

.Clean Code

*notes de cours
2016*

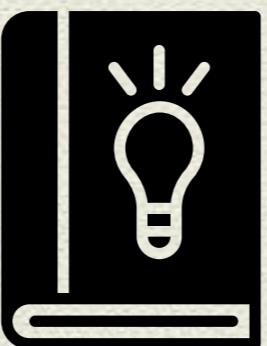


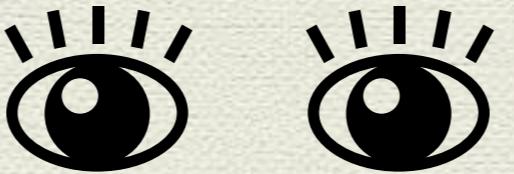
Rui Carvalho - cours@rui.fr - @rbwey

« Notre métier c'est d'apprendre en continu.

Mais pour pouvoir apprendre il faut déjà savoir
que l'on ne sait pas. »

-@rhwY





L'objectif de cet humble petit cours est donc de vous ouvrir les yeux et l'esprit pour que vous puissiez commencer à apprendre et de vous donner quelques clefs pour débuter votre aventure!

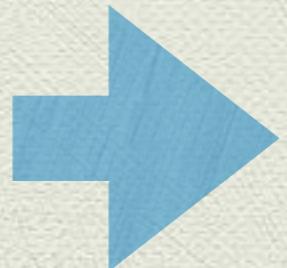




Fondamentaux

Pourquoi la qualité

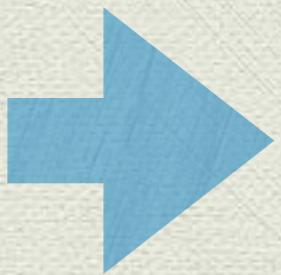
- ◆ Continuous attention to technical excellence and good design enhances agility
 - ◆ (9e principe du manifeste agile)



plus vous avez de qualité
plus vous êtes capables
d'être agiles

Pourquoi l'agilité?

- ◆ Responding to change over following a plan
 - ◆ (3e valeur du manifeste agile)

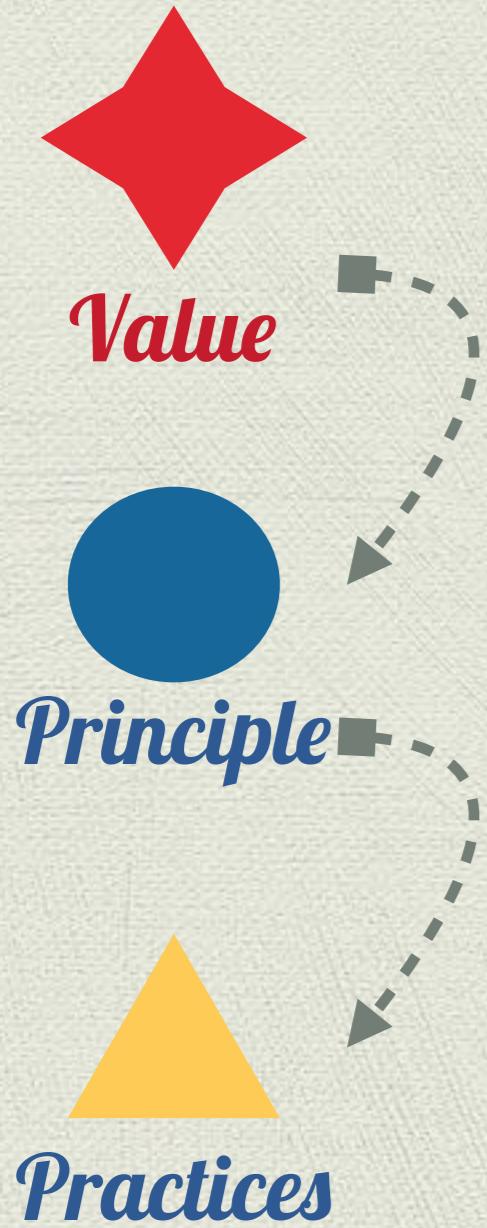


Nous voulons être agiles car nous savons que les grands plans ne fonctionnent pas bien

Pourquoi un code clean?

