Principes d'architecture logicielle

Université de Nantes
M1 MIAGE 2024-2025

Bertrand Florat
Architecte Solutions

bertrand@florat.net

Avis / suggestions

Objectifs du cours

- Présenter quelques **principes d’un bon code**
- Présenter quelques principes d’une **bonne conception logicielle**
- Mettre en pratique ces principes
- **Échanger** en se basant sur son expérience personnelle et professionnelle

Agenda (à prioriser)

- **Les pré-requis à tout bon développement**
- **Les grands principes de l'architecture logicielle**
- Les familles de tests
- Introduction au TDD et BDD
- Le découpage en couches et l'architecture hexagonale
- Les applications cloud-native
- Les méthodes de développement agiles
- L'injection de dépendance
- Aperçu des design patterns principaux du GoF

Thèmes au choix

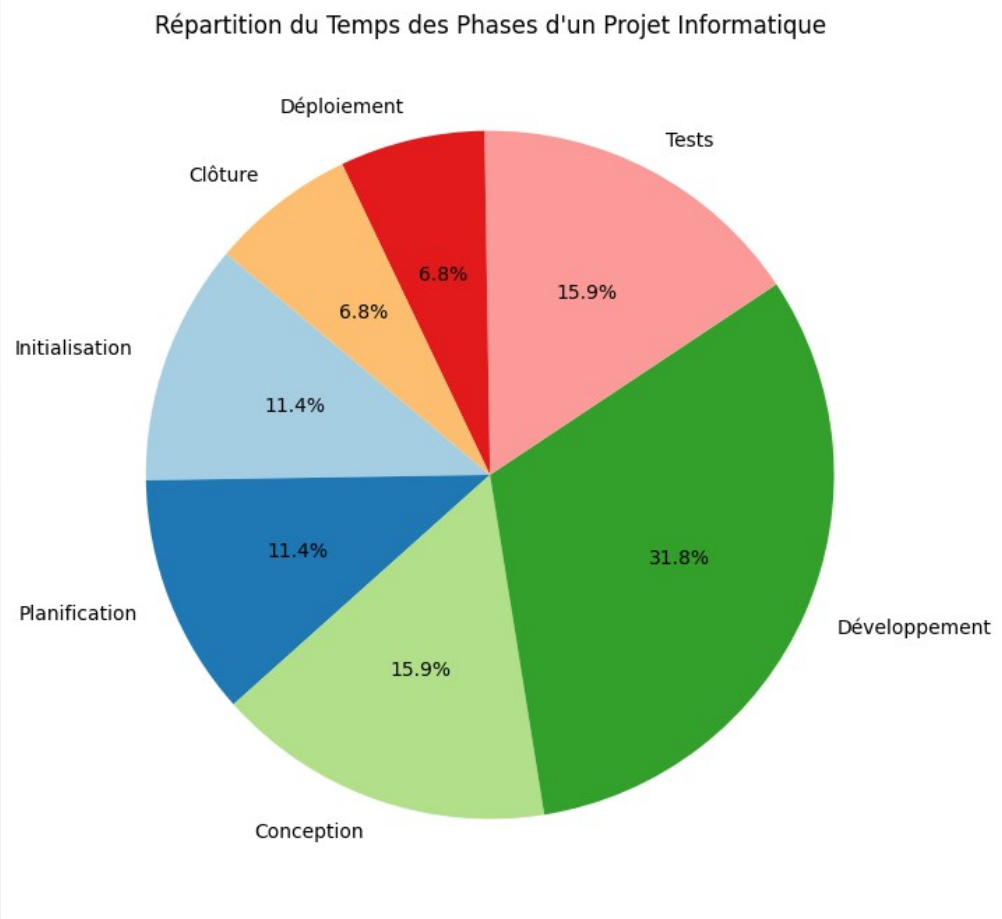
https://forms.gle/C8KAecTfSxG3oP4N9

Les pré-requis à tout bon développement

Pré-requis no 1 : faire les bonnes choses

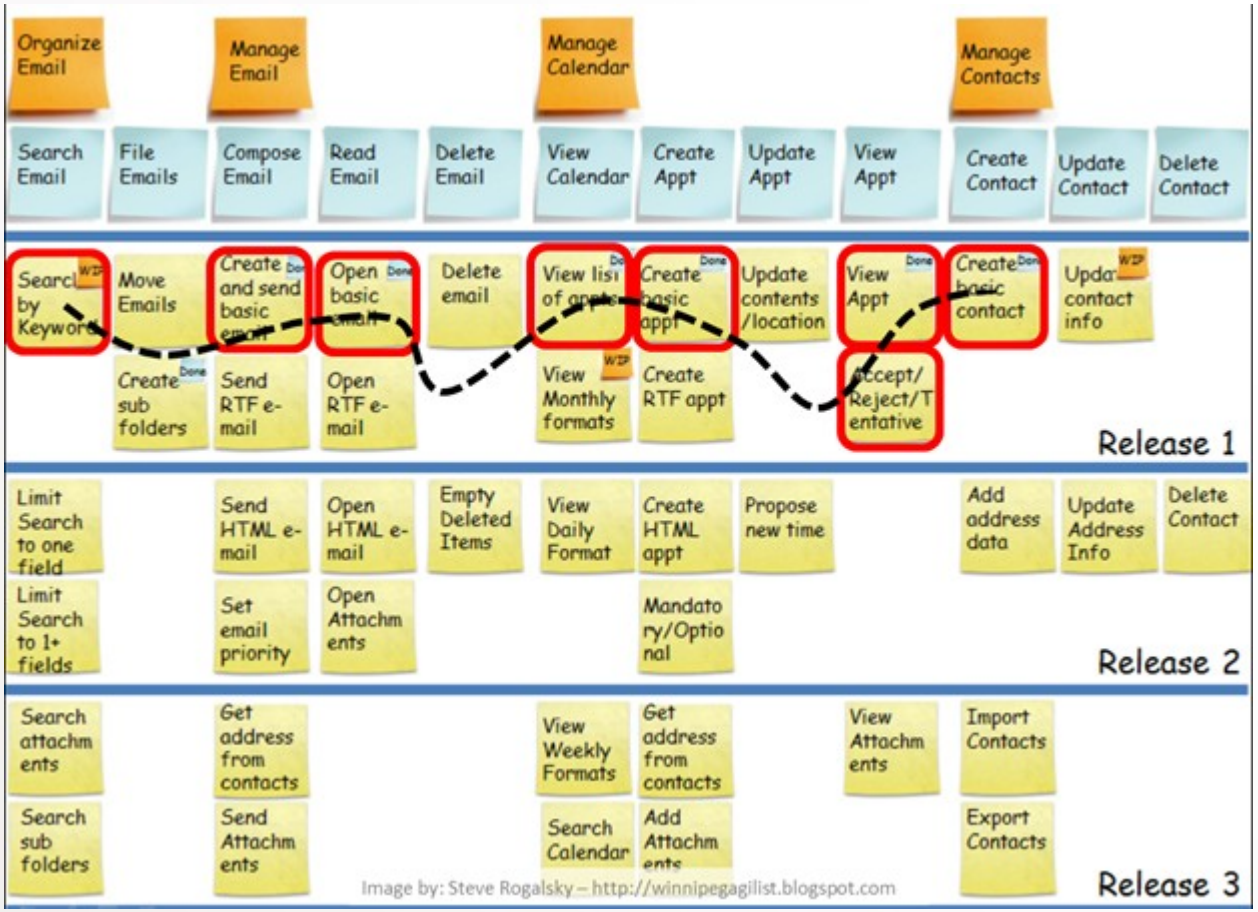
- **Faire les bonnes choses** : Bien plus important que de faire les choses bien.
- Comment ? via les **pratiques agiles**, principalement :
  - Le **story mapping** et autres méthodes de conception (impact mapping, feature mapping)
  - Les **personas**
  - Les **spécifications exécutables** et vivantes
  - Les **démos** de fin de sprint
  - Le **DDD** (ubiquitous language)
  - Toute **collaboration** avec les métiers (ateliers...)

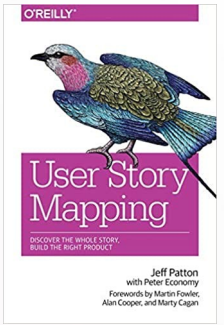
Le code : ce n’est qu’une partie du cycle de vie !



Source : Le Chat (Mistral-AI 7B)

Exemple de story mapping





Les personas

Takako Kimura

28 ans, professeur d'Histoire à Kyoto

Apprendre toujours plus et partager mes connaissances.

Aisance numérique :

Expertise domaine :

Fréquence d'usage : Plusieurs fois pendant le séjour

Takako est une jeune femme qui enseigne l'Histoire dans un collège de Kyoto ; son travail la passionne.

Pour la première fois, elle et son compagnon Haru vont aller visiter Paris. Ils ont réservé leurs billets d'avion et un hôtel sur Internet. Takako a choisi la période du 5 au 19 juillet car elle sera en congés et elle avait très envie de voir le défilé du 14 juillet.

Elle a acheté un guide de voyage pour préparer des visites. Pendant le séjour, elle préférerait utiliser une application sur son iPhone car le guide imprimé est un peu lourd à transporter.

Buts clés

- Visiter tous les lieux historiques de Paris.
- Faire certaines visites avec un guide qui parle le Japonais.
- Trouver des informations détaillées sur l'Histoire des lieux, des personnages etc.
- Conserver des traces de ses visites pour les présenter à ses élèves.

Personnalité

- Curieuse, patiente et passionnée.
- Accorde de l'importance à l'esthétique des choses.
- Achète rarement des applications sur son iPhone.

Figure 1 - Exemple de fiche persona

Source : http://www.weloveusers.com/formation/apprendre/personas.html

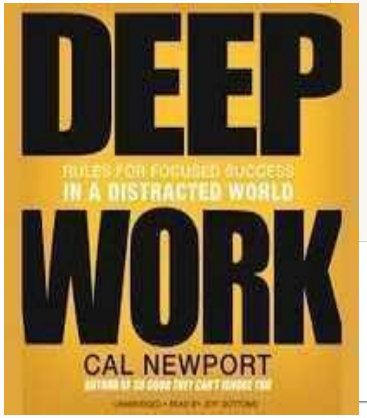
Le BDD : une spécification exécutable

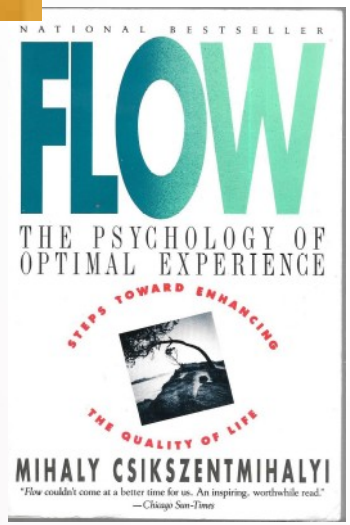
```
@DAF:rpm
Feature: Calculer une pension RPM de droit direct d'un marin
 En tant qu'agent instructeur ou chef de groupe
 Je souhaite calculer une pension RPM de droit direct d'un marin

Scenario: Je veux effectuer un calcul de pension RPM de droit direct pour le marin PELLEN Francois
Given un marin dont l'ID est 10064756 et un demandeur dont l'ID est 10064756
When je lance le calcul d'une pension RPM avec les caracteristiques suivantes : <codeTypePension> <codeConcession> <codeExo> <categorie>
Then la pension possède une retenue ? : <possedeRetenues>

Examples:
|codeTypePension|codeConcession |codeExo|categorie  |dateJouissance |dateDemande|dateCompletude |tauxPension|possedeRetenues|
|-- Cas code EXO 3 : il doit y avoir des retenues dans ce cas
|2              |1              |2      |7          |07/10/2015     |07/10/2015 |07/10/2015     |10         |true           |
|-- Cas code EXO different de 2 : pas de cotisations sociales
|2              |1              |3      |7          |07/10/2015     |07/10/2015 |07/10/2015     |10         |false          |
```

Voir la chapitre consacré au BDD

Pré-requis no 2 : le focus

- **Un seul projet** à la fois
- Un **Scrum Master** qui protège
- 5 h max de dev / j
- Éliminer les voleurs de temps
- Auto-discipline :
  - pas de procrastination (limiter réseaux sociaux)
  - mono-tâche : technique **Pomodoro**
  - Chercher le « **flow** »
  - Éviter d'interrompre (utiliser messagerie instantanée)
  - Sommeil





Pré-requis no 3 : la collaboration entre devs

- Open Space de 5 à **10 personnes max**
  - communication osmotique
  - Two pizza team, DevOps

- Même style de code, formatage
- **Pair programming**
- **Peer review (MR)**,
  - De préférence outillé (Gitlab, Gerrit, ...)
- De préférence, même IDE

Pré-requis no 4 : disposer de bons outils

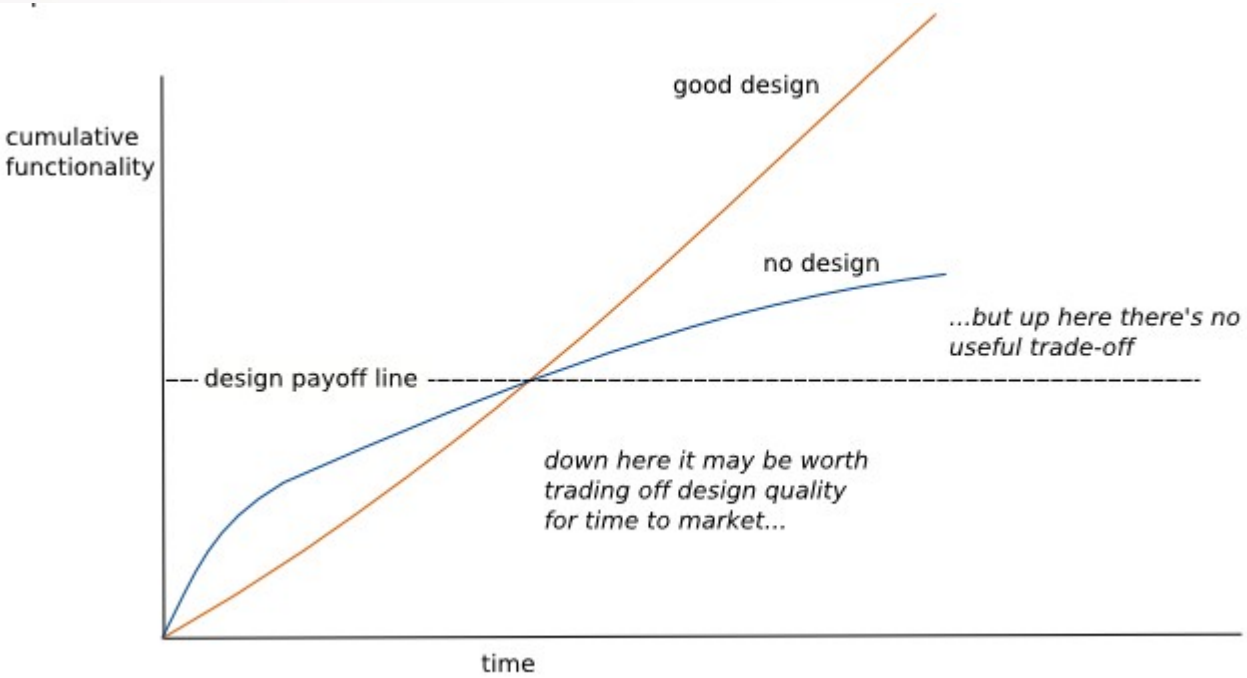
- **Poste** de développement **performant** :
  - CPU : moyen
  - Mémoire : élevée (idéalement 32 Gio)
  - Disque : important (SSD)
  - Au moins deux écrans 24+ pouces
  - De préférence un portable (réunions, présentations...)
- Bonne usine logicielle (**cobotique logicielle**)
- **Messagerie instantanée** (RocketChat, Slack...)

Pré-requis no 5 : être compétent

- **Veille techno**, tutos, ex : https://dzone.com/, developpez.com
- Demander des **formations** ou **MOOC**, ex : OpenClassrooms, CodecAcademy, Udemy, fun-mooc
- Maîtriser son outil de gestion de versions (**Git**...)
- S’améliorer toute sa carrière
- Lire des **livres**
- Etudier le code produit par l'**IA** (ChatGPT)
- **Lire du code** éprouvé (GitHub)
- **Comprendre** : OK pour StackOverflow mais comprendre
- Be fluent in **English**

Les grands principes de l’architecture logicielle

Prévoir le bon niveau de design en fonction de l'importance et de la durée de vie du projet



Source : https://medium.com/4th-coffee/on-devops-5-how-do-you-know-you-are-doing-it-right-architecture-fb6f653e4f6a
inspiré de Martin Fowler

- **Éviter l'overdesign** pour les **petits produits** / POC
- **Éviter** le code « à **la rache**'» pour les **produits importants/évolutifs**

YAGNI (You Ain’t Gonna Need It)

- En a-t-on besoin ?
  - si oui : en a-t-on *vraiment* besoin ?
    - si oui, tout de suite ?

  → le remède : le **just-in-time** (flux tiré en lean) : il est souvent urgent d’attendre ...

DRY : Don’t Repeat Yourself

(Réutilisation, factorisation)

En intégration aussi :

conf/**myproject**/**myproject**-deploy.yml **:** NON (`myproject`) en double

conf/**myproject**/deploy.yml : Oui

KISS (KEEP IT SIMPLE STUPID)

- Laisser les solutions **émerger**
- Ne pas s'**inventer des besoins** ou des contraintes
- Pas d'**optimisation prématurée**
- **Limiter les dépendances externes**
- Écrire le code **pour des humains, pas l'ordinateur**

« La simplicité est la sophistication suprême » – Léonard de Vinci

« Any fool can write code that a computer can understand. **Good programmers write code that humans can understand**. » - Martin Fowler

Less is more

- Derrière chaque ligne de code un bug peut se cacher
- Créer le système **le plus simple qui fonctionne**
- C'est aussi :
  - de la **documentation**
  - du **temps de calcul**
  - du temps de **build**
  - du code à **maintenir** si refactoring
  - de la **qualimétrie** à maintenir
  - failles de **sécurité**....

