffrable ;
- **Suffisamment petite** : une User Story doit être réalisable sur un sprint (< 1 mois) ;
- **Testable** : une User Story doit être accompagnée de ces critères d'acceptabilité pour faciliter sa validation.

# Cas d'utilisation

Technique pour capturer les exigences fonctionnelles d'un système

- Déterminer ses limites
- Déterminer ce qu'il devra faire, quels services il rendra
  - Mais pas comment il devra le faire
  - Point de vue de l'utilisateur

Pour cela, il faut :

- Déterminer les **acteurs** (humain ou machine) qui interagissent avec le système
  - Rôles
- Déterminer les **grandes catégories d'utilisation**
  - Cas d'utilisation
- Décrire textuellement des interactions
  - Scénarios

Un **acteur** est décrit précisément en quelques lignes

Catégories d'acteurs :

- Acteurs principaux (fonctions principales du système)
- Acteurs secondaires (administration / maintenance)
- Matériel externe
- Autres systèmes

Un cas d'utilisation

- Définit un ensemble de scénarios d'exécution impliquant le même acteur (déclencheur) avec le même objectif utilisateur

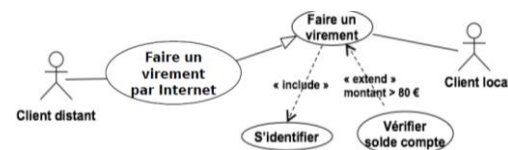
- Recense les informations échangées et les étapes l'utilisation du système + les points d'extension et cas d'erreur

**Scénario** est :

Séquence particulière d'étape dans la réalisation d'un CU.

Séquence particulière de messages dans le CU pendant une interaction particulière

- « chemin » dans le cas d'utilisation
- Peut-être vu comme un test
- Se termine par un succès ou un échec
- Décrit comment on arrive à ce terme
- Décomposés en séquences d'étapes
- Avec des objectifs de plus bas niveau, pouvant donner lieu à des sous-cas d'utilisation



« **include** »

- La réalisation d'un CU nécessite la réalisation d'un autre, sans condition, à un point d'extension (le seul important)
- Syntaxe textuelle : scénario inclus souligné

« **extend** »

- CU1 « extend » CU2 : le comportement de CU1 peut être complété par le comportement de CU2 (option avec condition et point d'extension)
- Conseil : ne pas utiliser, ou seulement si on ne peut toucher à CU1

« **generalize** »

- Héritage. (conseil : ne pas utiliser)

**La différence entre une « user-story » (au sens Scrum ou XP) et un cas d'utilisation (au sens d'Alistair Cockburn)**

Une user-story est très concise, ne détaille pas les étapes individuelles. Un cas d'utilisation est aussi plus général (par exemple un cas d'utilisation avec extensions correspond en général à plusieurs user-stories).

## A quoi servent les CU ?

-Clarifier les processus métier

- Bien comprendre le domaine, l'organisation pour laquelle on va concevoir et fabriquer le SI
- Complète la modélisation du domaine

-Fixer les limites du système

- Bien comprendre ce qui relève du système à concevoir et à construire
- ... et ce qui n'en relève pas





-Orienter la discussion

- Entre les concepteurs, le client, les futurs utilisateurs

Découvrir / fixer les besoins fonctionnels

- Fixer des exigences (contrat), mais pas toutes les exigences
- Importance des conditions d'échec pour ne rien laisser dans l'ombre
- Le plus important pour toute conception : décrire ce que le système permet de faire
- Remarque
  - Les cas d'utilisation seront réalisés avec des interactions d'objets : base de l'analyse et de la conception proprement dites

## Différents systèmes « emboîtés » à considérer :

1. Entreprise – organisation (pour fixer le contexte)
  - a. Intervenants :  
 actionnaires,  
 fournisseurs, administration, clients
  - b. Acteurs principaux : clients, fournisseurs
2. Système logiciel (le plus souvent)
  - a. Intervenants :  
 utilisateurs, société, administration, autres programmes
  - b. Acteurs principaux : utilisateurs, autres programmes
3. Sous-partie logicielle (si besoin)  


## Plusieurs niveaux d'objectif

- Objectif stratégique
  - Fonction du SI dans organisation
  - On se rapproche des processus métier

- Objectif utilisateur
  - Fonction du SI pour l'utilisateur
- Objectif sous-fonction
  - Fonction interne au système, utile pour l'informaticien

### 1. Entreprise / Organisation :

Possibilité de considérer l'entreprise comme

- Boîte noire
  - Vue uniquement de l'extérieur
- Boîte blanche
  - Fonctionnement interne explicite

Exemple :

- Fonctionnement de l'organisation Université au sein de l'Éducation Nationale
- Fonctionnement interne de l'organisation Université

### 2. système à construire :

Possibilité de considérer le système comme

- Boîte noire
  - Pour définir ses interactions avec l'extérieur (acteurs)
  - De loin le plus important pour l'expression des besoins
- Boîte blanche
  - Pour révéler le fonctionnement des composants

Exemple :

- Système de gestion des emplois du temps de l'Université
- Vu du point de vue de ses interactions avec les utilisateurs et les autres systèmes de l'Université
- Vu du point de vue interne

### 3. sous-système :

On décrit une sous-partie du système à construire

- Fonctionnement d'une des parties du système

Exemple :

- Sous-système de description des caractéristiques des salles dans le système de gestion des emplois du temps