- ◆ Pour s'adapter au changement à moindre coût
- ◆ Pour pérenniser les produits
- ◆ Pour valoriser le travail d'un développeur
- ◆ Pour réconcilier progrès économique et social
 - ◆ (Kent Beck, extreme programming explained)

Perspective

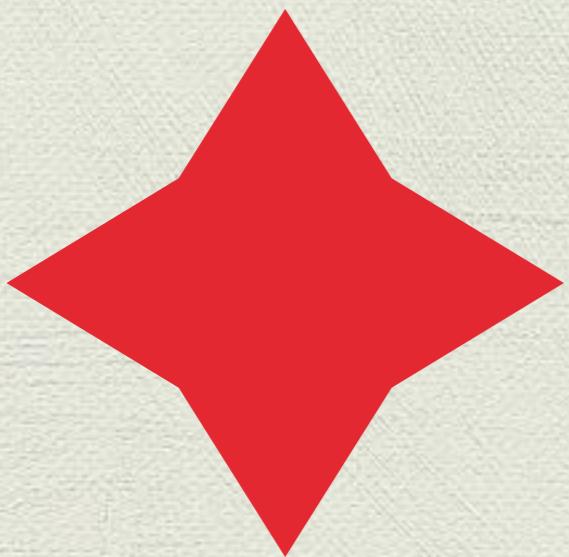


Responding to change over
following a plan

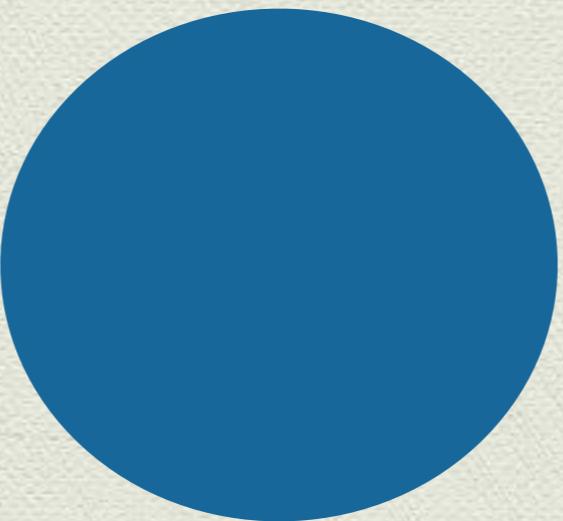
Continuous attention to technical
excellence and good design enhances
agility

Write Clean Code

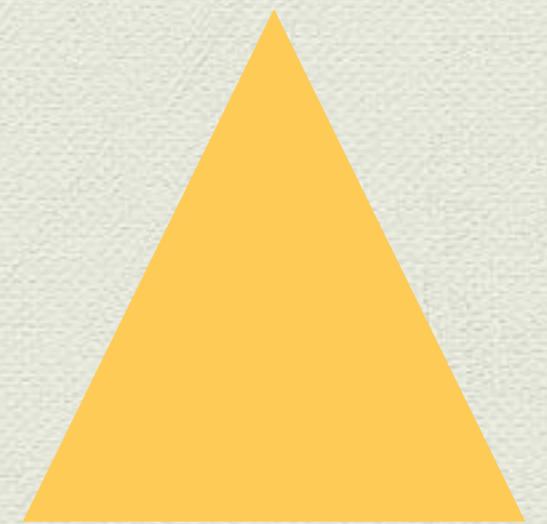
Fondamentaux



VALUES

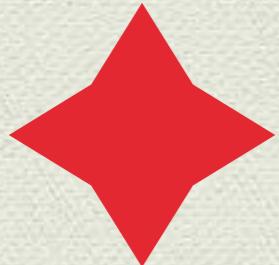


PRINCIPLES

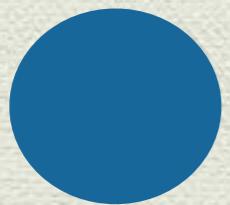


PRACTICES

Fondamentaux: exemple



Afin de répondre au *changement*



Nous apportons une attention *continue*
à l'*excellence technique*



Pour cela nous appliquons des *pratiques*
comme le **TDD** (*parmi d'autres*)