Lean (Toyota)

Le pire ***Muda*** (gaspillage) est la **surproduction** car la cause de la plupart des autres *muda* (attente, stock, ...)

Les lois « sociales »

- La **loi de Conway** : un système informatique reflète toujours l'organisation qui l'a créé
- **Loi de Knuth** : "Premature optimization is the root of all evil" (mais ne pas s'en servir comme prétexte…)
- **Le culte du cargo** et ses effets en entreprise
  - ne pas appliquer sans comprendre ;
  - **challenger les décisions du passé** (ADR)

Couplage faible / cohésion forte

- un composant doit être le moins dépendant possible des autres (**couplage faible**)
- mais avoir une **forte cohérence** (voir critères de P ressman)

Comment determiner les limites ?

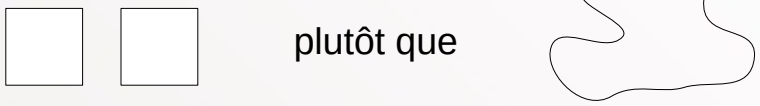
Voir l'approche DDD (**Bounded Contexts**)

KISS > DRY

- **Si** c'est **compliqué de mutualisé** ;
- **Si** c'est **simple de maintenir** du code dupliqué par le l'outillage :

ALORS : **privilégier la duplication**

KISS prioritaire sur DRY



Convention over configuration

(architecture dite « on rails »)

- **Limiter** autant que possible **les décisions à prendre**
- Toujours **respecter** les **standards** de l'entreprise et des règles de l'art
- Exemples : Maven, Ruby on rails, Grails, Hibernate
- Principe du ***least astonishment*** appliqué au code

et le plus important ...

toujours revenir au (vrai) besoin

C'est à dire la réponse aux (maximum) **cinq « pourquoi ?»** qui permettent de déterminer le vrai besoin

Christophe, un chef de projet :

- « J'ai besoin de deux nouveaux serveurs (à 6K€ pièce) en DMZ, c'est Sébastien de la prod qui m'a dit ça » → « Pourquoi ? » (no 1)

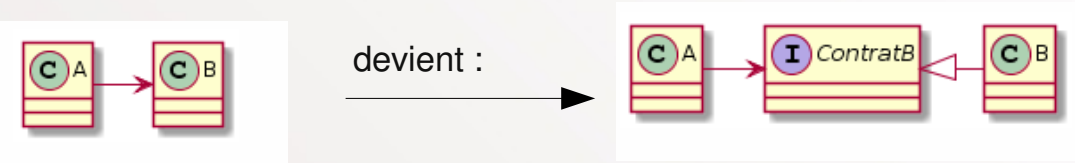
- « pour monter un cluster d'Apache supportant 1M d'appels par mois en HA pour notre nouveau site » → « Pourquoi faire ? » (no 2)

- « pour héberger des images » → « à quoi servent ces images » (no 3) ?

- « elles sont utilisées dans nos courriels » → « **on va faire autrement**, on va insérer les images en inline directement dans le courriel, plus besoin de serveurs Web dédiés à ce besoin. » → plusieurs 10K€, semaines de travail économisés, architecture plus simple et plus robuste, moins d'administration à prévoir.

Les principes SOLID pour l'OO (Robert Martin)

- **Single Responsability** : cohérence fonctionnelle forte, une seule raison de changer
- **Open-Close** : ouvert aux évolutions (être étendu), fermé aux modifications
- **Liskov substitution** principle : on peut remplacer une classe par une sous-classe
- **Interface segregation** principle : sur-mesure par client plutôt que général. On ne doit pas forcer un client à implémenter des méthodes qu'il n'utilise pas.
- **Dependency Inversion** Principle (DIP) : les clients ne connaissent que les interfaces, pas les implémentations



Les bonnes pratiques
La gestion des erreurs

- Bien différencier **erreurs techniques** et erreurs **fonctionnelles**
  - Technique : imprévisible, on ne peut rien faire (exceptions unchecked)
  - Fonctionnelle : fait parti du contrat
- Utiliser les **exceptions**, pas des code retours
- Utiliser les **exceptions standards**
- **Programmation défensive** aux frontières du composant : contrôles de durcissement
- **Contexte** dans les **messages**
- Non à l'anti-pattern **couche-culotte** (j'attrape et je garde)

Les marronniers (sujets récurrents)

Encodage des chaînes de caractères tout le long des chaînes de liaison

Solution : tout en Unicode / **UTF-8**

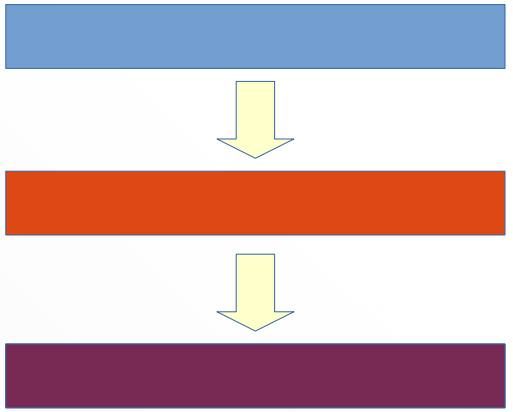
La gestion des fuseaux horaires tout le long des chaînes de liaison

Solution : **tout en UTC** sauf à l'affichage final et/ou **TZ explicite (voir ISO-8601)**

Le découpage en couches et l’architecture hexagonale

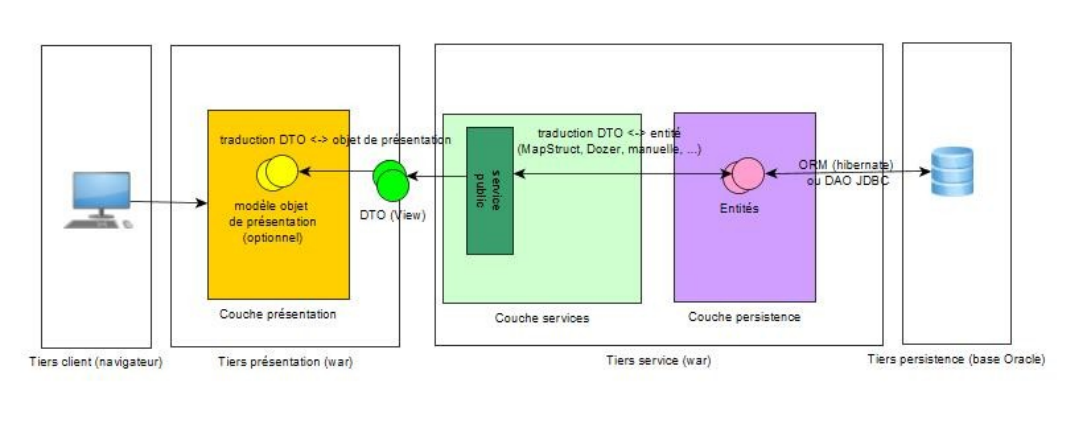
Les couches ?

- Séparation du code en ensembles fortement cohérents et faiblement couplés
- Une couche n ne dépend que de la couche n+1



Ne pas confondre couche (layer) et niveau (tiers)

- Changement de **tiers** si on change de machine ou de composant d'infrastructure (ex : Tomcat)
  - Exemple : architecture 3 tiers Web (Le navigateur / le middleware / la base de donnée)
- Une **couche** ne s'exécute que **sur un seul tiers**
- Un tiers peut contenir plusieurs couches



Pourquoi faire ?

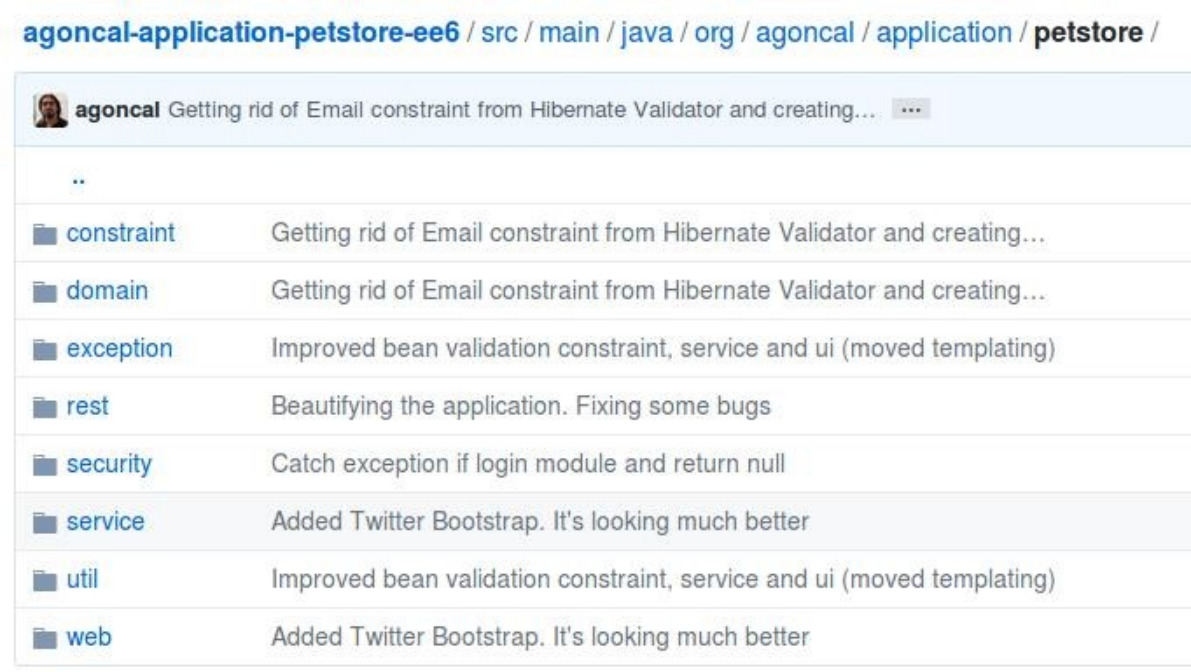
- Séparation des **préocupations**

  (SOLID)

  - Essayez de maintenir une page PHP avec du SQL...

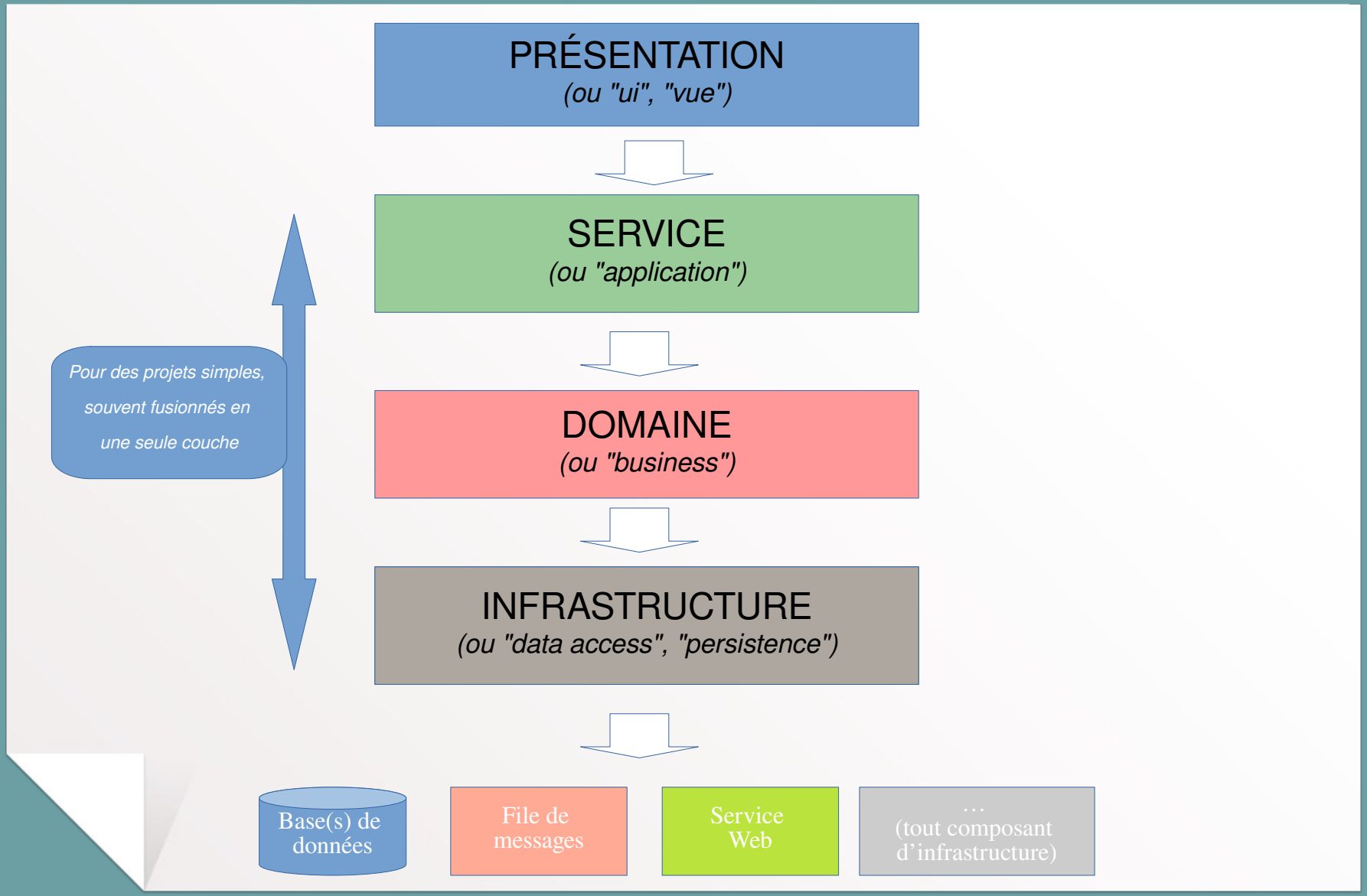
- Principe de la **boite noire**
  - une couche masque les couches derrière
    pour appeler un service, aucune connaissance de la base nécessaire
- **Testabilité**
- **Maintenabilité**
- **Répartition** du travail
- **Structure** les projets
- Imposé par l'**architecture n-tiers**

Comment ça se traduit concrètement ?

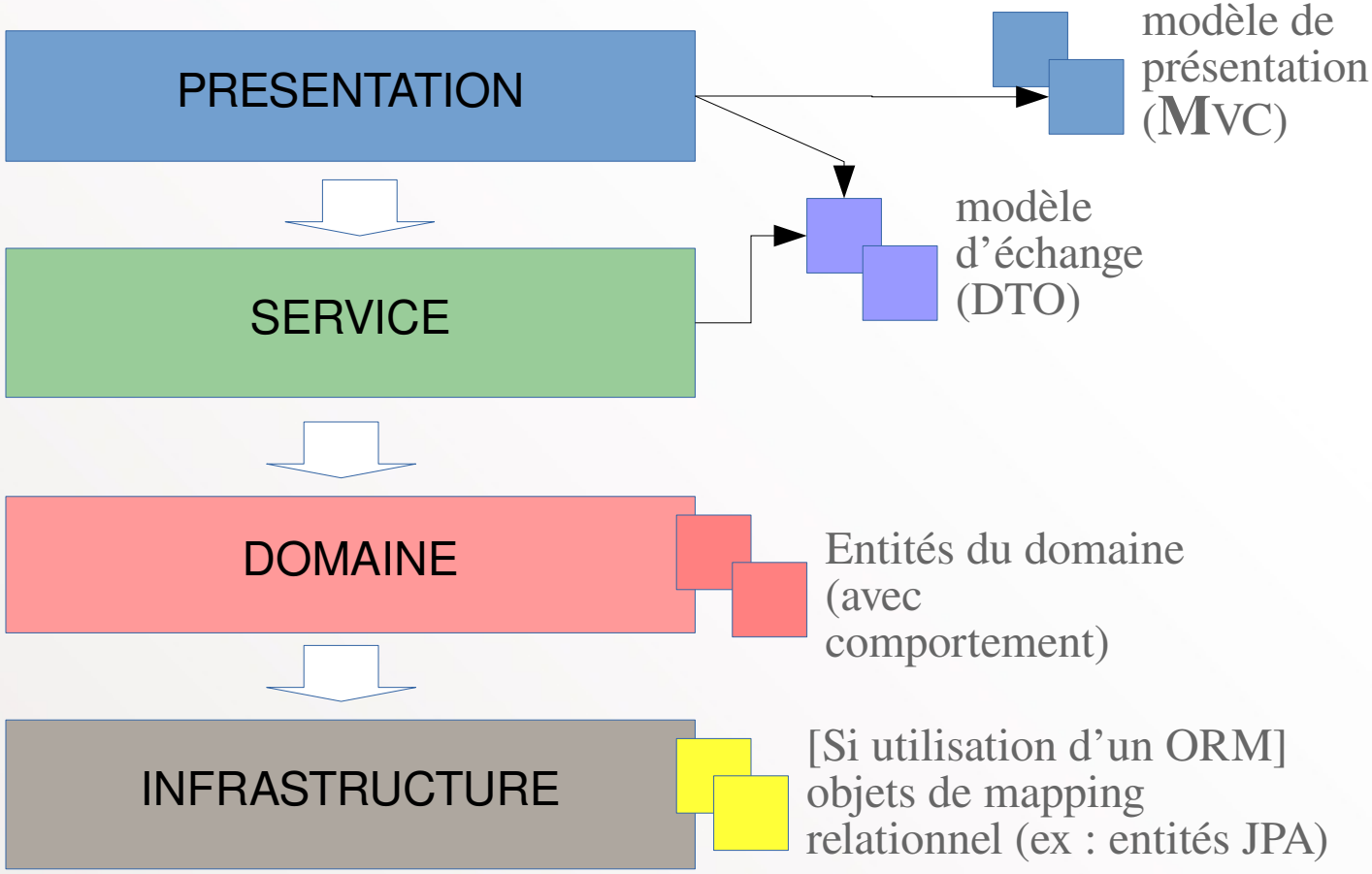


Un Java petstore (https://github.com/agoncal/agoncal-application-petstore-ee6/tree/master/src/main/java/org/agoncal/application/petstore)

Le découpage en couche le plus courant



Les modèles objet recommandés



Il y a un prix...