## Exemple de Cas d'Utilisation :

Client qui veut retirer de l'argent

**CU :** Retirer de l'argent

**Portée :** système DAB

**Niveau :** objectif utilisateur

**Acteur principal :** Client

**Intervenants et intérêts :** Banque, Client

**Préconditions :** compte approvisionné

**Garanties minimales :** rien ne se passe

**Garanties en cas de succès :** de l'argent est retiré, le compte est débité de la même somme

**Scénario nominal :**

1. Le Client introduit sa carte dans le lecteur.
2. Le DAB déchiffre l'identifiant de la banque, le numéro de compte et le code secret de la carte, valide de la banque et le numéro de compte auprès du système principal.
3. Le client saisit son code secret. Le DAB valide par rapport au code secret chiffré lu sur la carte.
4. Le client sélectionne retrait, et un montant multiple de 10 € (min 20 €)
5. Le DAB soumet au principal système de la banque le compte client et le montant demandé, et reçoit en retour une confirmation et le nouveau solde du compte
6. Le DAB délivre la carte, l'argent et un reçu montrant le nouveau solde
7. Le DAB consigne la transaction

**Extensions :**

**\*a.** Panne générale.

**\*a1.** Le DAB annule la transaction, signale l'annulation, et rend la carte.

**2a.** Carte volée.

**2a1.** Le DAB confisque la carte volée

**4a.** Plus de billets de 10 €

**4a1.** Le DAB arrondit la somme demandée à un multiple de 20 €.

**4a2.** Le Client valide la nouvelle somme demandée.

**5a.** Solde insuffisant.

**5a1.** Le DAB signale que la somme demandée est trop élevée et rend la carte.

# Patterns

## Les patrons sont :

- Des solutions éprouvées à des problèmes récurrents
- Spécifiques au domaine d'utilisation
- Rien d'exceptionnel pour les experts d'un domaine
- Une forme littéraire pour documenter des pratiques
- Un vocabulaire partagé pour discuter de problèmes
- Un moyen efficace de réutiliser et partager de l'expérience

## Les patrons ne sont pas :

- Limités au domaine informatique
- Des idées nouvelles
- Des solutions qui n'ont fonctionné qu'une seule fois
- Des principes abstraits ou des heuristiques
- Une panacée

## Principles Are Not Patterns

- Principes généraux très utiles, mais qui ne s'appliquent pas à un problème concret
- Exemples
  - "Patterns" GRASP
  - Keep it Simple, Stupid (KISS)
  - Don't Repeat Yourself (DRY) / Duplication is Evil (DIE)
  - You aren't gonna use it

## SOLID

- Single responsibility
  - Une classe, une fonction ou une méthode doit avoir une et une seule responsabilité
- Open/closed
  - Les classes doivent être ouvertes à l'extension, mais fermées à la modification
- Liskov substitution
  - Une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme
- Interface segregation
  - Préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale
- Dependency inversion
  - Dépendre d'une abstraction (interface) et non d'une implémentation

## General Responsibility Assignment

### Software Patterns (GRASP):

Ensemble de principes (plutôt que patterns) généraux d'affectation de responsabilités pour aider à la conception orientée objet

### 9 Patterns GRASP :

- |                              |   |              |
|------------------------------|---|--------------|
| 1. Expert en information     | } | Principes    |
| 2. Créateur                  |   |              |
| 3. Faible couplage           |   |              |
| 4. Forte cohésion            |   |              |
| 5. Fabrication pure          | } | Outils       |
| 6. Indirection               |   |              |
| 7. Protection des variations | } | Applications |
| 8. Polymorphisme             |   |              |
| 9. Contrôleur                |   |              |

### 1. Expert en informatique :

Problème :

- Quel est le principe général d'affectation des responsabilités aux objets ?

Solution :

- Affecter la responsabilité à l'expert en information
  - La classe qui possède les informations nécessaires pour s'acquitter de la responsabilité

### 2. Créateur :

Problème :

- Qui doit avoir la responsabilité de créer une nouvelle instance d'une classe donnée ?

Solution :

- Affecter à la classe B la responsabilité de créer une instance de la classe A si une ou plusieurs de ces conditions est vraie :
  - B contient ou agrège des objets A
  - B enregistre des objets A
  - B utilise étroitement des objets A
  - B a les données d'initialisation qui seront transmises aux objets A lors de leur création
    - B est un Expert en ce qui concerne la création de A

```
class A {  
    int ia;  
    A (int ia) { this.ia = ia; }  
}
```

```
class B {  
    A a;  
    B (A a) { this.a = a; }  
    B (int ia) {  
        this.a = new A (ia);  
    }  
}
```

```
void main () {  
    // not good  
    A a = new A (10);  
    B b = new B (a);  
    // better create A  
    // inside B  
    B b = new B (10);  
}
```

## 3. Faible couplage

Problème :

- Comment minimiser les dépendances ?
- Comment réduire l'impact des changements ?
- Comment améliorer la réutilisabilité ?

Solution :

- Affecter les responsabilités des classes de sorte que le couplage reste faible
- Appliquer ce principe lors de l'évaluation des solutions possibles

Problèmes du couplage fort :

- Un changement dans une classe force à changer toutes ou la plupart des classes liées
- Les classes prises isolément sont difficiles à comprendre
- Réutilisation difficile : l'emploi d'une classe nécessite celui des classes dont elle dépend

## 4. Forte cohésion

Problème : maintenir une complexité gérable

- Comment s'assurer que les objets restent
  - Compréhensibles ?
  - Faciles à gérer ?
- Comment s'assurer au passage que les objets contribuent au faible couplage ?