A titre personnel

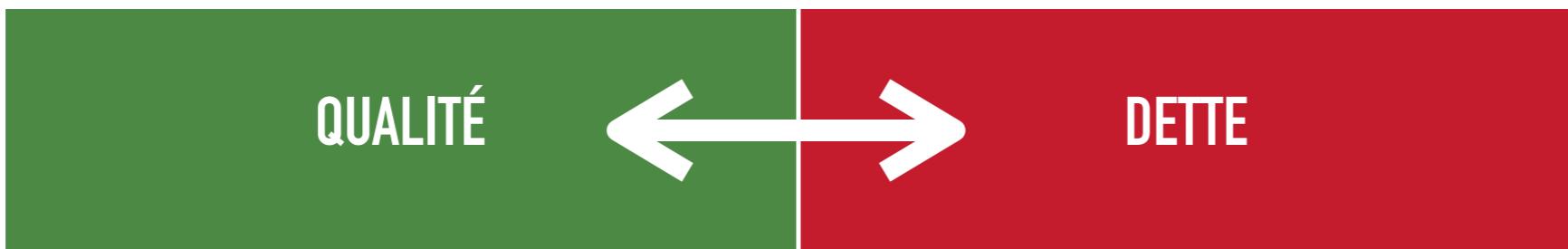


Toujours se poser la question du
Pourquoi en premier



Le comment ne doit intervenir qu'à la fin
Il est facile de trouver un outil ou une méthode
pour faire quelque chose si on comprends pourquoi
on doit le faire!