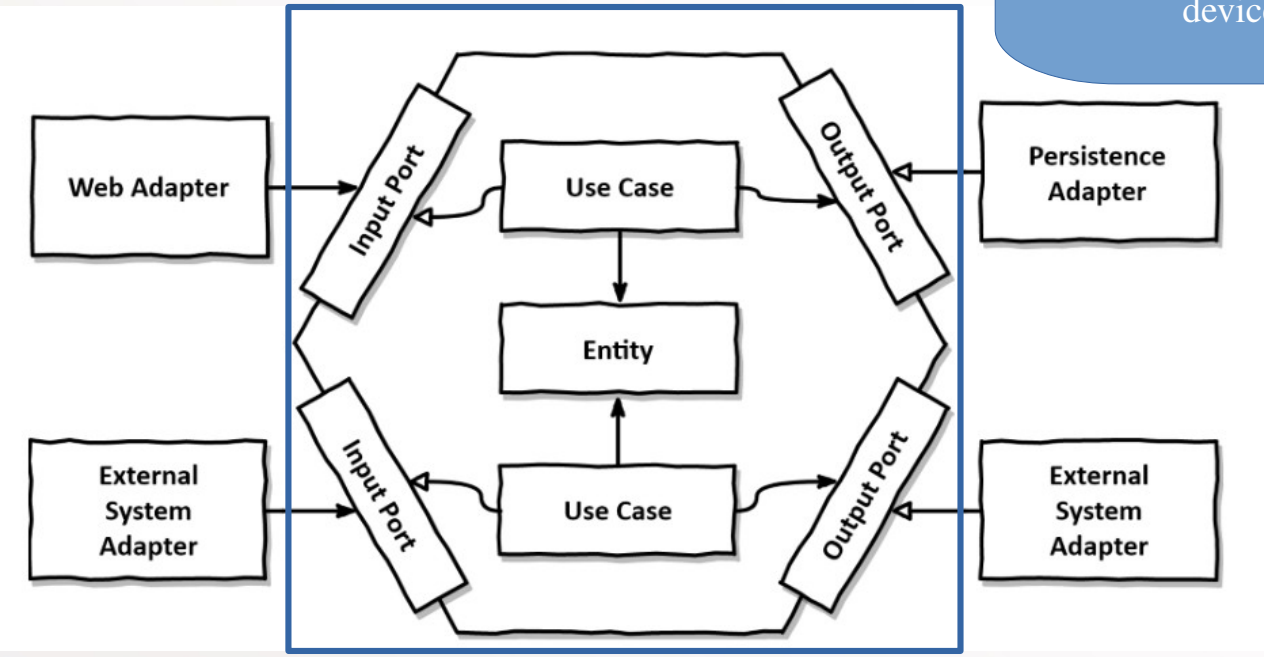
- Souvent **plus de code** → Ne pas multiplier les couches
- Peut impliquer de la **duplication**,
  utiliser des couches transverse
- Attention aux couches «**passe-plats**»
- Attention aux **performances** (mapping)
- On ne respecte pas l'inversion de dépendance (SOLID) entre le domaine et l'accès aux données
- Favorise le **design orienté base de donnée**
- Favorise les **raccourcis** (saut de layers)
- **Domaine difficile à tester**


il existe d’autres
approches
émergentes

L'architecture hexagonale (Alistair Cockburn)

« Ports and adapters »

« Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and **tested in isolation** from its eventual run-time devices and databases. »



Source : Tom Hombergs
http://leanpub.com/get-your-hands-dirty-on-clean-architecture

L’architecture hexagonale (Alistair Cockburn)

- Séparation forte entre :
  - Le **domaine**/logique métier
    - Objets « nus » : aucune technologie
  - Les briques technologiques : IHM, API, bases de données …
    - adaptateurs
  - Deux sortes d'adaptateurs:
    - adaptateurs **sollicitant le domaine** (utilisent des ports « in »)
    - adaptateurs **utilisés indirectement par le domaine** (via les ports « out »)

L’architecture hexagonale

Principe du Dependency Inversion :
les dépendances vont toujours vers le domaine.

Les adapters ne connaissent que des **ports** du domaine.

- **Ports in** : point d’entrée du domaine (fonctions exposées uniquement - SOLID). Implémentés par le domaine.
- **Ports out** : fonctions d’infrastructure dont a besoin le domaine (persister, envoyer un mail...). Implémentés par les adaptateurs.

L’architecture hexagonale, les avantages

- **Isole les aspects techniques** (dans les adaptateurs) et les aspects **fonctionnels** purs (domaine)
- **Favorise l’orienté objet riche** (et non anémique) dans le domaine
- Respect du **DIP** (inversion de dépendance)
- Favorise les tests **en isolation** (domaine seul)
  - Plutôt TU et spécifications exécutables du domaine coté domaine
  - Plutôt TI coté adapter.

Seul problème : favorise la duplication, exemple : entité JPA/ classes du domaine. Mais peu impactant si bon outillage (ex: MapStruct).

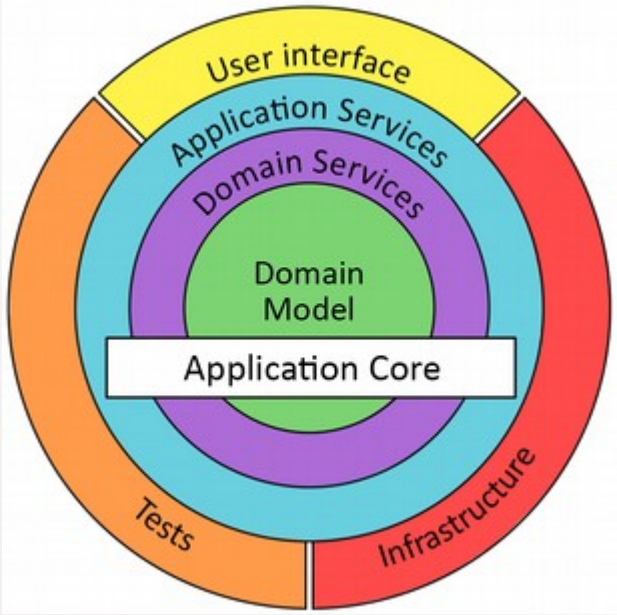
http://leanpub.com/get-your-hands-dirty-on-clean-architecture

```
buckpal
└── account
    ├── adapter
    |   ├── in
    |   |   └── web
    |   |       └── AccountController
    |   ├── out
    |   |   └── persistence
    |   |       ├── AccountPersistenceAdapter
    |   |       └── SpringDataAccountRepository
    ├── domain
    |   ├── Account
    |   └── Activity
    └── application
        └── SendMoneyService
        └── port
            ├── in
            |   └── SendMoneyUseCase
            └── out
                ├── LoadAccountPort
                └── UpdateAccountStatePort
```

Architectures similaires

Onion Architecture (Jeffrey Palermo)

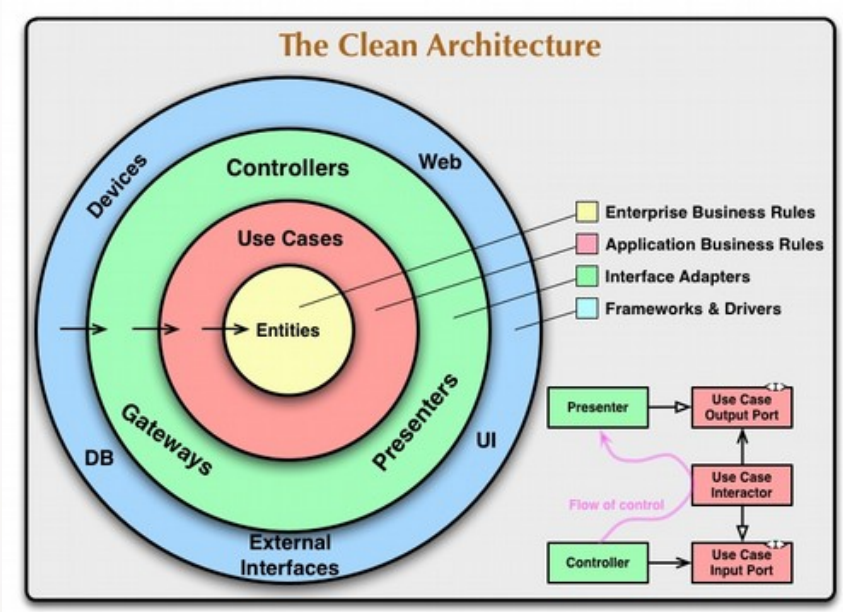
- Voir exemple en [8]



Source : https://dzone.com/articles/onion-architecture-is-interesting

Clean Architecture (Robert Martin)

Voir exemple en [8]



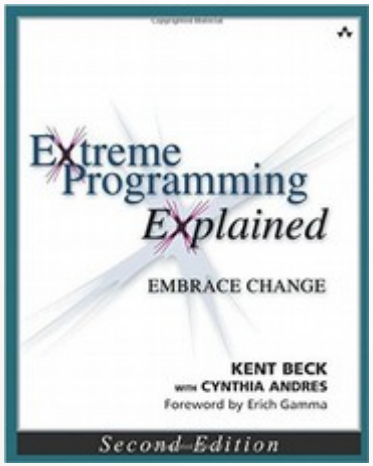
Source : https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html

Les méthodes de développement agiles

Méthodes et pratiques multiples...

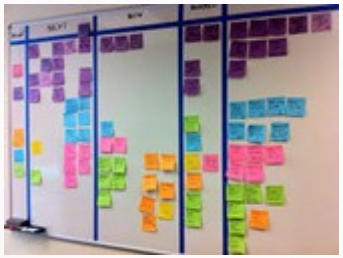
Orienté ingénierie logicielle

Software craftmanship



XP

Scrum



Lean, Kanban

Orienté gestion de projet

Le software craftmanship

- **Professionnaliser** le métier de développeur
- Réponse au code jetable
- De la **qualité**, **pas que du délais**
- Logiciels plus **maintenables**, **performants**, **sécurisés**
- S'auto-former
- Donner du **sens à ce qu'on fait**
- **Collaborer**, échanger entre professionnels

Le manifeste du software craftmanship : extension du manifeste agile

« Non seulement un logiciel opérationnel,
mais aussi au logiciel conçu dans les règles de l'art

Non seulement la réactivité face au changement,
mais aussi à la création permanente de valeur

Non seulement les individus et leurs interactions,
mais aussi à une communauté de professionnels

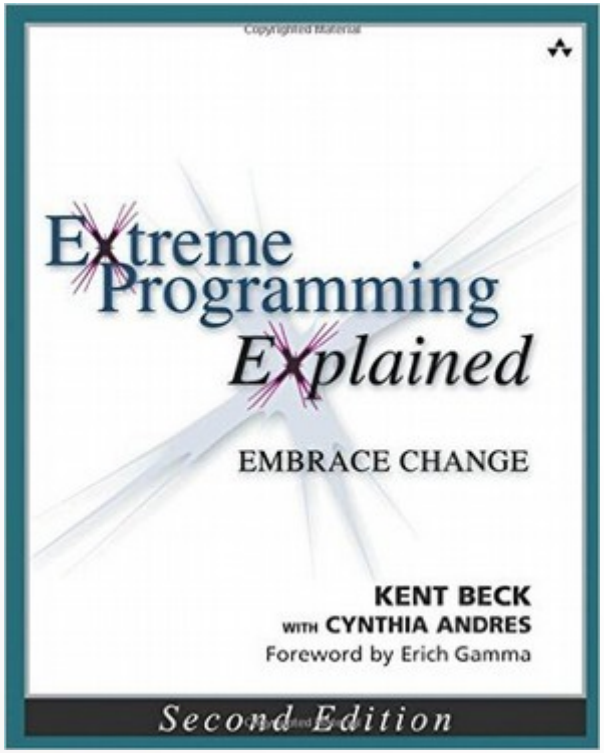
Non seulement la collaboration avec le client,
mais aussi aux partenariats productifs

Dans notre recherche des premiers éléments, nous avons découvert que les seconds étaient indispensables. »

Source : https://agilitateur.azeau.com/post/2010/06/21/Manifeste-pour-l-artisanat-du-logiciel-%3A-proposition-de-traduction-en-fran%C3%A7ais

Le XP (eXtreme Programming)

- Kent Beck, 1999
- Ensemble de valeurs et de pratiques de développement
- Versant « code » de l'agile

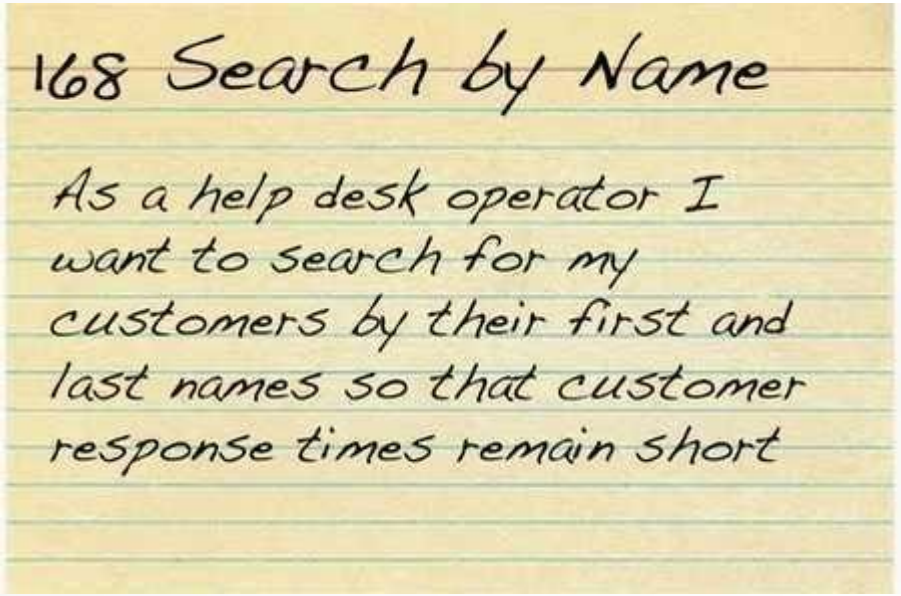


Les pratiques du XP
Pratiques orientées projet

- Sit together : colocalisation de l'équipe
  - **communication osmotique**
- Informative workspace
  - war room, burndown charts…
- Energized work
  - code : 4-5H/J
- **Pair programming**
- Cycles courts
  - Publier souvent et tôt

Les pratiques du XP
Pratiques orientées spécifications

- Incremental design
  - MVP + increments
- Stories
  - En tant que...
  - Je veux …
  - [pour ...]



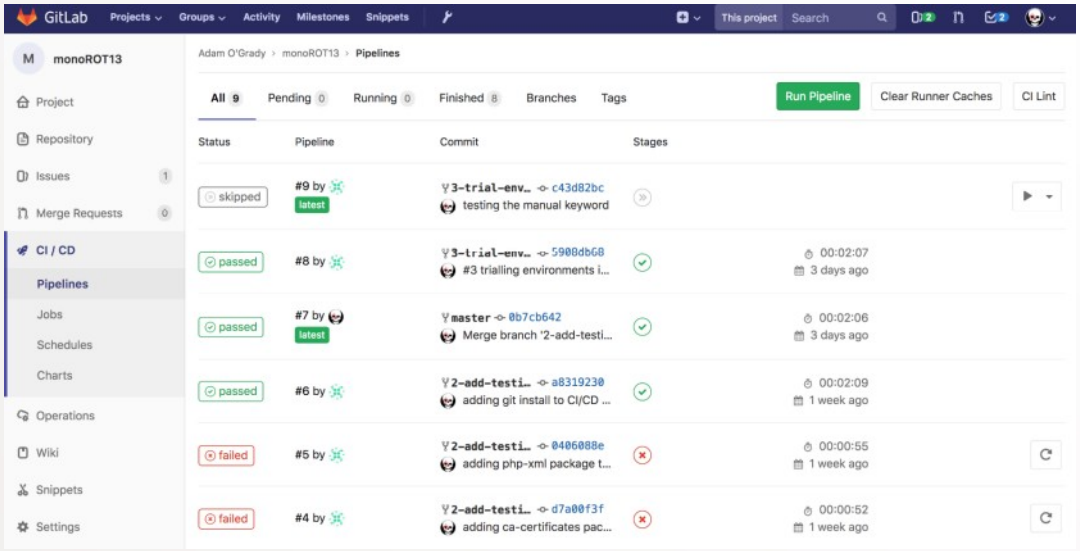
Source : http://itsadeliverything.com/agile-requirements-snail-feature-to-user-story-to-scenario

Les pratiques du XP
Pratiques orientées génie logiciel

- **Refactoring**
- Optimisations à la fin
- Convention over Configuration

Les pratiques du XP
Pratiques orientées build

- Ten Minutes Build
- Continuous Integration (CI)

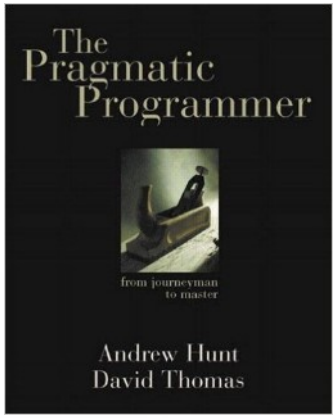


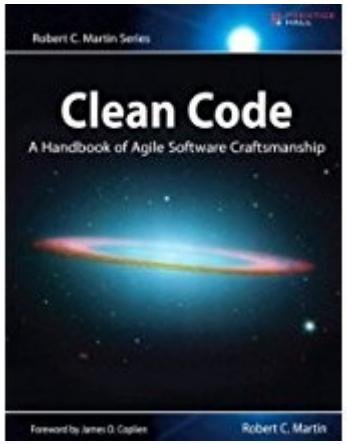
Les pratiques du XP
Pratiques orientées test

- **TDD (Test driven Development)**
  - Test First
- Couverture de code TU très élevée
- Tous les tests doivent passer avant de livrer
- **Tests de confirmation** à chaque bug
- Un bug est un test oublié

Le clean code

- Courant issu de :
  - The Pragmatic Programmer (Andy Hunt and Dave Thomas)
  - Clean Code (Robert Martin)
- Rien de superflu (code, besoin)
- Comprendre le besoin, faire le design
- Le nommage, c'est crucial !
- TDD





Clean code
Les commentaires

- **Le code doit avant tout hurler ce qu'il fait** (screaming architecture)
  - Le nom de la fonction = réponse à « ça fait quoi ? »
  - Pas grave si noms longs

```
List<Personne>  getListePersonnesAvecLivretA()
```

  - pas :

```
/** Retourne la liste des personnes qui possèdent un livret A */

List<Personne> getItems()
```

- Chaque commentaire est un échec
  - Commentaires non maintenus contre-productifs

```
/** @return une moto */
Voiture getVoiture()
```

Quand commenter ?