Solution :

- Attribuer les responsabilités de telle sorte que la cohésion reste forte
- Appliquer ce principe pour évaluer les solutions possibles

Cohésion (ou cohésion fonctionnelle)

Définition :

- Mesure informelle de l'étroitesse des liens et de la spécialisation des responsabilités d'un élément (d'une classe)
  - Relations fonctionnelles entre les différentes opérations effectuées par un élément
  - Volume de travail réalisé par un élément
- Une classe qui est fortement cohésive
  - A des responsabilités étroitement liées les unes aux autres
  - N'effectue pas un travail gigantesque

Un test :

- Décrire une classe avec une seule phrase

Problèmes des classes à faible cohésion :

- Elles effectuent
  - Trop de tâches
  - Des tâches sans lien entre elles
- Elles sont
  - Difficiles à comprendre
  - Difficiles à réutiliser
  - Difficiles à maintenir
  - Fragiles, constamment affectées par le changement

## 5. Fabrication pure

Problème :

- Que faire
  - Pour respecter le Faible couplage et la Forte cohésion
  - Quand aucun concept du monde réel (objet du domaine) n'offre de solution satisfaisante

Solution :

- Affecter un ensemble fortement cohésif à une classe artificielle ou de commodité, qui ne représente pas un concept du domaine
  - Entité fabriquée de toutes pièces

Exemple :

- Problème
  - Les instances de Prêt doivent être enregistrées dans une BD
- Solution initiale (d'après Expert)
  - Prêt a cette responsabilité
  - Cela nécessite

| Prêt                |
|---------------------|
| livresPrêtés:Livre  |
| idAbonné            |
| Serveur:SGBD        |
| editerBulletin()    |
| insertionBD(Object) |
| majBD(Object)       |
| ...                 |

- Un grand nombre d'opérations de BD

- Prêt devient donc non cohésif

- De lier Prêt à une BD

- Le couplage augmente pour Prêt

| Prêt               |
|--------------------|
| livresPrêtés:Livre |
| idAbonné           |
| editerBulletin()   |
| ...                |

- Constat
  - L'enregistrement d'objets dans une BD est une tâche générique utilisable par de nombreux objets

| GestionArchivage  |
|-------------------|
| Serveur:SGBD      |
| insertion(Object) |
| maj(Object)       |
| ...               |

L'enregistrement d'objets dans une BD est une tâche générique utilisable par de nombreux objets

- Pas de réutilisation, beaucoup de duplication

- Solution avec Fabrication pure
  - Créer une classe artificielle GestionArchivage
- Avantages
  - Gains pour Prêt
    - Forte cohésion et Faible couplage
  - Conception de GestionArchivage « propre »
    - Relativement cohésif, générique et réutilisable

## 6. Indirection

Problème :

- Où affecter une responsabilité pour éviter le couplage entre deux entités (ou plus)
  - De façon à diminuer le couplage (objets dans deux couches différentes)
  - De façon à favoriser la réutilisabilité (utilisation d'une API externe)

Solution :

- Créer un objet qui sert d'intermédiaire entre d'autres composants ou services
  - L'intermédiaire crée une indirection entre les composants
  - L'intermédiaire évite de les coupler directement

Utilité :

- Réaliser des adaptateurs, façades, etc. (pattern Protection des variations) qui s'interfaçent avec des systèmes extérieurs
  - Exemples : proxys, DAO, ORB
- Réaliser des inversions de dépendances entre packages

Mise en œuvre :

- Utilisation d'objets du domaine
- Création d'objets
  - Classes : cf. Fabrication pure
  - Interfaces : cf. Fabrication pure + Polymorphisme

## 7. Protection des variations

Problème :

- Comment concevoir des objets, systèmes, sous-systèmes pour que les variations ou l'instabilité de certains éléments n'aient pas d'impact indésirable sur d'autres éléments ?

Solution :

- Identifier les points de variation ou d'instabilité prévisibles
- Affecter les responsabilités pour créer une interface (au sens large) stable autour d'eux (indirection)

Mise en œuvre :

- Cf. patterns précédents (Polymorphisme, Fabrication pure, Indirection)

Exemples de mécanismes de PV :

- Encapsulation des données, brokers, machines virtuelles...

## Ne pas parler aux inconnus

Cas particulier de Protection des variations

- Protection contre les variations liées aux évolutions de structure des objets

Problème :

- Si un client utilise un service ou obtient de l'information d'un objet indirect (via un objet direct (familier du comment le faire sans couplage ?

Solution :

- Éviter de connaître la structure d'autres objets indirectement
- Affecter la responsabilité de collaborer avec un objet indirect à un objet que le client connaît directement pour que le client n'ait pas besoin de connaître ce dernier.

Cas général à éviter :

a.getB().getC().getD().methodeDeD() :

- Si l'une des méthodes de la chaîne disparaît, A devient inutilisable

Préconisation :

- Depuis une méthode, n'envoyer des messages qu'aux objets suivants
  - L'objet this (self)
  - Un paramètre de la méthode courante
  - Un attribut de this
  - Un élément d'une collection qui est un attribut de this
  - Un objet créé à l'intérieur de la méthode

Implication :

- Ajout d'opérations dans les objets directs pour servir d'opérations intermédiaires

## 8. Polymorphisme

Problème :

- Comment gérer des alternatives dépendantes des types ?
- Comment créer des composants logiciels « enfichables » ?