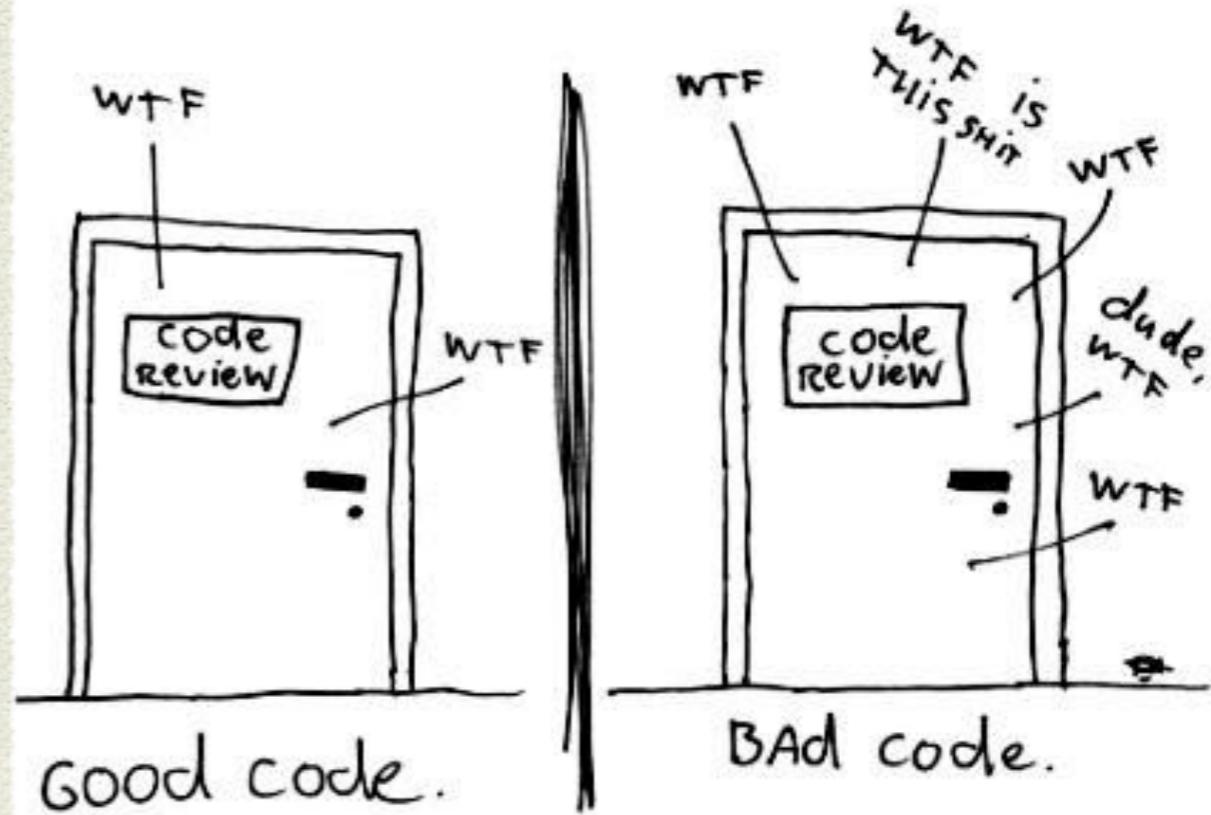
DETTE



Moins nous mettons de *qualité*
dans notre produit, **plus** nous
produisons de la *dette*