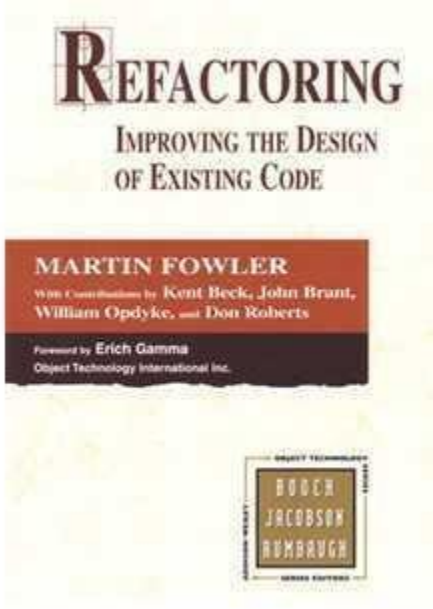
- Lien avec des **notions externes ou métier**

```
// suit la norme XXX
```

- Au besoin dans la description de la classe
- Impérativement **commenter les contrats** (interfaces, endpoints)
  - Être complet
  - Présenter tous les cas, les exceptions runtime …
  - Prendre comme exemple la javadoc du JRE
- Impérativement bien commenter les librairies et classes utilitaires
- en dehors : pas besoin de commentaires. **Chaque commentaire est un échec** de clarté du code.

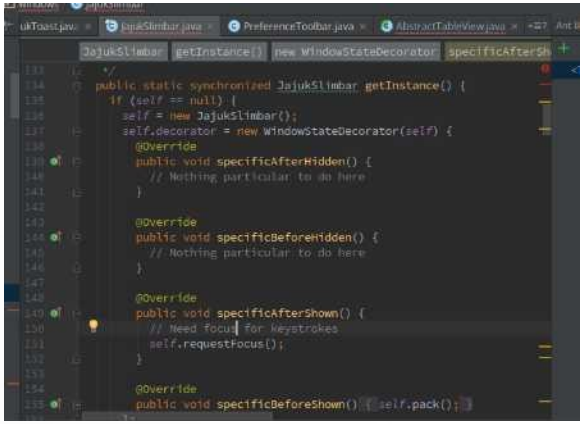
Le refactoring

- Pas de correction de bugs, pas de nouvelles fonctionnalités
- **Hygiène**, respiration du logicielle
- Remboursement dette technique
- Le principe du **boy-scout** : « Leave the camp cleaner than you found it. »
- Doit être provisionné
  - 20 % du budget développement idéalement
- Techniques (voir [11], [12], [17]), utiliser l'IDE



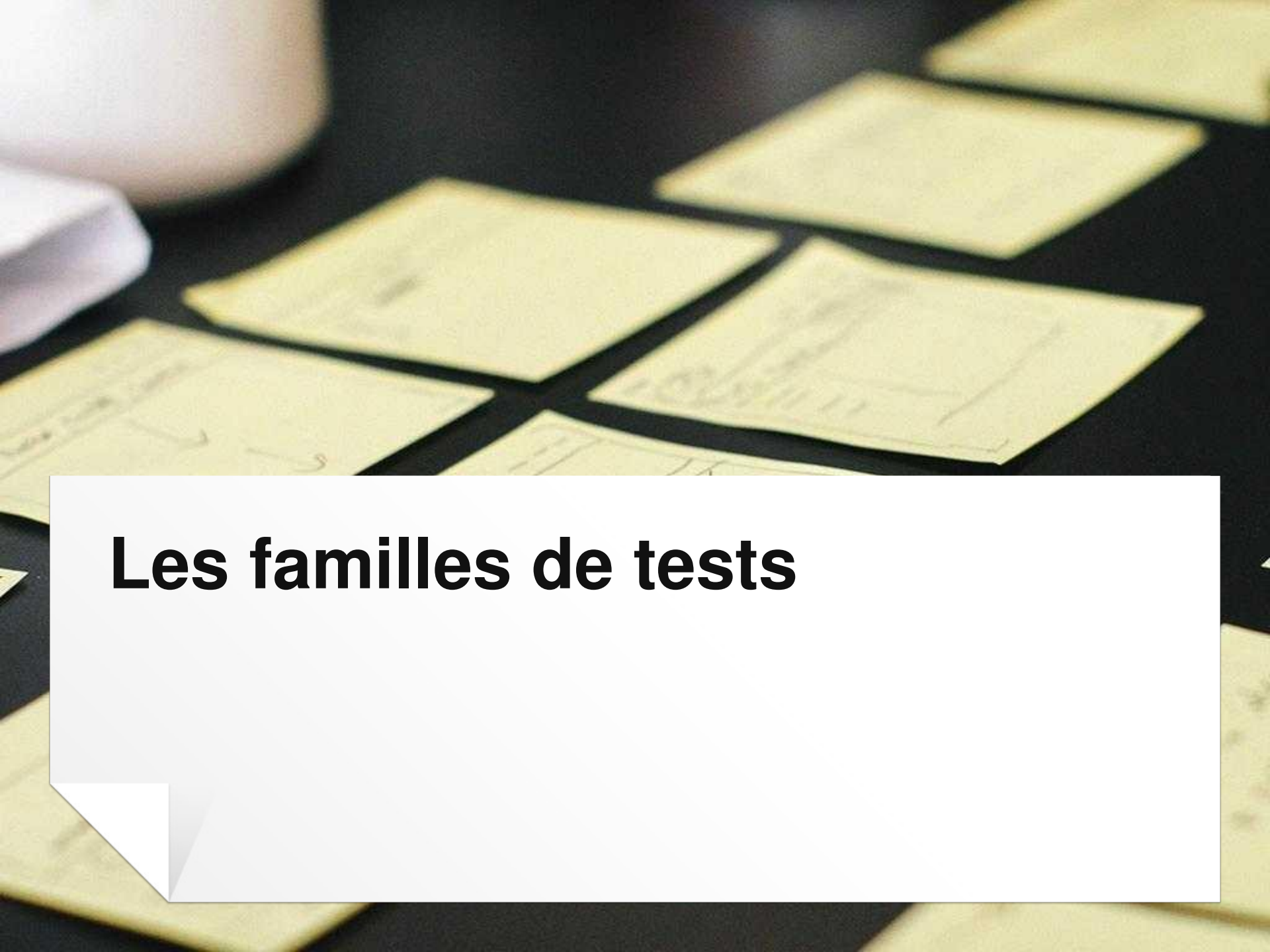
Le formatage du code

- Plus **important** qu'on le pense
- Tout le monde utilise le même
- Partage de la configuration
  - SCM (Git)
  - ou (mieux): utiliser les défaut de l'IDE
- Attention aux risques sur l'historique SCM



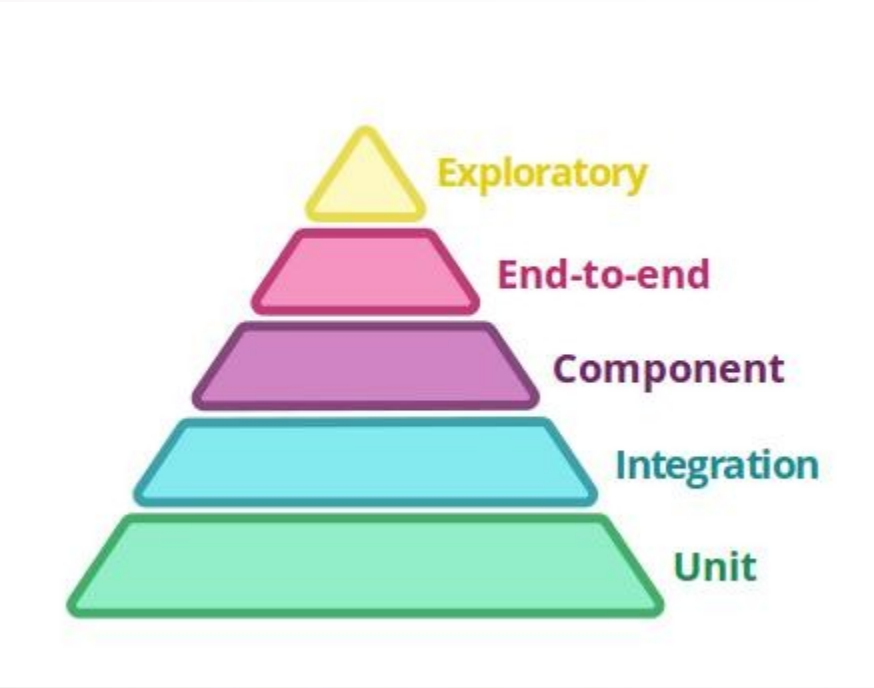
L'artisanat et l'industriel : complémentaires
La cobotique logicielle

- IDE
  - **Refactoring** (ex : extract method)
  - **Formatage**
  - Tests en continu
- CI (intégration continue)
  - **Tâches longues** (build, analyse, tests), prévient en cas de problème
- Analyse **qualimétrique** (dans CI ou IDE)



Les familles de tests

La pyramide des tests



source : https://martinfowler.com/articles/microservice-testing/

Il faut tester avant de livrer

Orienté fonctionnel

Support du projet (aide au développement, permet le refactoring, l'émergence de design...)

automatisés - Spécifications exécutables (BDD) Q2	**manuels** - tests d'acceptation utilisateur (UAT) - tests d'usabilité - tests exploratoires Q3
automatisés Q1 - tests unitaires - tests d'intégration - tests de composants - tests bout en bout	Q4 **outils spécifiques** - tests de charge - tests de sécurité ...

Critique du projet (on tente de trouver les problèmes)

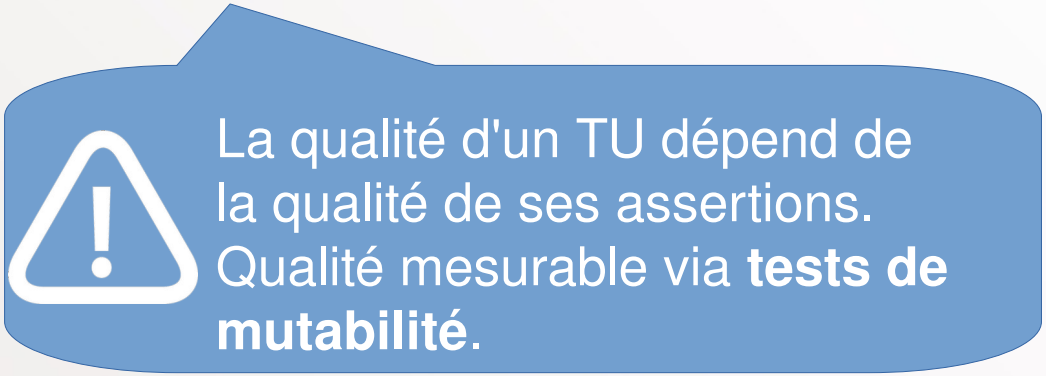
Orienté technologie

Adapté librement du Testing quadrant de Brian Marick et de l'Agile test Quadrant de Lisa Crispin

les principaux types de tests
Q1 Tests unitaires

Automatisé ?	Exemple d'outils	Black ou white box ?	Ecrit par
toujours	junit, Spock, testNG, jest	white box	le développeur

- Teste une (unique) méthode publique
- À écrire avant le code (test first)
- Ne dépend que du code : pas de base de données, pas d'API
- bouchons : dummy, fake, stubs, mocks...



les principaux types de tests
Q1 Tests d'intégration

Automatisé ?	Exemple d'outils	Black ou white box ?	Ecrit par
toujours	junit, spock,testNG	white box	le développeur

- Teste les **dépendances techniques** d'un composant (mapping JPA, serveur SMTP, autre API...)
- Séparés des TU dans la CI car plus longs et plus fragiles

les principaux types de tests
Q1 Tests de composants

Automatisé ?	Exemple d'outils	Black ou white box ?	Ecrit par
toujours	junit, spock, testNG	black box	un autre développeur / un intégrateur

- Teste le **composant depuis son interface**
- Uniquement le composant : dépendances bouchonnées
- Bouchonne les services externes, persistance
  - SGBD en mémoire quand possible (H2, HSQLDB…)
- Appels HTTP « in process » possibles (ex : inproctester)

les principaux types de tests
Q1 Tests de bout en bout

Automatisé ?	Exemple d'outils	Black ou white box ?	Ecrit par
toujours	Selenium/WebDriver, Codecept.js, Playright	black box	un autre développeur, un testeur formé

- **Teste l'IHM** en pilotant le navigateur Web
- Complexe, coûteux, fragile
- A réserver à quelques smoke tests

les principaux types de tests
Q2 Spécifications exécutables / BDD

Automatisé ?	Exemple d'outils	Black ou white box ?	Ecrit par
toujours	Spock, Cucumber, Jbehave, SpecFlow, Serenity	black box	développeur à partir des RG fournies par la MOA

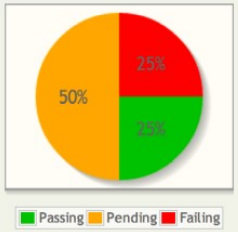
- Spécifications humaines (**Gherkin** : given-when-then)
- **100 % fonctionnel**, 0 % technique
- Steps : relie les spécifications au code
- en général appel in-process à l'entrée du composant

Cucumber + Serenity

```
@DAF:rpm
Feature: Calculer une pension RPM de droit direct d'un marin
 En tant qu'agent instructeur ou chef de groupe
 Je souhaite calculer une pension RPM de droit direct d'un marin

Scenario: Je veux effectuer un calcul de pension RPM de droit direct pour le marin PELLEN Francois
Given un marin dont l'ID est 10064756 et un demandeur dont l'ID est 10064756
When je lance le calcul d'une pension RPM avec les caracteristiques suivantes : <codeTypePension> <codeConcession> <codeExo> <categorie>
Then la pension possède une retenue ? : <possedeRetenues>

Examples:
|codeTypePension|codeConcession |codeExo|categorie  |dateJouissance |dateDemande|dateCompletude |tauxPension|possedeRetenues|
|-- Cas code EXO 3 : il doit y avoir des retenues dans ce cas
|2              |1              |2      |7          |07/10/2015     |07/10/2015 |07/10/2015     |10         |true           |
|-- Cas code EXO different de 2 : pas de cotisations sociales
|2              |1              |3      |7          |07/10/2015     |07/10/2015 |07/10/2015     |10         |false          |
```

Capability: Learn Word Definitions

In order to learn the meaning of a word that I don't know
As an online reader
I want to be able to find out the meaning of the word



Child Requirements:	2
Child requirements without tests:	*0*
Tests:	4
Passing tests:	*1*
Failing tests:	*1*
Pending tests:	*2*
Estimated unimplemented tests:	0

Stories (2)

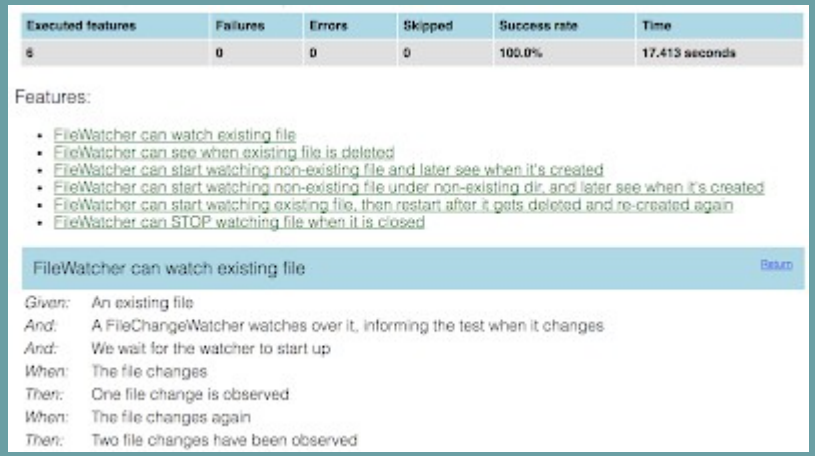
Show 10 entries
Search:

ID	Story	Tests	Pass	Fail	Pend	Coverage
	Lookup definitions *In order to understand what I am reading about* *As a reader* *I want to be able to look up definitions of words*	3	1	1	1	33.3%
	Word of the day *In order to improve my vocabulary* *As an online reader* *I want to regularly learn the meaning of new words*	1	0	0	1	0%

Showing 1 to 2 of 2 entries

Spock

```
class OrderedInteractionsSpec extends Specification {
  def "collaborators must be invoked in order"() {
    def coll1 = Mock(Collaborator)
    def coll2 = Mock(Collaborator)

    when:
    // try to reverse the order of these invocations and see what happens
    coll1.collaborate()
    coll2.collaborate()

    then:
    1 * coll1.collaborate()

    then:
    1 * coll2.collaborate()
  }
}
```