Solution :

- Affecter les responsabilités aux types (classes) pour lesquels le comportement varie
- Utiliser des opérations polymorphes

Polymorphisme :

- Donner le même nom à des services dans différents objets
- Lier le « client » à un supertype commun

Principe :

- Tirer avantage de l'approche OO en sous classant les opérations dans des types dérivés de l'Expert en information
  - L'opération nécessite à la fois des informations et un comportement particulier

Mise en œuvre :

- Utiliser des classes abstraites
  - Pour définir les autres comportements communs
  - S'il n'y a pas de contre-indication (héritage multiple)
- Utiliser des interfaces
  - Pour spécifier les opérations polymorphes
- Utiliser les deux (CA implémentant des interfaces)

- Fournit un point d'évolution pour d'éventuels cas particuliers futurs

## 9. Contrôleur

Problème :

- Quel est le premier objet au-delà de l'IHM qui reçoit et coordonne (contrôle) une opération système (événement majeur entrant dans le système) ?

Solution :

- Affecter cette responsabilité à une classe qui représente
  - Soit le système global, un sous-système majeur ou un équipement sur lequel le logiciel s'exécute
    - Contrôleur Façade ou variantes
  - Soit un scénario de cas d'utilisation dans lequel l'événement système se produit
    - Contrôleur de CU ou contrôleur de session

Principes :

- Un contrôleur est un objet qui ne fait rien reçoit les événements système délègue aux objets dont la responsabilité est de les traiter
- Il se limite aux tâches de contrôle et de coordination
  - Vérification de la séquence des événements système
  - Appel des méthodes ad hoc des autres objets
- Il n'est donc pas modélisé en tant qu'objet du domaine → Fabrication pure

Règle d'or :

- Les opérations système des CU sont les messages initiaux qui parviennent au contrôleur dans les diagrammes d'interaction entre objets du domaine

Mise en œuvre :

- Au cours de la détermination du comportement du système (besoins, CU, DSS), les opérations système sont déterminées et attribuées à une classe générale Système
- À l'analyse/conception, des classes contrôleur sont mises en place pour prendre en charge ces opérations

## Contrôleur Façade

Représente tout le système :

- exemples: ProductController, RetailInformationSystem, Switch, Router, NetworkInterfaceCard, SwitchFabric, etc.

À utiliser quand :

- Il y a peu d'événements système
- Il n'est pas possible de rediriger les événements systèmes à un contrôleur alternatif

## Contrôleur de cas d'utilisation (contrôleur délégué)

Un contrôleur différent pour chaque cas d'utilisation

- Commun à tous les événements d'un cas d'utilisation
- Permet de connaître et d'analyser la séquence d'événements système et l'état de chaque scénario

À utiliser quand :

- Les autres choix amènent à un fort couplage ou à une cohésion faible (contrôleur trop chargé – bloated)
- Il y a de nombreux événements système qui appartiennent à plusieurs processus
- Permet de répartir la gestion entre des classes distinctes et faciles à gérer

Élément artificiel :

- Ce n'est pas un objet du domaine

# Patterns de création

- Singleton (Singleton)

- Fabrique (Factory Method)

- Fabrique abstraite (Abstract Factory)

- Monteur (Builder)

- Prototype (Prototype)

## Singleton

Objectif :

- S'assurer d'avoir une instance unique d'une classe
  - Point d'accès unique et global pour les autres objets
  - Exemple : Factory

Fonctionnement :

- Le constructeur de la classe est privé (seules les méthodes de la classe peuvent y accéder)
- L'instance unique de la classe est stockée dans une variable statique privée
- Une méthode publique statique de la classe
  - Crée l'instance au premier appel
  - Retourne cette instance

## Fabrique (Factory Method)

Factory :

- Un objet qui fabrique des instances conformes à une interface ou une classe abstraite
- Par exemple, une Application veut manipuler des documents, qui répondent à une interface Document
  - Ou une Equipe veut gérer des Tactique...

## Abstract Factory

Objectif :

- Création de familles d'objets
- Généralisation du pattern Factory Method

Fonctionnement : « fabrication de fabriques » :

- Regroupe plusieurs Factories en une fabrique abstraite
- Le client ne connaît que l'interface de la fabrique abstraite
- Il invoque différentes méthodes qui sont déléguées à différentes fabriques concrètes

## Monteur (Builder)

Objectif :

- Instancier et réaliser la configuration initiale d'un objet en s'abstrayant de l'interface de l'objet
- Fournir une instance à un client

Remarques :

- S'applique en général à des objets complexes
- Différence avec le pattern [Abstract] Factory
  - Plutôt utilisé pour la configuration que pour la gestion du polymorphisme

## Prototype

Objectifs :

- Réutiliser un comportement sans recréer une instance
  - Économie de ressources

Fonctionnement

- Recopie d'une instance existante (méthode clone())
- Ajout de comportements spécifiques : « polymorphisme à pas cher »



# Patterns de structure

- Objet composite (Composite)
- Adaptateur (Adapter)
- Façade (Facade)
- Proxy (Proxy)
- Décorateur (Decorator)

## Objet composite (Composite)

Objectif :

- Représenter une structure arborescente d'objets
- Rendre générique les mécanismes de positionnement / déplacement dans un arbre
  - Exemple : DOM Node

Fonctionnement :

- Une classe abstraite (Composant) qui possède deux sous classes
  - Feuille
  - Composite : contient d'autres composants

## Adaptateur (Adapter)

Objectif :

- Résoudre un problème d'incompatibilité d'interfaces (API)
  - Un client attend un objet dans un format donné
  - Les données sont encapsulées dans un objet qui possède une autre interface

Fonctionnement :

- Insérer un niveau d'indirection qui réalise la conversion

Patterns liés :

- Indirection, Proxy

## Façade (Facade)

Objectif :

- Cacher une interface / implémentation complexe
  - Rendre une bibliothèque plus facile à utiliser, comprendre et tester
  - Rendre une bibliothèque plus lisible
  - Réduire les dépendances entre les clients de la bibliothèque

Fonctionnement :

- Fournir une interface simple regroupant toutes les fonctionnalités utiles aux clients

Patterns liés

- Indirection, Adaptateur

## Proxy (Proxy)

Objectif :

- Résoudre un problème d'accès à un objet
  - À travers un réseau
  - Qui consomme trop de ressources...