Good code and bad code

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



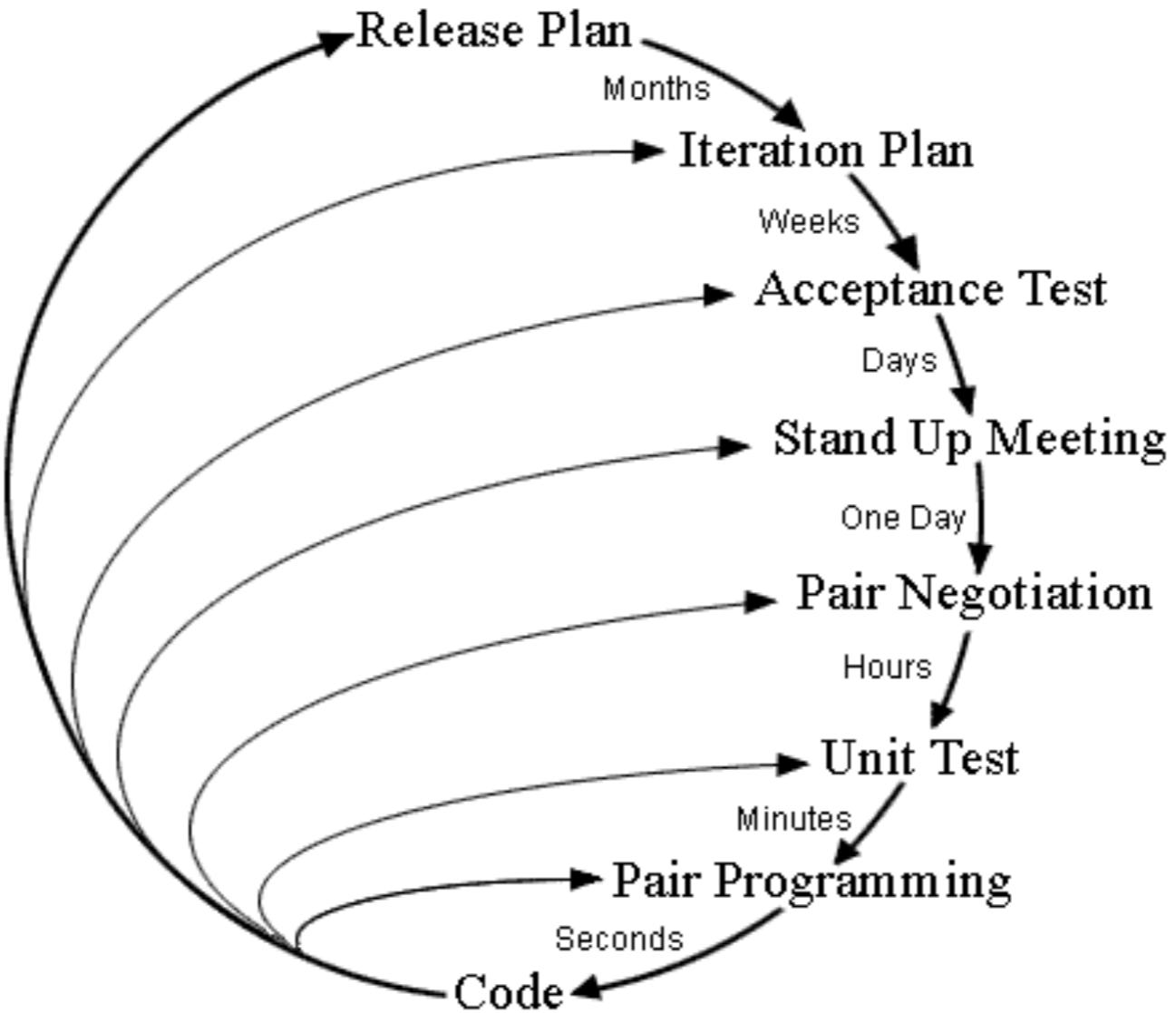
Un code plus clean, en gros c'est quoi ?