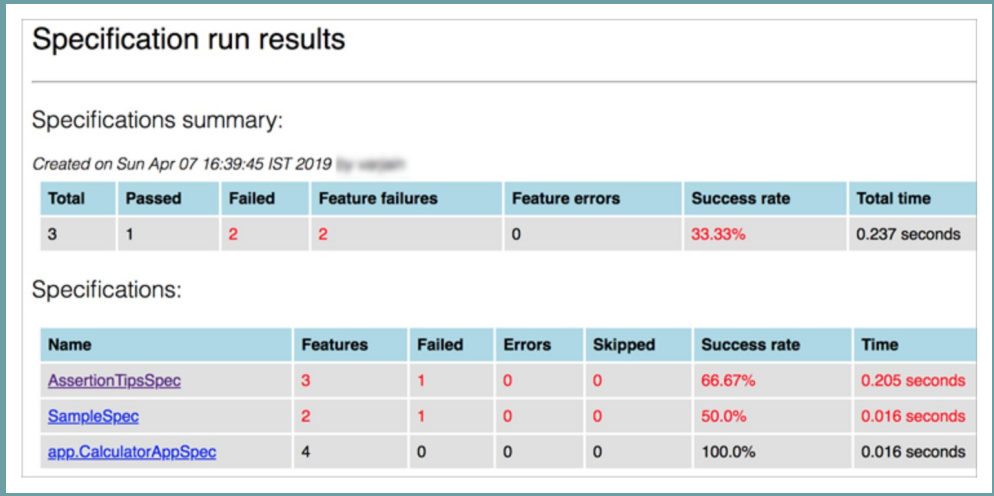
Executed features	Failures	Errors	Skipped	Success rate	Time
6	0	0	0	100.0%	17.413 seconds

Features:

- FileWatcher can watch existing file
- FileWatcher can see when existing file is deleted
- FileWatcher can start watching non-existing file and later see when it's created
- FileWatcher can start watching non-existing file under non-existing dir, and later see when it's created
- FileWatcher can start watching existing file, then restart after it gets deleted and re-created again
- FileWatcher can STOP watching file when it is closed

FileWatcher can watch existing file

Given:	An existing file
And:	A FileChangeWatcher watches over it, informing the test when it changes
And:	We wait for the watcher to start up
When:	The file changes
Then:	One file change is observed
When:	The file changes again
Then:	Two file changes have been observed



Specification run results

Specifications summary:

Created on Sun Apr 07 16:39:45 IST 2019

Total	Passed	Failed	Feature failures	Feature errors	Success rate	Total time
3	1	2	2	0	33.33%	0.237 seconds

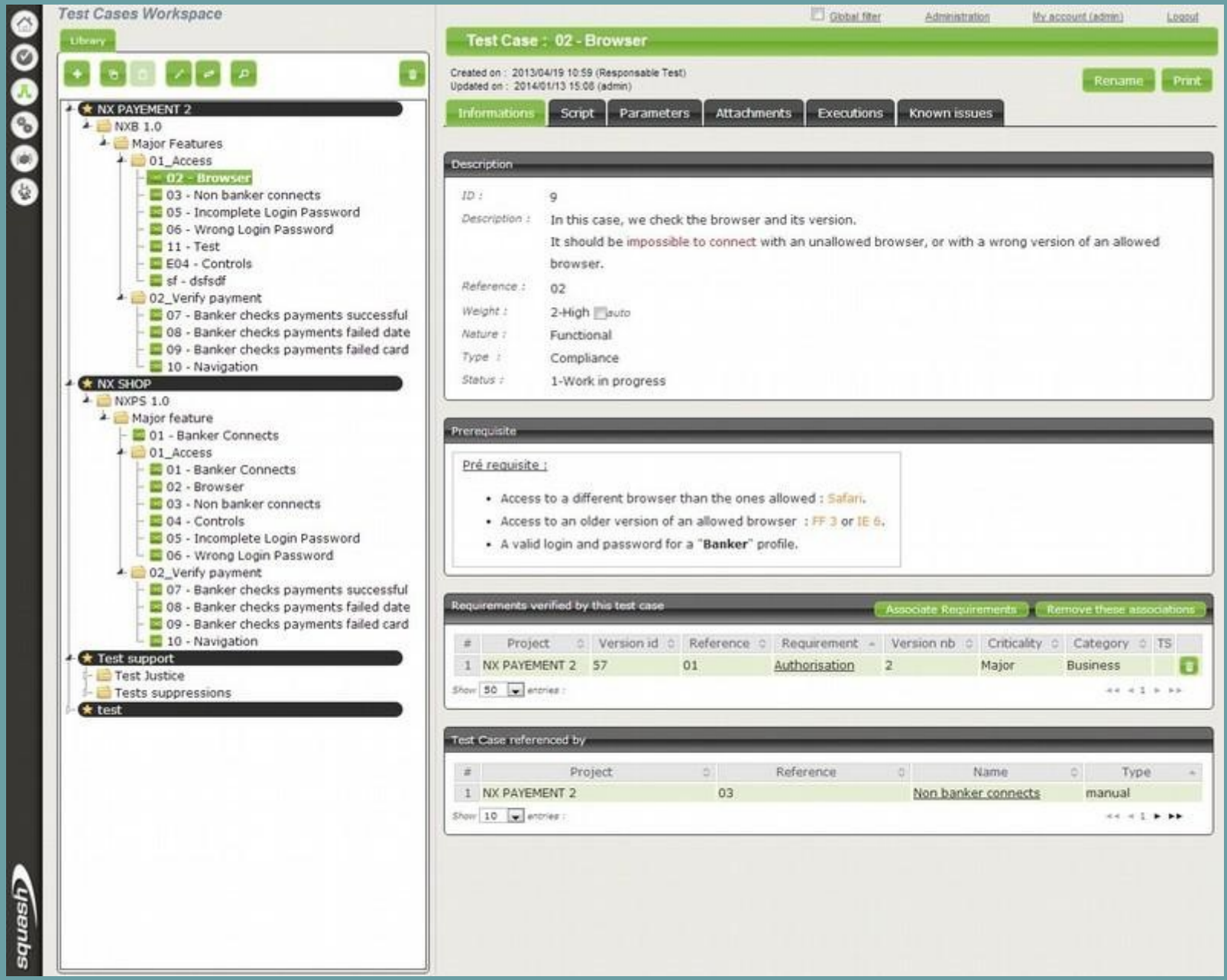
Specifications:

Name	Features	Failed	Errors	Skipped	Success rate	Time
AssertionTipsSpec	3	1	0	0	66.67%	0.205 seconds
SampleSpec	2	1	0	0	50.0%	0.016 seconds
app.CalculatorAppSpec	4	0	0	0	100.0%	0.016 seconds

les principaux types de tests
Q3 Tests d'acceptation fonctionnelle (User Acceptance Test=UAT)

Automatisé ?	Exemple d'outils	Black ou white box ?	Ecrit par
Non mais piloté	HP ALM, SquashTM, TestLink	black box	MOA / fonctionnels MOE

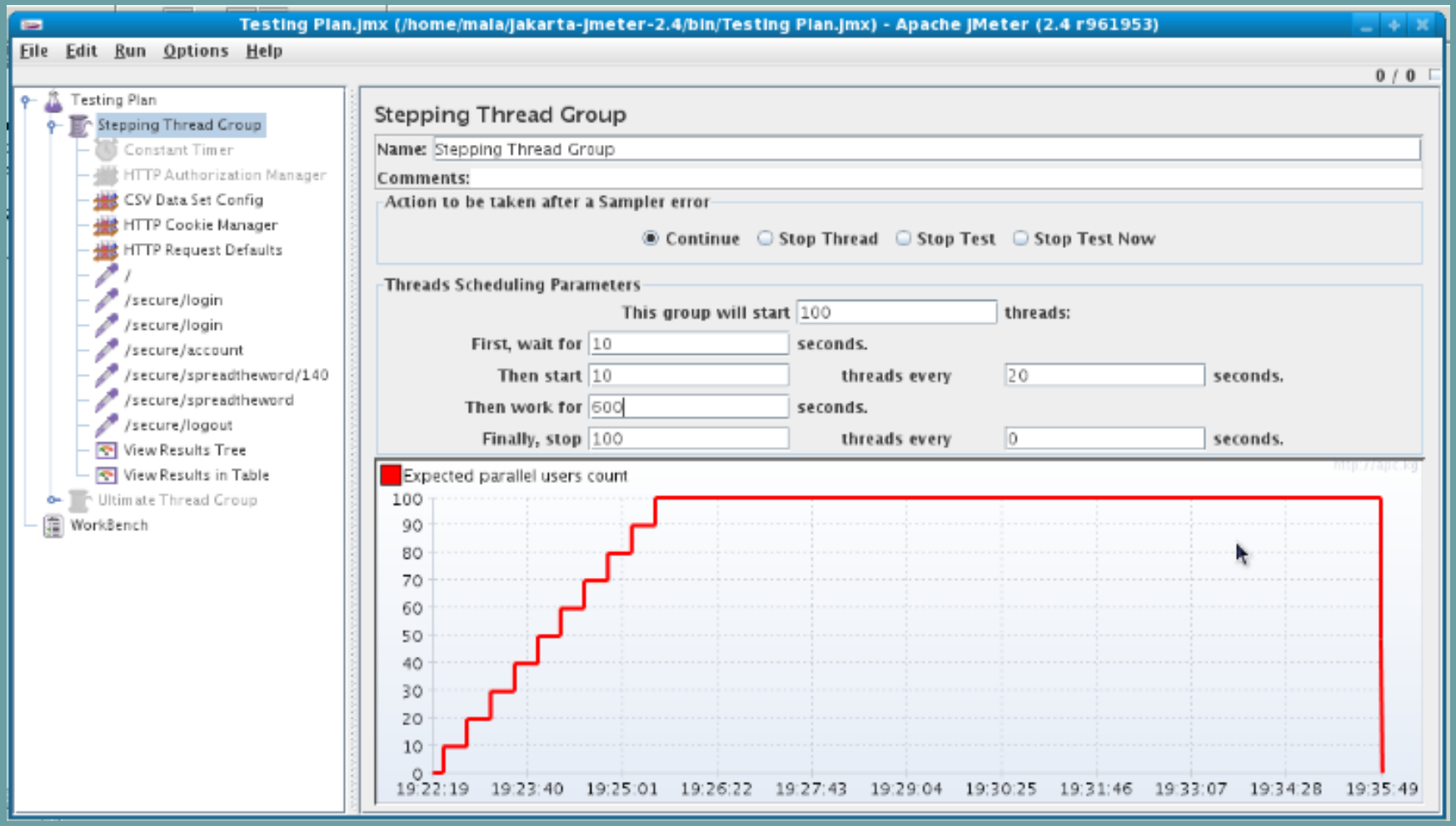
- **Description de tests manuels pas à pas**
- Tests déroulés via un guide
- Gestion de campagnes de tests
- Très cher et peu fiable : limiter le volume


Test Cases Workspace
Library
NX PAYEMENT 2
NXB 1.0
Major Features
01_Access
02 - Browser
03 - Non banker connects
05 - Incomplete Login Password
06 - Wrong Login Password
11 - Test
E04 - Controls
sf - dsfsdf
02_Verify payment
07 - Banker checks payments successful
08 - Banker checks payments failed date
09 - Banker checks payments failed card
10 - Navigation
NX SHOP
NXPS 1.0
Major feature
01 - Banker Connects
01_Access
01 - Banker Connects
02 - Browser
03 - Non banker connects
04 - Controls
05 - Incomplete Login Password
06 - Wrong Login Password
02_Verify payment
07 - Banker checks payments successful
08 - Banker checks payments failed date
09 - Banker checks payments failed card
10 - Navigation
Test support
Test Justice
Tests suppressions
test
Global filter
Administration
My account (admin)
Logout
Test Case : 02 - Browser
Created on : 2013/04/19 10:59 (Responsable Test)
Updated on : 2014/01/13 15:06 (admin)
Rename
Print
Informations
Script
Parameters
Attachments
Executions
Known issues
Description
ID : 9
Description : In this case, we check the browser and its version.
It should be impossible to connect with an unallowed browser, or with a wrong version of an allowed browser.
Reference : 02
Weight : 2-High auto
Nature : Functional
Type : Compliance
Status : 1-Work in progress
Prerequisite
Pré requisite :
Access to a different browser than the ones allowed : Safari.
Access to an older version of an allowed browser : FF 3 or IE 6.
A valid login and password for a "Banker" profile.
Requirements verified by this test case
Associate Requirements
Remove these associations
Project Version id Reference Requirement Version nb Criticality Category TS
1 NX PAYEMENT 2 57 01 Authorisation 2 Major Business
Show 50 entries :
Test Case referenced by
Project Reference Name Type
1 NX PAYEMENT 2 03 Non banker connects manual
Show 10 entries :
tyseus

les principaux types de tests
Q4 Tests de charge

Automatisé ?	Exemple d'outils	Black ou white box ?	Ecrit par
Non	LoadRunner, JMeter	black box	Intégrateur / expert

- Recueil des exigences (dossier d'architecture)
- Enregistrement trames HTTP, variabilisation
- Chargement du système via utilisateurs virtuels
- Analyse indicateurs système
- Différents types dont principalement:
  - Tests de charge (on vérifie que le système supporte la charge nominale)
  - Tests de scalabilité (on observe l'évolution du temps de réponse par rapport à la charge)
  - Tests de stress (on augmente la charge jusqu'à la rupture)
  - Tests d'endurance (pour détecter les fuites mémoire typiquement)


Testing Plan.jmx (/home/mala/jakarta-jmeter-2.4/bin/Testing Plan.jmx) - Apache JMeter (2.4 r961953)
File Edit Run Options Help
0 / 0
Testing Plan
Stepping Thread Group
Constant Timer
HTTP Authorization Manager
CSV Data Set Config
HTTP Cookie Manager
HTTP Request Defaults
/
/secure/login
/secure/login
/secure/account
/secure/spreadtheword/140
/secure/spreadtheword
/secure/logout
View Results Tree
View Results in Table
Ultimate Thread Group
WorkBench
Stepping Thread Group
Name: Stepping Thread Group
Comments:
Action to be taken after a Sampler error
Continue Stop Thread Stop Test Stop Test Now
Threads Scheduling Parameters
This group will start 100 threads:
First, wait for 10 seconds.
Then start 10 threads every 20 seconds.
Then work for 600 seconds.
Finally, stop 100 threads every 0 seconds.
Expected parallel users count
100
90
80
70
60
50
40
30
20
10
0
19:22:19
19:23:40
19:25:01
19:26:22
19:27:43
19:29:04
19:30:25
19:31:46
19:33:07
19:34:28
19:35:49

Introduction au TDD et BDD

Les trois lois du TDD

- Loi no 1 : Vous devez écrire un **test** qui échoue **avant** d'écrire le **code** de production correspondant.
- Loi no 2 : Vous devez écrire **une seule assertion** à la fois, qui fait échouer le test (ou qui échoue à la compilation).
- Loi no 3 : Vous devez écrire le **minimum de code** de production pour que **l'assertion** du test actuellement en échec soit **satisfaite**.

(Définition de Robert Martin) Exemple : [18]

Assertions, tests et exigences

- Le cycle du TDD concerne **UNE unique assertion**
- Pour chaque exigence à respecter par le code (règle de gestion), il faut 1..n assertions
- **Exemple pour une calculatrice** :
  - RG1 : la fonction '+' doit prendre deux arguments
    - Assertion 1 : erreur si 0 arguments
    - Assertion 2 : pas erreur si 2 arguments
  - RG2 : la fonction '+' doit retourner la somme des deux arguments
    - Assertion 1 : 2+2 = 4

Les types de bouchons (doubles)

- **Dummy** : valeur obligatoires mais non utilisées
  - ex : faire(**12**,**"foo"**)
- **Fake** : implémentation light du composant
  - ex : base H2
- **Stub** : renvoi toujours la même réponse ou une réponse en fonction de la requête
  - ex : fichier xml ou json derrière un serveur HTTP
  - ex : fonction stub(x) = "foo" si x=3, "bar" si x=4 …
- **Spy** : appel de l'objet initial avec écoute
- **Mock** : « objets simulés qui reproduisent le comportement d'objets réels de manière contrôlée »

source : https://martinfowler.com/articles/mocksArentStubs.html

Les mocks

- Mock : **contrôle du comportement** (on vérifie les arguments passés ou le nombre d'appels)
- Peut être couplé à du stubbing
- Quelques frameworks Java :
  Spock, Jmock, Mockito, EasyMock

```
interface Subscriber {
    void receive(String message);
}
```

Show for: JUnit 3 | JUnit 4 | Other

```
import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;
import org.jmock.Expectations;

@RunWith(JMock.class)
public class PublisherTest {
    Mockery context = new JUnit4Mockery();

    @Test
    public void oneSubscriberReceivesAMessage() {
        // set up
        final Subscriber subscriber = context.mock(Subscriber.class);

        Publisher publisher = new Publisher();
        publisher.add(subscriber);

        final String message = "message";

        // expectations
        context.checking(new Expectations() {{
            oneOf (subscriber).receive(message);
        }});

        // execute
        publisher.publish(message);
    }
}
```

Source : http://jmock.org

Spock

```
class ItemServiceTest extends Specification {

    ItemProvider itemProvider
    ItemService itemService
    EventPublisher eventPublisher

    def setup() {
        itemProvider = Stub(ItemProvider)
        eventPublisher = Mock(EventPublisher)
        itemService = new ItemService(itemProvider, eventPublisher)
    }
```

```
def 'should publish events about new non-empty saved offers'() {
    given:
    def offerIds = ['', 'a', 'b']

    when:
    itemService.saveItems(offerIds)

    then:
    1 * eventPublisher.publish('a')
    1 * eventPublisher.publish('b')
}
```

```
1 * itemProvider.getItems(['item-id']) >> [new Item('item-id', 'name')]
```

Quelques bonnes pratiques

- Faire les **tests avant le code** (test first)
- Un test ne vérifie qu'**une seule chose** (une méthode)
- Mocks, spy ou stubs ?
  - **Au cas par cas**, préférence aux stubs
- Pas de tests sans assertions
- **TU très rapide** (de l'ordre de 10 / sec maximum)
- Utiliser l'approche Gherkin (Given-When-Then) même pour les TU/TI.
- **Éviter la duplication** : utiliser le BDD ou les tests paramétrés junit (voir https://github.com/junit-team/junit4/wiki/Parameterized-tests)

Quelques bonnes pratiques (suite)

- Tests et classes testées dans le **même package**
  - mais dans des répertoires de sources différents
- Les tests ne doivent **pas** obliger à **tordre le code** pour le rendre testable
- A chaque bug, commencer par écrire un **test de confirmation** (retesting)
  - Le test ne doit pas passer
  - puis corriger le code et vérifier que le test passe
  - En TDD, un bug est un test oublié

Quelques bonnes pratiques (suite)

- **Injection** possible dans les tests aussi mais **pas obligatoire**
  - Voir CDIUnit avec JEE
  - Préférer l'injection par constructeur ou setter
- Un TU ne teste **qu'une méthode**
- Ne pas tester les méthodes privées
  - **uniquement publiques ou package**
- Ne pas tester les **méthodes triviales** (getter/setters)
- Ne pas chercher à couvrir 100% du code
- Ne pas tester le **code externe**
  - frameworks (hibernate), JRE ...