Fonctionnement :

- Créer une classe qui implémente la même interface
- La substituer à la classe recherchée auprès du client

Patterns liés :

- Indirection, État, Décorateur

## Décorateur (Decorator)

Objectif :

- Résister au changement
  - Principe général :

Les classes doivent être ouvertes à l'extension, mais fermées à la modification

- Permettre l'extension des fonctionnalités d'une application sans tout reconcevoir

Fonctionnement :

- Rajouter des comportements dans une classe qui possède la même interface que celle d'origine
- Appeler la classe d'origine depuis le décorateur
- Effectuer des traitements « autour » de cet appel

Pattern lié :

- Proxy

Pattern antagoniste :

- Polymorphisme

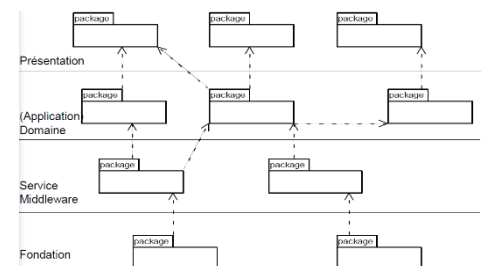
# Patterns de comportement

- Interpréteur (Interpréter)
- Memento (Memento)
- État (State)
- Stratégie (Strategy)
- Visiteur (Visitor)
- Commande (Command)
- Chaîne de responsabilité (Chain of responsibility)
- Observateur (Observer)
- Fonction de rappel (Callback)
- Promesse (Promise)

## Patterns architecturaux

- Architecture en couches
- Architecture multi tiers
- MV\*
- IoC
- Contexte
- Observer
- DAO, DTO

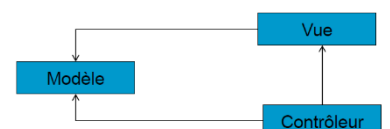
### Architecture en couches



### Modèle-Vue-Contrôleur

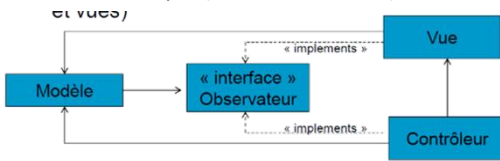
Version modèle passif :

- La vue se construit à partir du modèle
- Le contrôleur notifie le modèle des changements que l'utilisateur spécifie dans la vue
- Le contrôleur informe la vue que le modèle a changé et qu'elle doit se reconstruire



Version modèle actif :

- Quand le modèle peut changer indépendamment du contrôleur
- Le modèle informe les abonnés à l'observateur qu'il s'est modifié
- Ceux-ci prennent l'information en compte (contrôleur et vues)



Model View Adapter (MVA)

- Pas de communication directe entre modèle et vue
  - Un pattern Adapteur Médiateur ) prend en charge les communications
  - Le modèle est intentionnellement opaque à la vue

Model View Presenter (MVP)

- La vue est une interface statique (templates)
- La vue renvoie (route) les commandes au Presenter
- Le Presenter encapsule la logique de présentation et l'appel au modèle

Model View View Model (MVVM)

- Mélange des deux précédents : le composant View Model
  - Sert de médiateur pour convertir les données du modèle
  - Encapsule la logique de présentation
- Autre nom : Model View Binder (MVB)

## Agile

**Traditional project:** rationalize the process, document as much as possible (V cycle, "say what you do, do what you say")

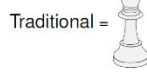
- Planning easy: all specifications are there, metrics from previous projects too, ...
- One developer leaving the team ) the next ones will read the docs
- Reproducible: just follow the procedures

**Agile:** keep in mind that working software is the main goal

- If specifications are broken, change them.
- Doing specification is hard. Discussing a prototype is easier → "Release early."

Release often" (Eric S. Raymond, The Cathedral and the Bazaar, 1997)

- Many traditional project management techniques turn out to have bigger overhead than benefit. Abandon them. ("eliminate waste", lean principle)
- Developers are important. Give them power.



Start → Projet → End

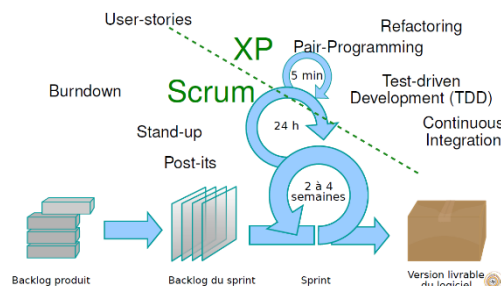


Start → Iterations → ...



Global trend, many complementary tools or variants:

- **eXtrem Programing:** focused on coding practices (code review, tests, ...)
- **Scrum:** divide a project into iterations, plan each iteration at once
- **Kanban:** ≈ lightweight variant of Scrum
- **Lean:** Eliminate waste (usually a global, company-wide approach)
- **DevOps:** Automate what can be (from development to deployment)



## Scrum

- Set of practices to organize a team (7 +/- 2 developers)
- Focused on human interactions (inside and outside the team)
- Short iterations (called "sprint") & delivery cycle: 1 sprint = 1 week to 1 month
- The most popular in companies today(?)