- ◆ Testé!
- ◆ Elégant et facile à lire
- ◆ Montre clairement l'intention de son auteur
 - ◆ Ecrit à destination des prochains développeurs
 - ◆ Il est difficile pour les bug de se cacher
- ◆ Simple
 - ◆ Non dupliqué
 - ◆ Contient un minimum d'artefacts
- ◆ Les dépendances sont minimales et explicites



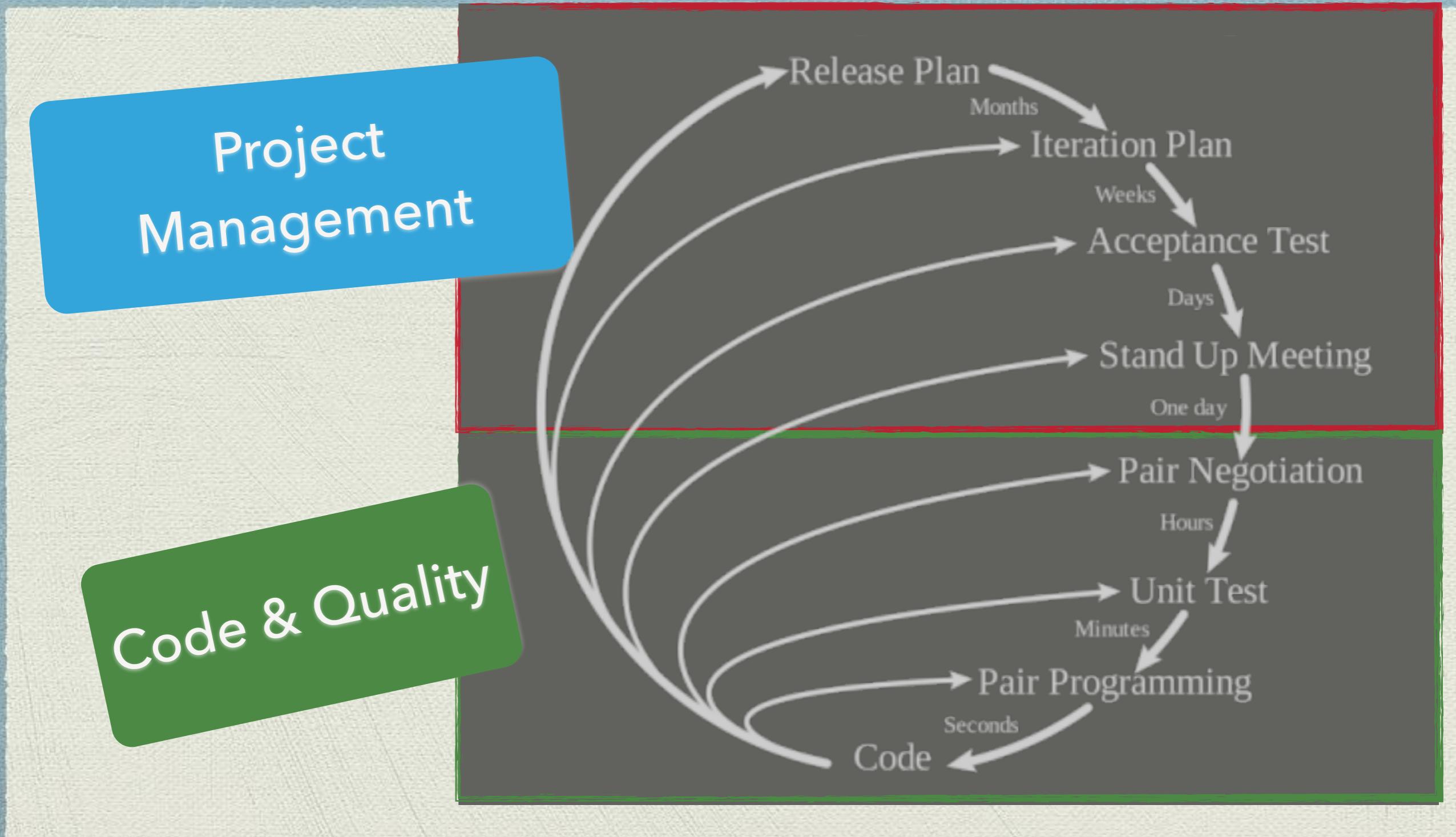
Valeurs

Planning/Feedback Loops



XP Feedback loops

Toutes les boucles de feedback sont nécessaires pour avoir de la qualité!





Pratiques

Les pratiques XP

- ◆ Client sur site
- ◆ Whole team
- ◆ Planning Game
 - ◆ Release planning
 - ◆ Sprint planning
 - ◆ Planning Poker
- ◆ Livraisons petites et fréquentes
 - ◆ Itérations
- ◆ Métaphores

Les pratiques XP

- ◆ Standards de développement
- ◆ Intégration continue
 - ◆ Déploiement continu
- ◆ Refactoring
- ◆ Design simple et incrémental
 - ◆ Couplé au Refactoring
 - ◆ Pas de Big Design Up Front
- ◆ Rythme soutenable

Les pratiques XP

- ◆ TDD
- ◆ Tests unitaires
- ◆ Tests d'acceptation
- ◆ Pair programming
- ◆ Collective Code Ownership
 - ◆ Courage / Respect
- ◆ Revues de code