Quelques bonnes pratiques (suite)

- **TU != TI** (confusion courante)
- Un TU ne doit dépendre que de **code en mémoire**
  - Pas de base de données (même mémoire)
  - Pas de système de fichiers
  - Pas de serveur de mail
  - Pas d'appels d'API externe
- Ordre de grandeur : 10 TU /sec minimum

Quelques bonnes pratiques (suite)

- S'assurer d'avoir un **feedback rapide** en cas d'échec
  - Assistance de l'intégration continue
  - Lancer souvent ses **tests en local**
  - Utiliser un outil de **test continu** (plugins sur tous les IDE voire intégration native comme sur IDEA)

Nommage des tests

- Les classes de test (TestCases)

  TU : fini par « `Test` »

  TI : fini par « `IT` »

  Specs : fini par « Spec »

- Pas de mot « `Test` » dans les noms de tests

  Utiliser la convention de nommage `<méthode testée>_<contexte>_<résultat attendu>`.

- Le nom doit **crier** ce qu'il fait :

```
@Test(expect=MotPasseException.class)
verifMotPasse_motPasseDeMoinsDe8Characteres_LeveErreur()
```

Introduction au BDD (Behavioral Driven Developement)

- Méthode agile (2003, Dan North)
- Complète le TDD avec une approche fonctionnelle
- Se concentre sur **le métier** (les RG - Règles de Gestion)
- **Rend formelles les spécifications fonctionnelles**
  - La MOA et la MOE parlent enfin la même langue (**ubiquitous language**)
  - Fini les specs floues / incohérentes
  - **Spécifications vivantes**, finis les documents Word
  - **Vérification** de l'alignement code/specs **au commit**

Introduction au BDD
Comment ça marche ?

1) On écrit les spécifications avec les responsables fonctionnels (**ateliers**)
   - On part de User Story
   - Langage humain
   - Tournure spécifique : le **Gerkin** (Given-When-Then)
   - Par scénarios, le happy path en priorité
   - Avec des **exemples**
   - Phrases ou **format tabulaire**

```
Scénario: Effectuer un virement bancaire
    Etant donné j'ai un livret A avec 150 euros
    Et j'ai un compte courant avec 300 euros
    Quand je transfère 100 euros du livret A vers le compte courant
    Alors mon solde du livret A est de 50
    Et mon solde du compte courant est de 400
```

Source : Baptiste Mille, Thibaut Rappet

Introduction au BDD
Comment ca marche ?

2) On relie (par regexp) ces phrases à des tests (junit en général), appelés fixtures ou steps

- Les fixtures appellent le véritable code et font les assertions

```
Etant donné j'ai un livret A avec 150 euros
```

```
[Given(@"j'ai un (.*) avec (.*) euros")]
0 références
public void SoitJAiUnAvecEuros(string typeCompte, decimal solde)
{
    CompteBancaire compteBancaire = new CompteBancaire(typeCompte, solde);
    comptesBancaire.Add(compteBancaire);
}
```

```
Quand je transfère 100 euros du livret A vers le compte courant
```

```
[When(@"je transfère (.*) euros du (.*) vers le (.*)")]
0 références
public void QuandJeTransfereDuVersLe(decimal montant, string typeCompteSource, string typeCompteDestination)
{
    CompteBancaire compteSource = ObtenirCompte(typeCompteSource);
    CompteBancaire compteDestination = ObtenirCompte(typeCompteDestination);

    compteSource.EffectuerVirement(montant, compteDestination);
}
```

```
Alors mon solde du livret A est de 50
Et mon solde du compte courant est de 400
```

```
[Then(@"mon solde du (.*) est de (.*)")]
0 références
public void AlorsMonSoldeEstDe(string typeCompte, decimal solde)
{
    CompteBancaire compte = ObtenirCompte(typeCompte);
    Assert.AreEqual(solde, compte.Solde);
}
```

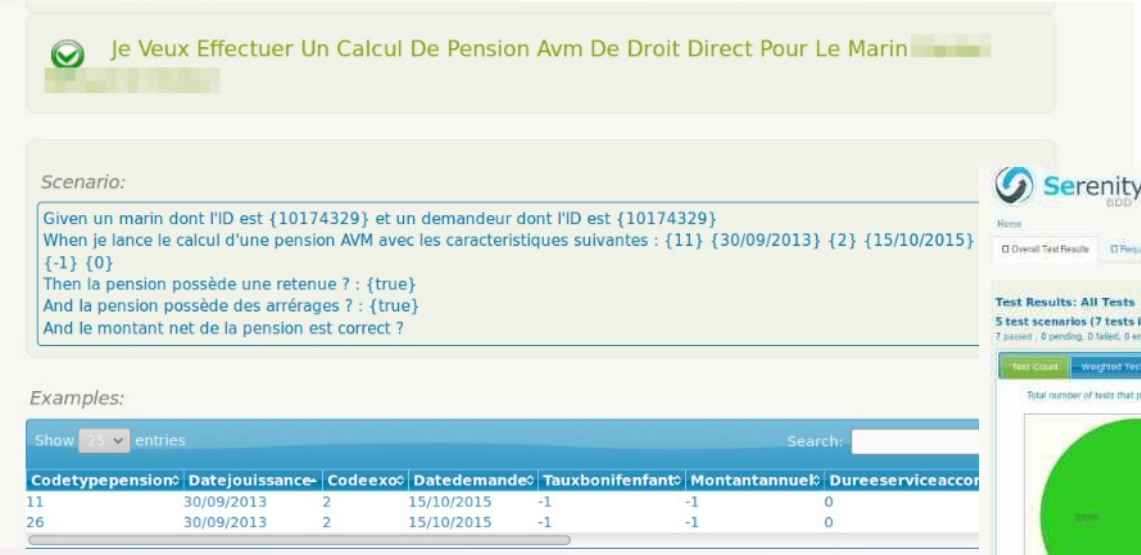
Source : Baptiste Mille, Thibaut Rappet

Introduction au BDD
Comment ca marche ?

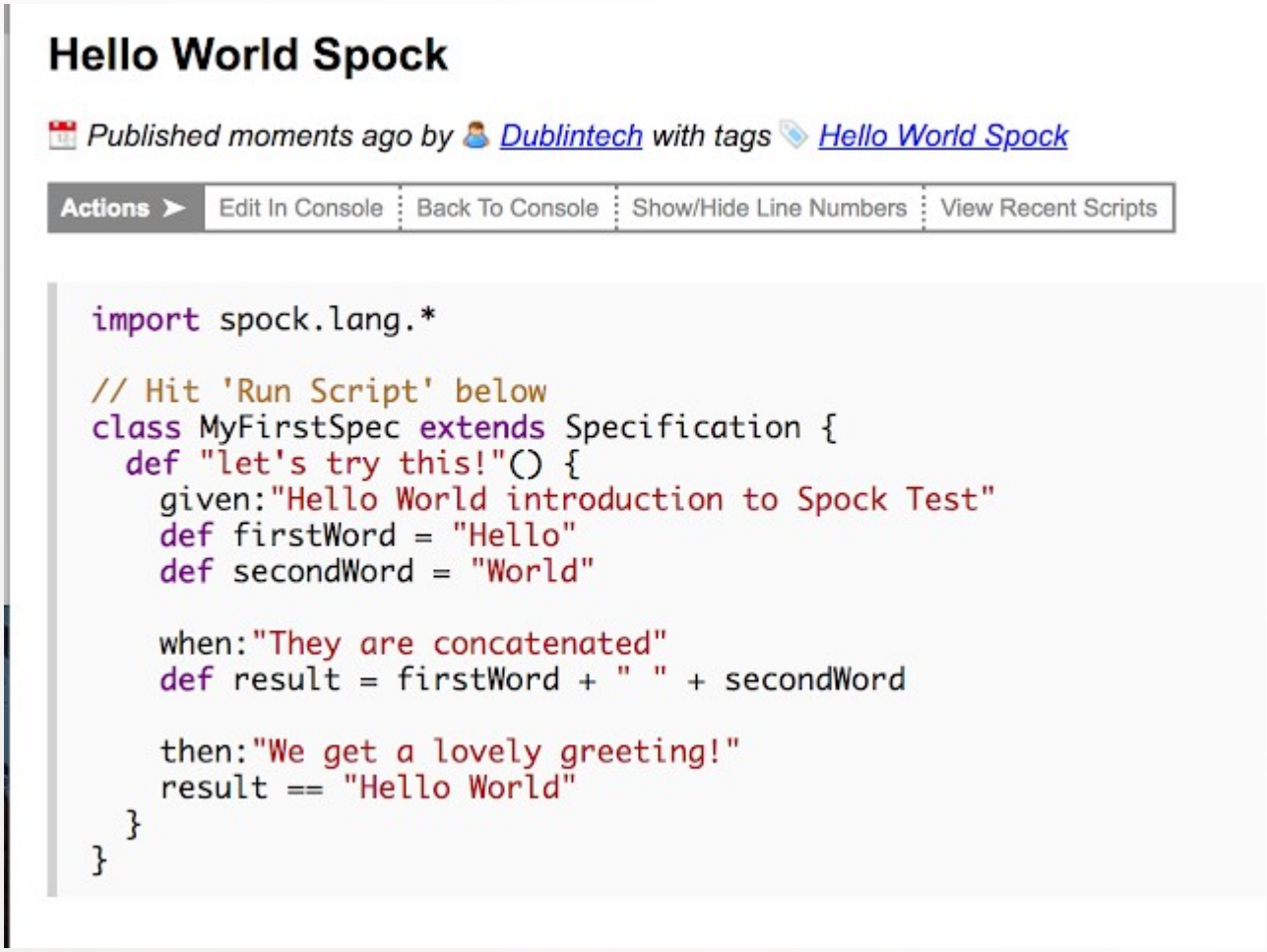
3) On lance les tests (en CI le plus souvent)

On fait du reporting (Serenity)

documentation **vivante**



Approche émergente : Gerkin et fixtures groupées (Spock)



http://spockframework.org/

Introduction au BDD
Le Gherkin

A Gherkin source file usually looks like this

```
1: Feature: Some terse yet descriptive text of what is desired
2:   Textual description of the business value of this feature
3:   Business rules that govern the scope of the feature
4:   Any additional information that will make the feature easier to understand
5:
6:   Scenario: Some determinable business situation
7:     Given some precondition
8:       And some other precondition
9:     When some action by the actor
10:      And some other action
11:      And yet another action
12:    Then some testable outcome is achieved
13:      And something else we can check happens too
14:
15:  Scenario: A different situation
16:      ...
```

Syntaxe de base
(source : https://docs.cucumber.io/gherkin/reference/)

1 feature = 1..n scenarios

Given a stock of <symbol> and a <threshold>
When the stock is traded at <price>
Then the alert status should be <status>

Examples:

symbol	threshold	price	status
STK1	10.0	5.0	OFF
STK1	10.0	11.0	ON

Format paramétré (haut)
et tabulaire (en bas)
(source : http://jbehave.org)

```
Given the traders:
|name|rank|
|Larry|Stooge 3|
|Moe|Stooge 1|
|Curly|Stooge 2|
When Traders are subset to ".*y" by name
Then the traders returned are:
|name|rank|
|Larry|Stooge 3|
|Curly|Stooge 2|
```

Introduction au BDD
Quelques bonnes pratiques

- Itératif / évolutif
  - Faire des tableaux d'exemples à étendre
- Découper les scénarios finement, **un seul *when***
- Factoriser les *given*
- **Test first**
- Données réelles et/ou personas
- Utiliser l'**ubiquitous langage**
- Pas de dépendances entre scénarios

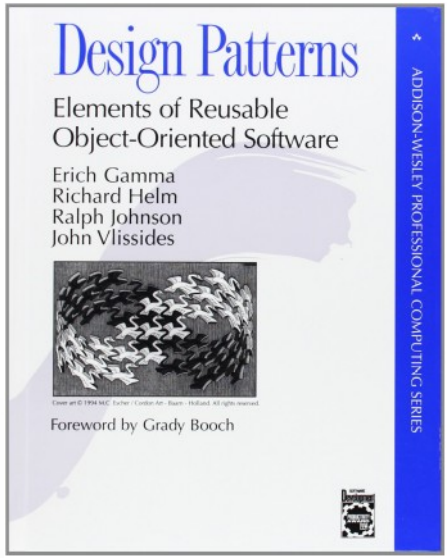
Introduction au BDD
Les outils

Source : Baptiste Mille, Thibaut Rappet

Aperçu des design patterns principaux du GoF

Les design patterns
Introduction

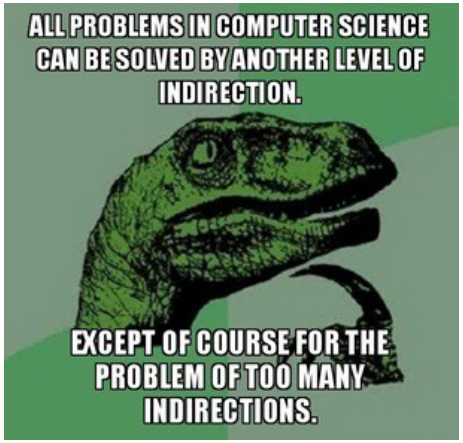
- Solution reconnue à un problème courant
- Souvent Orienté objet
- Christopher Alexander (années 1970) : A Pattern language
- GoG (Gang Of Four) : Design patterns (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides , 1995)



Le GoF : une des bibles de l'architecte logiciel

Les design patterns
Recommandations

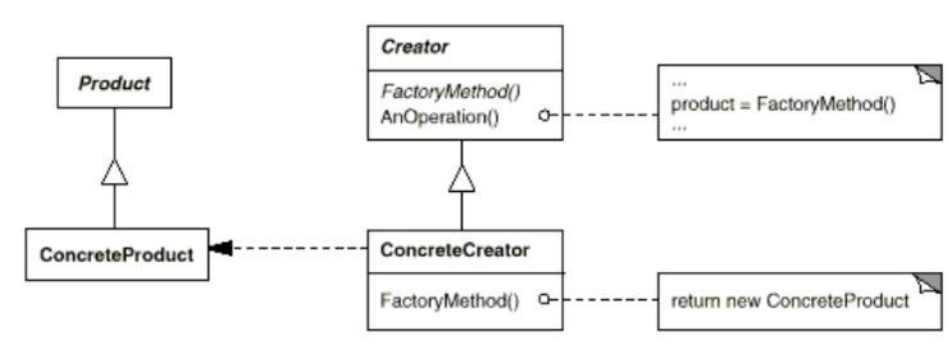
- Même si la plupart des patterns du GoF restent d'actualité :
  - Attention à l'over-design
  - Les langages intègrent de plus en plus ces patterns (ex : Iterator)
  - Débats / anti-pattern ? (ex : singleton et factory)
  - Difficile de tous les maîtriser (23 patterns)



- Ici : patterns les plus pertinents à ce jour

Les design patterns
La factory ou factory method (creational)

- Encore souvent rencontré
- Factory : renvoi un type concret ou un autre en fonction d'une logique
  - peut cacher / scoper
- En général, l'injection de dépendance le remplace



Source : GoF

```java
public class ReaderFactory {

public static Reader getReader(String readerType) {
 Reader reader = null;
 if (readerType.equalsIgnoreCase("XML")) {
     reader = new XMLReader();
 } else if (readerType.equalsIgnoreCase("CSV")) {
     reader = new CSVReader();
 } else if (readerType.equalsIgnoreCase("DB")) {
     reader = new DataBaseReader();
 }
 return reader;
 }
}
```

Source : Siva Prasad Rao Janapati https://dzone.com/articles/the-factory-design-pattern

Les design patterns
La factory simple

Factory « light »

```
public  class Client {

   public void faire(){
        Reader reader = ReaderFactory.getReader ("XML" ) ;
         [...]

   public class ReaderFactory{
       public static Reader getReader(String readerType){
          if ("XML".equals(readerType)){
            return new XmlReader() ;
          }
          else if ("CSV".equals(readerType)){
            return new CsvReader() ;
          }
          else {
             throw new illegalStateException("Type reader inconnu") ;
          }
       }

 }
```