Scrum Roles:

- **Product Owner (PO):** discussion with the client, discuss product specifications. Must understand the client needs (not necessarily a computer-scientist)
- **Scrum Master:** facilitator (6= boss), helps the team follow Scrum (or not), protect the team against distraction
- **Developers (Team member):** Write code, take technical decisions, develop the product.

Journey of a feature in Scrum :

1. Client wants "something"
2. Discussion with the PO (Product Owner)
3. Agreement on a (set of) user stories ("as a ... I want to ... in order to ...")
4. User stories not started = product backlog (PO fills-in the backlog, developers empty it)
5. Start of iteration: decide on the sprint backlog
6. User stories split into technical tasks
7. Each task goes from TODO → ongoing → Done.
8. Demo at the end of iteration
9. Release (or not)
10. Retrospective and celebrate!

Scrum Board :

- Can be physical (post-its) or virtual (GitLab or GitHub's issues, Trello, ...)
- Virtual: more traceability, multiple geographical sites, ...
- Physical also has benefits:
  - 1 post-it = 1 unit of information. Text doesn't fit on post-it → split into smaller items.
  - (Use a felt tip pen, not a thin pen)
  - Always visible on the wall → you can't claim you forgot!
  - No remote access → if your boss wants to see the board, she must come in the room.
  - Flexible (take a pen and draw a line Vs ask the admin of the project to create a column)
  - Satisfaction of moving post-its to DONE :-)

Steps of a sprint:

- Sprint planning: discuss/agree on the sprint backlog
- Development, continuous testing & integration. Daily scrum meetings.
- Demo
- Retrospective: discuss and improve
- goto 1 (forever?)

# Test

Sprint planning (1 meeting, a few hours) :

- Prepared by the PO: product backlog = set of user-stories, sorted by priority
- Evaluation (story points) of first stories
- By experience, 1 sprint = X points ) stop when  $\sum(\text{story points}) = X$
- Example discussion:
  - Team: we evaluate this story to 40 points.
  - PO: that's too high!
  - Team: that's not your business ...
  - PO: OK, this is high priority but it's too long for now, I'm changing its priority.
- Or:
  - PO: can we reduce the scope of the story to make it fit in 20 points?
  - Team: yes, for example we can make a rough UI and finish the business logic for 20 points.
  - PO: OK, I'm splitting the story, we'll make a nice UI in the next sprint.

End of sprint planning :

- Team split user-stories (user-spec) into technical tasks
- Some teams evaluate technical tasks in hours of work. Some just split the story points of the story.
- Make a nice scrum board!
- Initialize the burndown

The Burndown Chart :

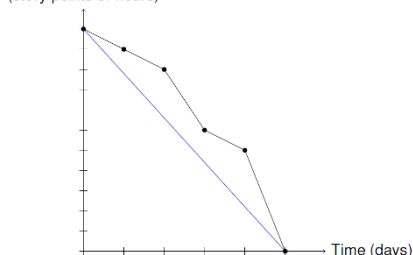
- 1 point every day
- Helps medium-term planning (are we late?)
- → helps having constant pressure all along the sprint (6= "Cool, we're on time" followed by "sleepless night before release")
- Count remaining work, not work done.

Why:

- Task estimated to 7
- 5 points done
- 4 point remaining (sorry, we underestimated the task!)

- Hopefully a decreasing function

Remaining work  
(story points or hours)

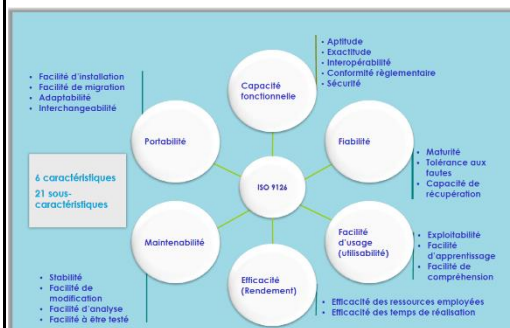


Pourquoi il faut tester?

- Parce que tout être humain peut faire une erreur (méprise) qui produit un défaut (bogue) dans le code, dans un logiciel ou un système, ou dans un document
- Et si un défaut du code est exécuté, il peut générer une défaillance (ou pas)

1. Pour limiter le coût des bogues
  - Baisse de réputation
  - Économique (Perte de contrat, Application de pénalités)
  - Humains, environnementaux, etc.
2. Pour assurer la qualité d'une application
  - Pour atteindre les objectifs de normes/lois (ex: DO = 178B pour l'aviation)
  - Pour atteindre un bon rapport qualité/ prix
  - Pour maintenir la confiance du client

La qualité d'un logiciel :



- Portabilité
- Maintenabilité
- Capacité fonctionnelle
- Fiabilité
- Facilité d'usage (utilisabilité)
- Efficacité (Rendement)

Définition(s) du test :

Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus

- IEEE (Standard Glossary of Software Engineering Terminology)

Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts

- G. Myers (The Art of Software testing)

Testing can reveal the presence of errors but never their absence

- Edgar W. Dijkstra. Notes on structured programming. Academic Press, 1972.