Les pratiques complémentaires

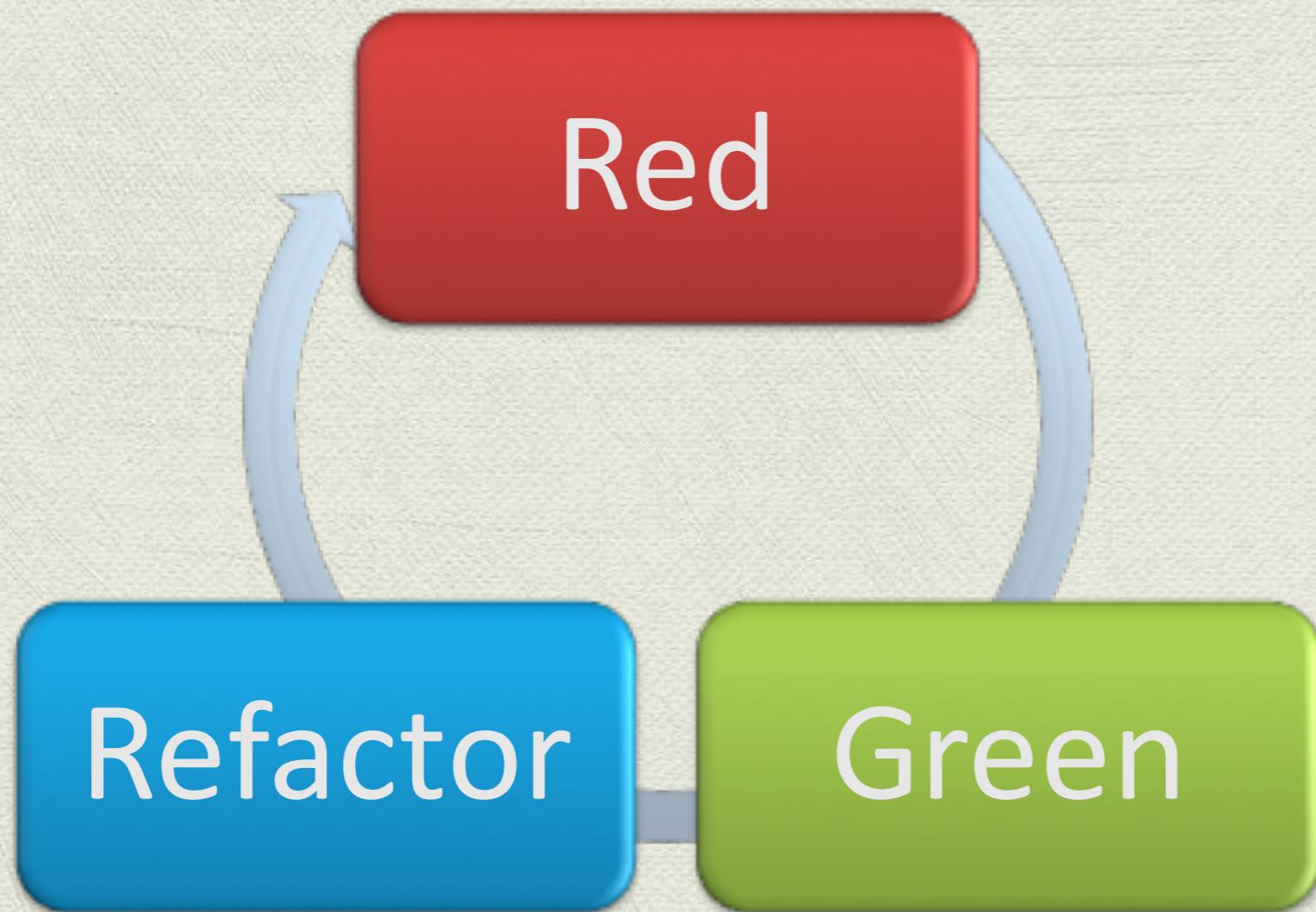
- ◆ User Stories
- ◆ ATDD / BDD
- ◆ Clean Code
- ◆ DDD
- ◆ Clean Architecture

TDD

- ▶ Discipline de conception
- ▶ Conception émergente
- ▶ Centrée sur le besoin
- ▶ Favorise et pousse au refactoring continu

TEST
DRIVEN
DESIGN

TDD, Aka : Red, Green, Refactor





Principes de développement

C'est mal ...

- ◆ Singleton
- ◆ Tight coupling
- ◆ Untestable
- ◆ Premature optimization
- ◆ Indescriptive naming
- ◆ Duplication

C'est bien

- ◆ SOLID
- ◆ YAGNI
- ◆ DRY
- ◆ KISS
- ◆ Moscow
- ◆ Loi de Demeter



SOLID

SOLID

- ◆ Single Responsibility Principle
- ◆ Open closed principle
- ◆ Liskov Substitution Principle
- ◆ Interface Segregation Principle
- ◆ Dependency Inversion Principle

SRP

- ◆ « *An object should have only a single responsibility* »
- ◆ « *A responsibility is a reason to change, and an object should only have one reason to change* ».
- ◆ ex : ma classe peut faire 2 choses qui vont ensemble comme lecture/écriture en base mais pas aller lire une base de données et dans un fichier
- ◆ Même sans changement, est-ce testable?