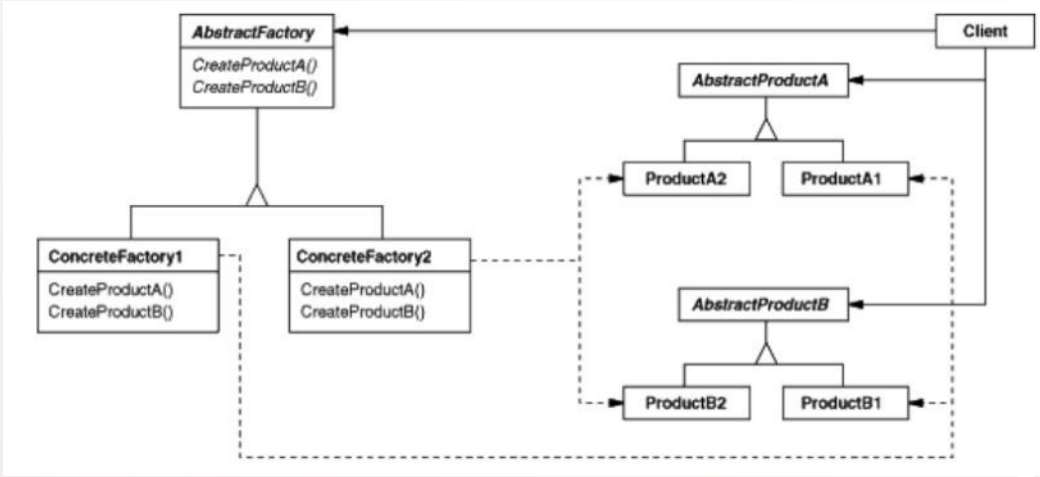
Les design patterns
Le vrai pattern Factory Method

```
public  class Client {

   public void faire(){
        Reader reader = new XmlReaderFactory.getReader () ;
        [...]

 public abstract class ReaderFactory{
   public abstract Reader getReader() ;
}

  public class XmlReaderFactory extends ReaderFactory{
       public Reader getReader(){
            return new XmlReader() ;
        }
  }

  public class CsvReaderFactory extends ReaderFactory{
       public Reader getReader(){
            return new CsvReader() ;
        }
  }
```

-

Les design patterns
L'abstract factory (creational)

- Idem factory mais pour un groupe d'objets apparentés
- C'est le client qui choisi la factory suivant un seul critère
- En général, l'injection de dépendance le remplace



Source : GoF

```
11 public class UnixComponents implements OSComponents {
12  @Override
13  public Window aWindow() {
14  return new UnixWindow();
15  }
16
17  @Override
18  public Menu aMenu() {
19  return new UnixMenu();
20  }
21
22  @Override
23  public Panel aPanel() {
24  return new UnixPanel();
25  }
26 }
27
28 public class WindowsComponents implements OSComponents {
29  // code
30 }
31
32 public class iOSComponents implements OSComponents {
33  // code
34 }
```

Source : Sebastian Malaca
https://dzone.com/articles/abstract-factory-vs-factory-method

Les design patterns
Le singleton (creational)

- Assure d'avoir une seule instance
- Peut faire du lazy-loading
- Peut générer des problèmes de concurrence si mal utilisé (source régulière de bugs subtils)
- A utiliser par injection plutôt que manuellement
- Fonctionnement par défaut de Spring

sur-utilisé

Les design patterns
Le singleton

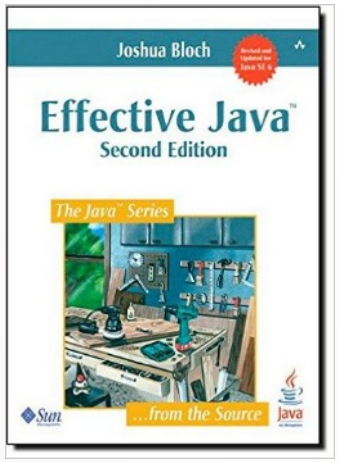
- Implémentation thread-safe et eager (pré-chargé):

```
public class MaClasse {
  private static final MaClasse SELF = new MaClasse() ;

        private MaClasse(){}

        public static MaClasse getInstance(){

                return SELF ;

        }
}
```

- Si disponible, utiliser l'injection de dépendance et un scope `@Singleton`

Les design patterns
Le builder (creational)

- Construire un objet complexe avec contrôle, plus clair qu'un constructeur
- Programmation fluent
- Variante avec interface de builder (peu utilisée)
- Deux écoles :
  - le builder qui produit une classe immuable [15]
  - le builder « simple » (moins verbeux)
- Moins utile avec les avancées des IDE (IDEA permet une lisibilité d'un 'new' avec de nombreux arguments)



Les design patterns
Le builder

- Sans builder :

```
OMailAvecModele oMail = new OMailAvecModele();
Map<String, Object> champs = new HashMap<String, Object>(2);
champs.put("champ1", "Mme");
champs.put("champ2", "Josette");
champs.put("champ3", "Michu");
champs.put("champ4", "http://urltropcool.com?id=545454");
oMail.setChamps(champs);
List<String> listeDestinataire = Arrays.asList(new String[] {
        "toto@site.fr","titi@moniste.fr" });
List<String> listeDestinatairesCopie = new ArrayList<>(2);
listeDestinatairesCopie.add("foo@bar.fr");
oMail.setListeDestinataire(listeDestinataire);
oMail.setCodeModele("MONMODELE");
OPieceJointe opj = new OPieceJointe();
opj.setNomFichier("exemple.pdf");
opj.setTypeMime("application/url");
opj.setContenuEnBytes(pjEnBytes);
List<OPieceJointe> listePieceJointe = new ArrayList<>(1);
listePieceJointe.add(opj);
oMail.setListePieceJointe(listePieceJointe);
```

Les design patterns
Le builder

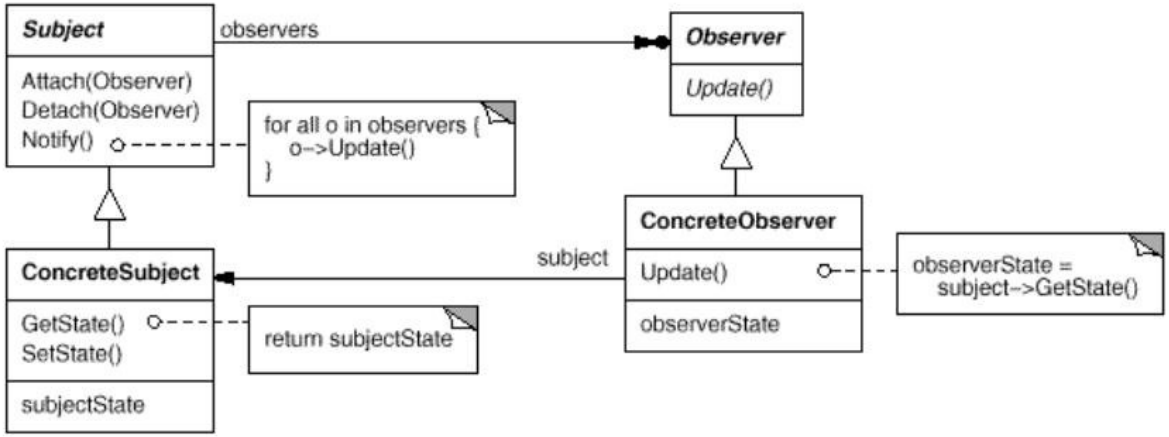
- Avec builder :

```
OMailAvecModele oMail = new OMailAvecModele.Builder("MONMODELE")
        .destinataire("toto@monsite.fr").destinataire("titi@monsite.fr")
        .destinataireCopie("foo@bar.fr").champ("champ1", "Mme")
        .champ("champ2", "Josette").champ("champ3", "Michu")
        .champ("champ4", "http://urltropcool.com?id=545454")
        .pieceJointe("application/url", "exemple.pdf", pjEnBytes).build();
```

Les design patterns
L'observer (behavioral)

- Créer une dépendance 1..n entre objets pour mise à jour simultanée sur événement
- Notification synchrone ou asynchrone
- Attention aux fuites mémoire et à détacher les observers inutiles, attention au debug.

« l'ESB du logiciel »

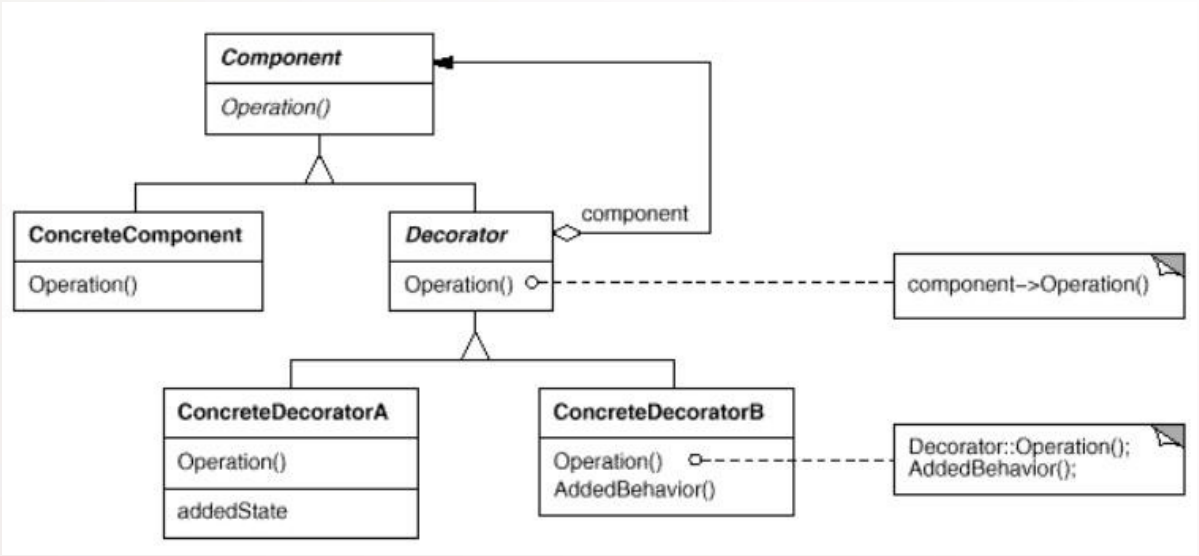


Source : GoF

Les design patterns
Le décorateur (structural)

- Ajouter un comportement à une classe sans héritage
- modification dynamique du comportement
- S**O**LID

«une alternative à l'héritage »



Source : GoF

Les design patterns
Le décorateur

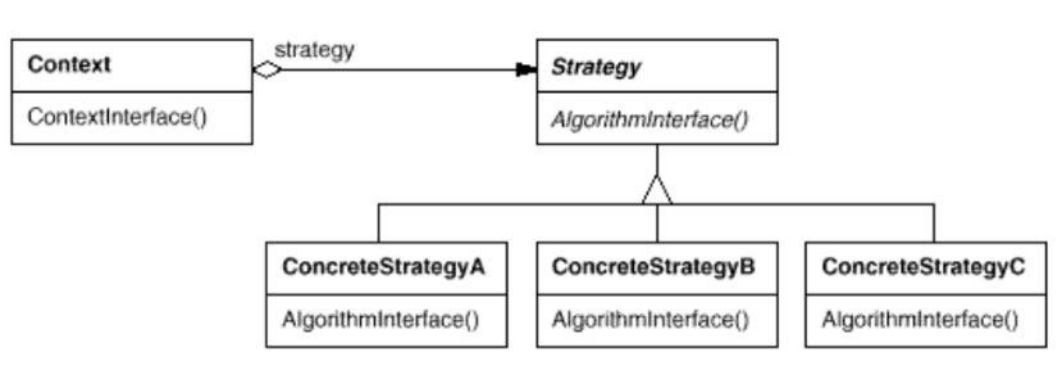
```
public interface Mail {
  public String getListeDestinataires() ;
  ...
}

public abstract class MailDecorator implements
Mail{
  Mail mailOriginal ;
}

// un client
public voids envoyerMail(Mail mail, boolean
important){
  if (important){
    MailAvecChef mac = new MailAvecChef(mail) ;
    smtp.envoyer(mac)
  }
  else smtp.envoyer(mail) ;
}
```

```
public class MailStandard implements Mail {
  public String getListeDestinataires() {
    ...
  }
}

public class MailAvecChef implements Mail {
  private Mail mailOriginal ;
  public MailAvecChef(Mail mail){
    this.mailOriginal = mail ;
  }
  public String getListeDestinataires() {
    return
addChef(mail.getListeDestinataires()) ;
  }
  private List addChef(List liste){
    list.add(chef) ;
    return liste ;
  }
}
```

Les design patterns
Le pattern strategy (behavioral)

- Sélectionner dynamiquement un algorithme ou un autre
- Alternative à l'héritage et aux switchs
- Le choix de l'algorithme est fait dans le contexte

«découpe le code complexe »



Source : GoF

Les design patterns
Le pattern strategy

```
public interface CalculNuite{
  public long calculerPrixNuit(int
nbNuits) ;
}

public class CalculNuiteBasseSaison{
 public long calculerPrixNuit(int nbNuits){
  // algo 1
 }
}

public class CalculNuiteHauteSaison{
 public long calculerPrixNuit(int nbNuits){
  // algo 2
 }
}
```

une stratégie +
deux stratégies concrètes

Le contexte

```
public class Panier {
 private CalculNuite calculNuite ;
  public long calculerCoutTotal(Date
dateEntree,Date dateSortie) {
    if (dateEntree.before(30_MAI){
     calculNuite = new CalculNuiteBasseSaison();
    }
    else{
      calculNuite = new CalculNuiteHauteSaison();
    }
    // algo…
     long prix += calculNuite.calculerPrixNuit(…) ;
  }
}
```

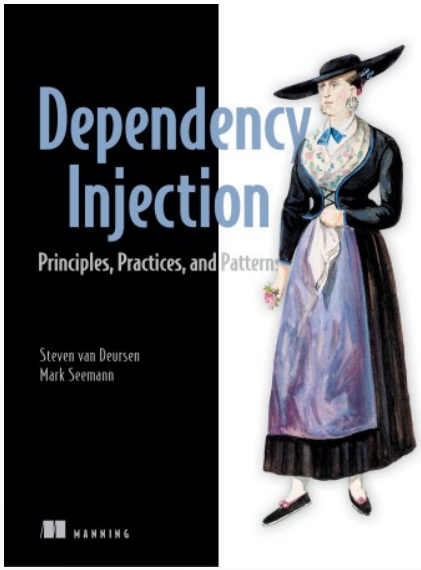
L’Injection de dépendance

L'injection de dépendance
DI : Dependency Injection

Définition : ensemble de patterns et de principes permettant d'écrire du code faiblement couplé

Le pattern de création le plus utile

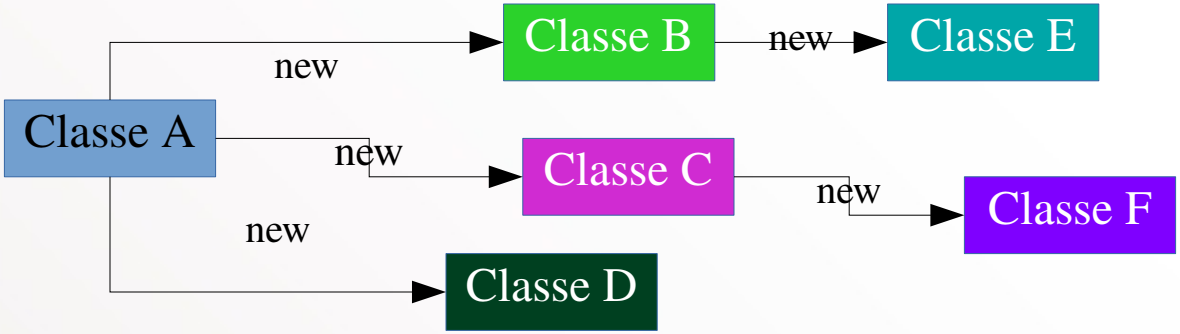
- **Injecte** les objets nécessaires aux objets
- Le '**New new**'
- Frameworks DI : Spring Core, Guice, CDI (Java) , Symphony, @lerna/bootstrap (Node.js) ...
- Les objets ne se connaissent plus entre eux, un container IoC tisse leurs relations



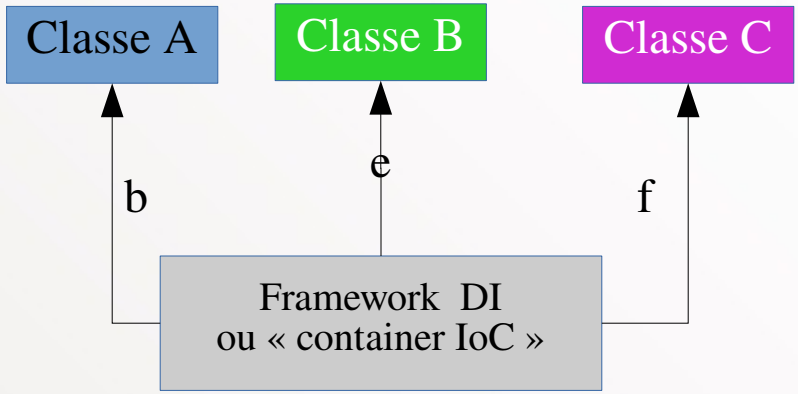
n'améliore pas les performances

L'injection de dépendance

- Avant :



- Après :



L'injection de dépendance
Exemple d'utilisation

```
class MailClient{
    SmtpServer server ;
    MessageFormatter ;

   @Inject
   public MailClient(SmtpServer server,MessageFormatter formatter){
       this.server = server ;
       this.formatter= formatter ;
    }
     public void envoyerMail(Mail mail){
           server.send(formater.format(mail)) ;
    }
}

class SmtpServer{
    Connexion connexion ;
     @Inject
     public SmtpServer(Connexion connexion){
        this.connexion = connexion ;
     }
     public void send(Mail mail){
        connexion.open() ;
        connexion.write(formater.format(mail)) ;
```

(noter l'arbre de dépendance)

```
MailClient -
  |- SmtpServer
       |- Connexion
  |- MessageFormatter
```

SmtpServer est une classe concrète
Connexion est une interface avec deux implémentations :
TCPConnexion et UDPConnexion

L'injection de dépendance
Exemple d'utilisation

Mais comment dans le cas de Connexion, comment le framework a-t-il sélectionné le bon type ?