Objectifs du test :

- Trouver des défauts
- Augmenter le niveau de confiance en la qualité d'un logiciel
- Fournir aux décideurs go/no go un état le plus précis possible de la qualité du logiciel
- Prévenir des défauts

Les 7 principes généraux :

1. Les tests montrent la présence de défauts
  - Jamais leur absence
2. Les tests exhaustifs sont impossibles
  - Exemple Sopra : test d'une IHM contenant 20 écrans, 4 menus, 3 options, 10 champs, 2 types de données, 100 valeurs possibles
  - $20 \times 4 \times 3 \times 10 \times 2 \times 100 = 480\,000$  cas de tests
  - Un expert (10s par test!) mettrait 180 jours
3. Il faut tester le plus tôt possible
4. Les défauts sont généralement regroupés
  - Loi de Pareto: 80% des défauts sont concentrés dans 20% des modules
5. Paradoxe du pesticide
  - Les mêmes tests ne décèlent plus de défauts
6. Les tests dépendent du contexte
  - Les tests de sécurité dépendent de l'application testée
7. L'illusion de l'absence d'erreur
  - Trouver et corriger des défauts ne sert à rien si le système conçu est inutilisable (opérativité, conformité, attractivité, performance)

Tester ≠ Déboguer

- Tester et déboguer, ce n'est pas la même activité et les responsables sont différents:
  - Les testeurs testent
  - Les développeurs déboguent
- Déboguer: activité de développement permettant d'analyser, trouver et supprimer les causes de la défaillances



Les différents niveaux de test

- **Tests unitaires** ou de composant : tester chaque élément de manière isolé
  - Élément: méthode, classe, composant, etc.
  - Acteurs: les développeurs
  - Le code est accessible
  - Aucun rapport d'incident
  - Vocabulaire:  
bouchon/simulateur/pilote, test statique/dynamique
- **Test d'intégration** : tester le bon comportement/l'interaction des éléments suite à une composition d'éléments
  - Acteurs: les développeurs ou équipe dédiée
  - Le code est accessible ou pas
  - Vocabulaire: big bang, bottom up, top down, ihm , simulateur
- **Test système ou de conformité** : assurer que l'application présente les fonctionnalités attendues
  - Acteurs: l'équipe dédiée
  - Le code n'est pas accessible
  - Vocabulaire: qualification, performance
- **Test de validation ou d'usine ou d'opération** : valider l'adéquation avec les besoins du client
  - Acteurs: l'équipe dédiée, le client
  - Le code n'est pas accessible
  - Vocabulaire: Alpha tests, Beta tests

Les différents types de test

- **Test des fonctionnels** : pertinence, exactitude, interopérabilité, sécurité, conformité
- **Test des caractéristiques non fonctionnelles** : fiabilité, facilité d'utilisation, rendement/efficacité, maintenabilité, portabilité
- **Test de la structure** : architecture logicielle
- **Test lié au changement**: test de confirmation, test de régression
- **Test de maintenance** : modification, migration, suppression

Les différents techniques de test

- (Comment sélectionner les tests?)
- **Test fonctionnel ou boîte noire** : la sélection se fait en se basant sur la spécification
    - On teste ce que doit faire l'application

- **Test structurel ou boîte blanche/transparente** : la sélection se fait en se basant sur le code
  - On teste la manière dont l'application le fait
- **Test probabiliste** : la sélection se base sur le domaine d'entrée en fonction d'arguments probabilistes
  - Ces différentes techniques sont utilisées de manières complémentaires car elles ne remontent pas les mêmes bogues.
  - Ils existent même des approches qui les combinent pour compenser les défauts des uns par les qualités de l'autre.

Le test fonctionnel :

- Avantages:
- Ne dépend pas du code: langage, technique, style
  - Peut donc être défini avant le développement
  - Basé sur l'interface et les fonctionnalités, le nombre de scénarios est de taille raisonnable
  - Assure l'adéquation du code avec la spécification
- Défauts:
- Ignore les défauts de programmation
  - La concrétisation des scénarios n'est pas toujours évidente

Utilisé pour tous les niveaux de test

Le test structurel :

- Avantages:
- Basé sur le code: le nombre de scénarios est plus important mais plus précis
  - Sensible aux défauts de programmation
- Défauts
- Le code doit être accessible
  - Problème d'Oracle: le résultat est il celui réellement attendu?
  - Ignore les fonctionnalités absentes

Utilisé pour le test unitaire

Le test aléatoire

- Avantages:
- Ne dépend pas du code: langage, technique, style
  - Basé sur l'interface et les fonctionnalités, le nombre de scénarios est de taille raisonnable
  - Permet du test en masse

- Défauts:
- Ignore les défauts de programmation
  - La concrétisation des scénarios n'est pas toujours évidente

Utilisé pour tous les niveaux de test

- Un test s'est :
1. Choisir un scénario à exécuter (cas de test)
  2. Estimer le résultat attendu (oracle)
  3. Déterminer les données du test et le résultat concret attendu
    - État initial
    - Données du programme
    - Suite d'actions à exécuter
  4. Exécuter le test
  5. Comparer le résultat attendu et le résultat obtenu

Comment choisir les bons scénarios?

- (critères objectifs)
- Idéalement, il faudrait qu'ils soient exhaustifs mais cela est généralement impossible.
  - Ils doivent donc permettre d' exécuter un maximum de comportements différents de l'application
    - Couverture des cas dits nominaux : les cas de fonctionnements les plus fréquents
    - Couverture des cas limites ou délicats
    - Entrées invalides
    - Montée en charge
    - Sécurité

- (critères qualitatifs)
- Ils doivent aussi permettre de contribuer à assurer la qualité de l'application: en trouvant rapidement le plus de bogues possibles
  - Ils doivent aussi permettre de démontrer la qualité de l'application à un tiers: en étant le plus représentatif possible

- (critères plus subjectifs)
- L'expérience des anomalies: certains tests sont effectués en fonction d'anomalies récurrentes, de cas récurrents
  - La connaissance du développeur:
    - ses défauts/qualités,
    - ses affinités par rapport aux fonctionnalités,
    - ses classiques
  - L'analyse d'impact

## Critère d'arrêt: test fonctionnel

(Quand arrêter de tester ou comment déterminer un bon jeu de test?)

Un bon critère doit être: efficace, objectif, simple, automatisable

Test fonctionnel:

- couverture de scénarios d'utilisation
- couverture des partitions des domaines d'entrée
- couverture des cas limites

### Scénario minimal

- Pré conditions : L'utilisateur doit être authentifié en tant que client ou commercial (Cas d'utilisation « S'authentifier » package « Authentification »)
- Démarrage : L'utilisateur a demandé la page « Consultation catalogue »
- Post conditions : Aucun
- Ergonomie:
  - L'affichage des produits d'une catégorie devra se faire par groupe de 15 produits. Toutefois, afin d'éviter à l'utilisateur d'avoir à demander trop de pages, il devra être possible de choisir des pages avec 30, 45 ou 60 produits.
- Performance attendue :
  - La recherche des produits, après sélection de la catégorie, doit se faire de façon à afficher la page des produits en moins de 10 secondes.
- Problèmes non résolus :
  - Nous avons fait la description basée sur l'information que les produits appartiennent à une catégorie. Est ce qu'il existe des sous catégories ? Si tel est le cas, la description devra être revue
  - Etc.

### Scénario nominal

- L'utilisateur accède à la page Consultation catalogue
  - Le système affiche une page contenant la liste les catégories de produits.
- L'utilisateur sélectionne une des catégories.
  - Le système recherche les produits qui appartiennent à cette catégorie.
  - Le système affiche une description et une photo pour chaque produit trouvé.
- L'utilisateur peut sélectionner un produit parmi ceux affichés
  - Le système affiche les informations détaillées du produit choisi.
- L'utilisateur peut ensuite quitter cette description détaillée
  - Le système retourne à l'affichage des produits de la catégorie

## Scénarios alternatifs

- L'utilisateur décide de quitter la consultation de la catégorie de produits choisie.
- L'utilisateur décide de quitter la consultation du catalogue.
- L'utilisateur décide de quitter la consultation de la catégorie de produits choisie.
- L'utilisateur décide de quitter la consultation du catalogue
- L'utilisateur décide de quitter la consultation de la catégorie de produits choisie
- L'utilisateur décide de quitter la consultation du catalogue.

## Critère d'arrêt: test structurel

Un bon critère doit être: efficace, objectif, simple, automatisable

Test structurel:

- couverture des instructions, des enchaînements
- couverture des conditions
- couverture des dépendances de données

## Automatisation des tests

- C'est très important car l'activité est difficile et très chère
- Difficultés:
  - Trouver des défauts n'est pas naturel (surtout chez les développeurs)
  - La qualité dépend de la pertinence des jeux de test
- Une bonne automatisation est synonyme:
  - D'amélioration de la qualité
  - De maintenance simplifiée
- Il faut donc:
  - beaucoup de scénarios efficaces
  - une plateforme dédiée (serveur, base de données)
- Quelques outils: Sélénium, TestComplete, Quality Test Plan, Robot framework, etc.
- Bien choisir son outil: exécution, dépouillement, mise à jour

## Pourquoi encore des tests manuels?

- Application très paramétrable
- Application qui doit tourner:
  - Sur toutes les plateformes (Windows, Mac, Unix, Linux)
  - Sur tous les navigateurs (de IE6 à IE11, Firefox, Chrome, etc.)
  - Avec une base de données (sous divers Oracle, divers SQLServer, dans plusieurs langues)
- Un robot n'est pas un humain: ergonomie, design, temps de réponse, indiscipline, etc.

## Le test(ing) en entreprise

- Le test est effectué traditionnellement par deux types de profils: les développeurs et les testeurs
- Le test effectué généralement en entreprise est un test dynamique: il s'agit d'exécuter du code pour s'assurer d'un fonctionnement correct
- Il appartient aux méthodes dites V & V:
  - Validation: l'application répond-elle aux besoins du client?
  - Vérification: l'application fonctionne-t-elle correctement?

## Les différentes méthodes V&V

- Test dynamique
- Test statique: revue de code ou de spécifications, etc.
- Vérification symbolique: exécution symbolique, etc.
- Vérification formelle: preuve, model checking, etc.

## Le développeur, un bâtisseur

- C'est un expert qui trouve les bons algorithmes et la meilleure solution aux problèmes
- Il interprète les éventuelles ambiguïtés de la spécification
- Il peut oublier les détails visuels
- Il n'a pas une vue d'ensemble
- Il travaille sans limite de temps
- Il n'aime pas les erreurs qui montrent qu'il est faillible
- Il voit le testeur comme un messenger porteur de mauvaise nouvelle et voit donc son travail comme une activité destructive

## Le testeur, un destructeur ?

- Il trouve les cas où le logiciel peut être défaillant
- Il a une vue d'ensemble
- Il vérifie que le logiciel répond bien aux exigences
- Il anticipe les problèmes possibles
- Il donne de l'importance au détail surtout s'il s'agit d'une exigence
- Il est limité par le temps et n'a pas le droit aux débordements
- Plus il trouve de bogues, plus il est bon (vraiment?)
- Il voit son travail comme une activité constructive