⇨ Peut se faire explicitement dans le **code** ou dans la **configuration**

Dans la configuration XML (exemple Spring) :

```
<bean id="smtpServer" class="SmtpServer">
<constructor-arg ref="connexion"/>
</bean>
<bean id="connexion" class="TcpConnexion"/>
```

C'est un exemple, maintenant on n'utilise avec Spring que des annotations

Dans le code (exemple CDI avec le qualifier `@Tcp`) :

```
    public SmtpServer(@Tcp Connexion connexion){
}
```

(Peut être surchargé pour les tests : `Alternative` en CDI)

L'injection de dépendance
Les principaux types d'injection : par constructeur

- Préféré (dixit Martin Fowler) car objet construit dans un état valide et complet
- Construction au démarrage
- Parfait pour classes immuables

Martin Fowler, photo Thoughtworks

```
class MailClient{
SmtpServer server ;
MessageFormatter formatter ;

@Inject
public MailClient(SmtpServer server,MessageFormatter formatter){
   this.server= server;
   this.formatter = formatter;

}
```

L’injection de dépendance
Les principaux types d’injection : par setter

- Très utilisée dans Spring (`@Required`)
- Plus facile de tester

```
class MailClient{
SmtpServer server ;
MessageFormatter formatter ;


public MailClient(){}

@Inject
public void setServer(SmtpServer Server){
  this.server = server;
}

@Inject
public void setFormatter(MessageFormatter formatter){
  this.formatter = formatter;
}
```

L'injection de dépendance
Les principaux types d'injection : par attribut (field)

- Ne pas l'utiliser
- Uniquement pour du code jetable
- Rend presque impossible d'injecter des mocks
- Code très concis

```
class MailClient{
 @Inject
 SmtpServer server ;

 @Inject
 MessageFormatter formatter ;
```

L'injection de dépendance
Fonctions avancées des framework de DI

- Injection 'just in time' ou intégration avec code legacy via les providers
- **Scopes** (@Singleton, Prototype/Dependent, @Request, @Session, @Application, custom)
- **AOP :** @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() { // ... })
- Bootstraping du moteur dans une application Web
- **Tests unitaires** (ex : CDIUnit)
- En savoir plus : [19]

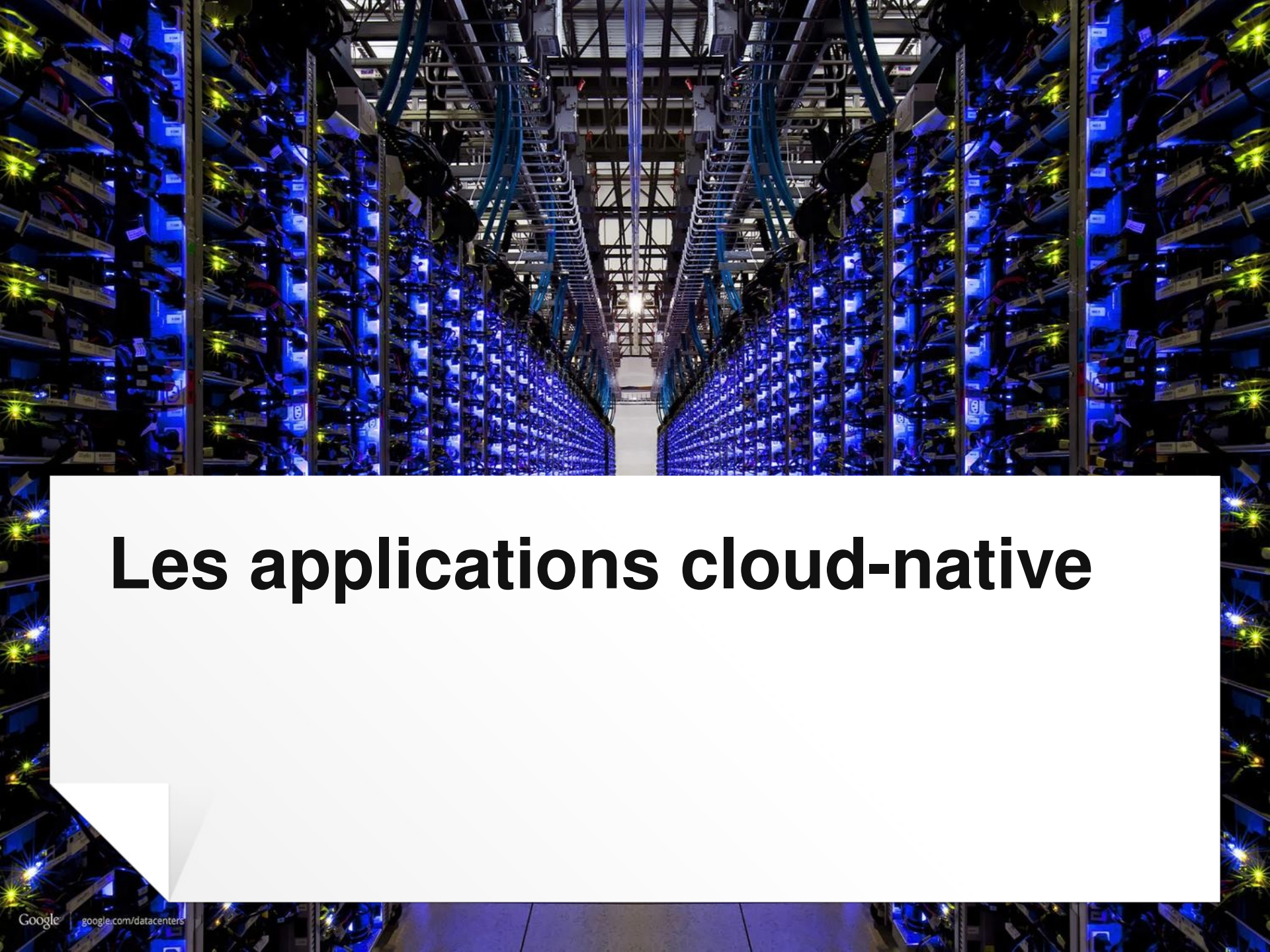
L'injection de dépendance
L' «injection manuelle » : l' injection du pauvre

- Dans des cas très simples ou particuliers, se passer de framework de DI en déclarant systématiquement les **dépendances en constructeur**
- Attention :
  - on déporte la création des objets dans les clients
  - fort impact si on change un type, il va falloir modifier tous les endroits où on l'a utilisé
  - le client doit connaître la structure interne des objets utilisés (perte d'encapsulation)

```
public static void main(){
  TcpConnexion tcp = new TcpConnexion() ;
  SmtpServer serveur = SmtpServerFactory.get("tcp") ;
  MessageFormatter formatter = new MessageFormatter() ;
  this.client = new MailClient(serveur,formatter) ;
}
```

L'injection de dépendance
Les anti-patterns

- **Pas de `static`** lorsqu'on fait de la DI
- Ne pas faire d'injection d'attributs
- Attention aux scopes (surtout en Spring : @Singleton par défaut)
- Pas d'injection de dépendance pour les beans (Entités, DTO...), surtout si construits au runtime



Les applications cloud-native

Qu’est ce qu’une application cloud-native ?

- Application pensée pour **s’exécuter dans un cloud**
- A la croisée des paradigmes de :
  - **DevOps** (exploitants et développeurs ensemble)
  - **Micro Services** (découpage SI en blocs autonomes)
  - **IaC** (le déploiement en production est un non-événement)
  - S’appuie sur la **conteneurisation**

Les Twelve Factors d'Heroku

Les 12 facteurs

I. Base de code
Une base de code suivie avec un système de contrôle de version, plusieurs déploiements

II. Dépendances
Déclarez explicitement et isolez les dépendances

III. Configuration
Stockez la configuration dans l'environnement

IV. Services externes
Traitez les services externes comme des ressources attachées

V. Build, release, run
Séparez strictement les étapes d'assemblage et d'exécution

VI. Processus
Exécutez l'application comme un ou plusieurs processus sans état

VII. Associations de ports
Exportez les services via des associations de ports

VIII. Concurrence
Grossissez à l'aide du modèle de processus

IX. Jetable
Maximisez la robustesse avec des démarrages rapides et des arrêts gracieux

X. Parité dev/prod
Gardez le développement, la validation et la production aussi proches que possible

XI. Logs
Traitez les logs comme des flux d'évènements

XII. Processus d'administration
Lancez les processus d'administration et de maintenance comme des one-off-processes

https://12factor.net/fr/

Les Twelve Factors d'Heroku

I **Codebase** (aka « dépôt unique »)

- Tout le code doit se trouver dans un SCM (git, svn, mercurial...). Approche « **GitOps** ».
- **Un seul dépôt par module** (et pas environnement). Chaque environnement doit être en mesure de déployer tout commit.

II **Dependencies** (aka « build reproductible »)

- **Dépendances** (npm, Maven, apt, dockerfile...) doivent être **listées de façon exhaustives**.
- Et **fixées** (pinned): on donne la version exacte de chaque dépendances

```
FROM httpd:latest
```

```
FROM httpd:2.4.29
```

Les Twelve Factors d'Heroku (suite)

III Config (aka « variables d'environnements »)

- La configuration est **tout ce qui varie** d'un environnement à l'autre (mots de passe, URL...)
- La configuration doit être fournie au runtime par **variables d'environnement**

IV Backing services (aka « composants d'infrastructure interchangeables »)

- Les **backing services** = infrastructure logicielle utilisée (SGBD, MOM, caches distribués, serveur SMTP, API...) internes au SI ou publiques (API Google par exemple)
- Il ne faut **aucune adhérence entre le code et un backing service**, ils doivent être **interchangeables**

Les Twelve Factors d'Heroku (suite)

V Build, release, run (aka « séparation build, livraison et déploiement »)

- Build (construction du binaire indépendant de la plateforme cible) ; Release (livraison) : binaire du build + configuration ; Run : déploiement de la release sur des machines
- **Chaque étape strictement étanches** (ex : build agnostique du déploiement ; pas de modif du code lors de la livraison; pas forcement de déploiement à chaque livraison).

VI Processes (aka « stateless »)

- **Jamais d'états** (stockage d'informations entre deux requêtes en mémoire, sur disque...)
- Si persistance nécessaire, utiliser un service externe (SGBD, base NoSql, cache Redis...)
- Un composant = juste **un processus** (au sens Unix)

Les Twelve Factors d'Heroku (suite)

VII « Port binding » (aka « application auto-porteuse »)

- Plutôt que d'exécuter l'application dans un composant d'exécution (serveur d'application JEE, serveur web...), c'est **l'application qui tire le serveur Web** comme librairie
- Exemple : fat jar (WildFly Swarm, Spring boot avec Tomcat) , Tornado en python...

VIII « Concurrency » (aka « utiliser des processus isolés »)

- Exécuter une application avec un ou **plusieurs processus** (Unix) plutôt que par threads (ex : plutôt une JVM par composant qu'une grosse JVM avec de nombreux threads)
- Processus gérés uniquement par un **système d'initialisation** (comme systemd) : prend en charge le démarrage / arrête des services

Les Twelve Factors d'Heroku (suite)

IX Disposability (aka « démarrages rapide, arrêts gracieux »)

- **Démarrages et arrêts les plus rapides possibles** (élasticité sur les clouds : les conteneurs peuvent être détruits et redémarrés régulièrement, une application peut vivre seulement quelques minutes)
- **Arrêt « propre »** sur signal, par exemple sur SIGTERM, nettoyages

X Dev/prod parity (aka « limiter l'impédance dev/prod »)

- Utiliser la **CI/CD** et des tests d'intégration pour remonter les problèmes au plus vite
- Utiliser **dès le dev les mêmes backing services** qu'en production via les conteneurs (ex : pas de base H2 mémoire mais directement un postgresql via containeurs)

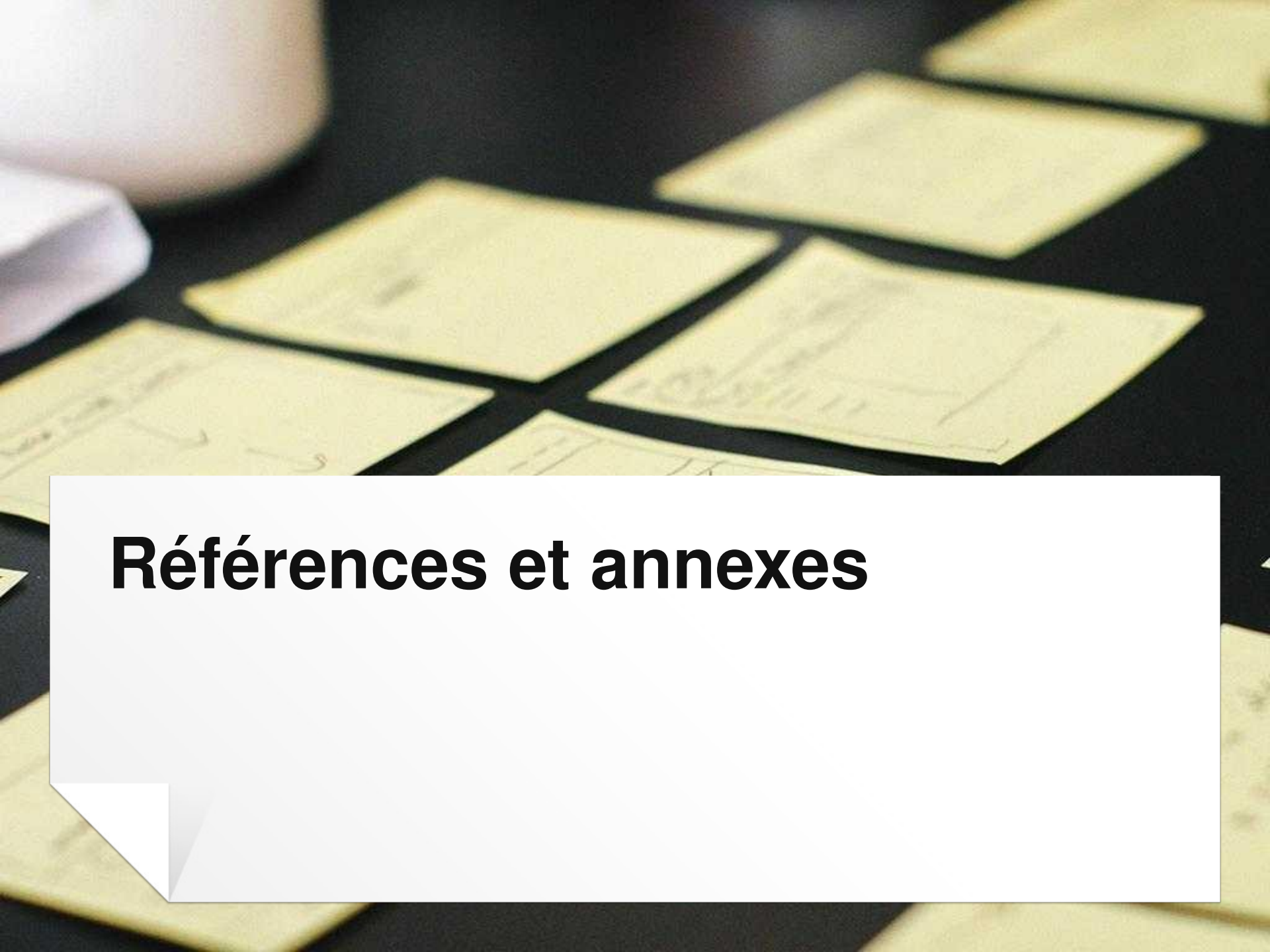
Les Twelve Factors d'Heroku (suite)

XI Logs (aka « sorties standards »)

- Ne **pas loguer dans des fichiers** (nœuds éphémères)
- Utiliser des frameworks de log (slf4j...) mais utiliser uniquement un **appender console** (sortie stdout)
- **Centralisation des logs** (typiquement stack ELK)

XII Admin processes (aka « `/health` »)

- Le composant doit intégrer des **endpoints (REST) d'administration** et supervision au même titre que les endpoints applicatifs, pas d'outils externes
- **Préoccupations d'exploitation dès le développement** (DevOps)
- Permet d'**éviter les problèmes** de synchronisation applicatif/administration (exemple : test verrou dans le code applicatif, verrou posé dans un `/backup`)



Références et annexes

Références

- [1] La méthode Pomodoro : http://cirillocompany.de/pages/pomodoro-technique
- [2] Deep-Work-Focused-Success-Distracted, Cal Newport, ISBN-10: 1455586692
- [3] Software Architecture for Developers (Simon Brown) https://leanpub.com/software-architecture-for-developers/read#leanpub-auto-types-of-architecture
- [4] Site de Robert « Uncle Bob » Martin : www.cleancoder.com
- [5] Site de Martin Fowler : https://martinfowler.com/
- [6] Cours gestion de version Bertrand Florat : https://florat.net/cours_vcs/cours.html
- [7] Architecture hexagonale : http://alistair.cockburn.us/Hexagonal+architecture
- [8] Exemple de Clean Architecture : https://www.slideshare.net/mattiabattiston/real-life-clean-architecture-61242830

Références

- [9] Exemple d'Onion Architecture : https://bitbucket.org/jeffreypalermo/onion-architecture/src/1df2608bc383?at=default
- [10] https://martinfowler.com/bliki/SacrificialArchitecture.html
- [11] Refactoring - Improving the Design of Existing Code by Martin Fowler
- [12] Clean Code – Robert Martin
- [13] Site du DDD (Eric Evans) : http://dddcommunity.org/
- [14] BDD in Action, edition Manning, John Ferguson Smart, 2014
- [15] Effective Java, Joshua Blosh
- [16] Harmcrest junit : https://github.com/junit-team/junit4/wiki/Matchers-and-assertthat

Références

- [17] Catalogue de patterns de refactoring (Martin Fowler) : https://refactoring.com/catalog/
- [18] Video de refactoring Red-green par Robert Martin : https://vimeo.com/43734265
- [19] Manning, « Dependency Injection », ed de mars 2019
- [20] « Get Your Hands Dirty on Clean Architecture » par Tom Hombergs
- [21] User Story Mapping (Jeff Patton)
- [22] RESTfull tutorial (bonnes pratiques REST)