OCP

- ◆ “Software entities should be open for extension, but closed for modification”
- ◆ En clair:
 - ◆ On ne devrait pas être obligé de modifier une classe pour rajouter une fonctionnalité
 - ◆ On doit fournir les mécanismes d'extension à cela

LSP

- ◆ Liskov substitution principle
- ◆ « *Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program* »
- ◆ Je travaille donc avec des abstractions et peut utiliser n'importe laquelle des implémentations indifféremment.

ISP

- ◆ Interface segregation principle
- ◆ « *Many client specific interfaces are better than one general purpose interface* »
- ◆ Ne pas respecter ce principe c'est créer des dépendances arbitraires inutiles
 - ◆ (je me retrouve à être obligé d'implémenter des choses qui n'ont pas de sens pour moi)

DIP

- ◆ Dependency Inversion principle
- ◆ “Don’t call us, we call you”
- ◆ 1. High level modules should not depend on low-level modules.
Both should depend on abstractions
- ◆ 2. Abstractions should not depend upon details. Details should depend upon abstractions.
- ◆ Applicable par : Service Locator (mal), Factory (moyen),
Dependency Injection (bien)...
- ◆ Aidé par un IoC container (mais pas obligatoire)

Loi de Demeter

- ◆ Un objet ne devrait invoquer que les méthodes des types d'objets suivants:
 - ◆ Lui même
 - ◆ Ses paramètres
 - ◆ Les objets locaux
 - ◆ Ses sous objets
- ◆ “Don’t talk to strangers”

DRY

- ◆ Don't repeat yourself
- ◆ réutiliser un même code de plomberie ce n'est pas si grave si cela évite de rajouter une dépendance
- ◆ ce qui est mal c'est de dupliquer des idées
- ◆ ex: j'ai 2 classes utilisateurs dans mon app, je ne sais plus laquelle utiliser...

KISS

- ◆ Keep it simple stupid
- ◆ Keep it short and simple
- ◆ C'est l'objectif!
- ◆ On a une tendance naturelle à compliquer les choses!
- ◆ Si je relis du code que je viens d'écrire et que je me dit “ouaouh, c'est simple et clair！”, c'est sûrement que je viens d'écrire un très bon code (by greg Young)
- ◆ “Simplicity is the ultimate sophistication” (by Leonardo da Vinci)

YAGNI

- ◆ You ain't gonna need it
- ◆ Ne pas ajouter une fonctionnalité tant que vous n'en avez pas vraiment besoin!
- ◆ Créer le code le plus simple qui réponde au besoin
- ◆ (facilité par la pratique du TDD)

MoSCoW Method

- ◆ M – Must : absolument nécessaire pour que la solution soit réussie
- ◆ S – Should : haute priorité à faire si possible mais moins critique
- ◆ C – Could : A faire si il y a le temps
- ◆ W – Won't : pas nécessaire maintenant mais peut être valable pour le futur



A plus haut niveau

SDP

- ◆ Stable dependencies principle
- ◆ Les dépendances entre packages dans le design d'un logiciel doivent se faire dans la direction de la stabilité des derniers
- ◆ Un package ne devrait dépendre que de packages plus stables que lui

SAP

- ◆ Stable abstraction principle
- ◆ Plus un package doit être stable plus il doit être abstrait
- ◆ Les packages les plus instables doivent être concrets
- ◆ Le niveau d'abstraction d'un package doit être en relation avec son niveau de stabilité



Des règles

Nommage

- ◆ Les noms doivent révéler l'intention
- ◆ Ne tronquez pas les mots
 - ◆ (coût du stockage des char de nos jours ?...)
- ◆ Les noms doivent porter du sens
- ◆ Pas d'encodage
 - ◆ (pas lisible hors contexte)
- ◆ Pas de noms imprononçables
 - ◆ (*à la machine à café* : “*tiens au fait j'ai changé Mgm12v3()*”...)
- ◆ Pas de préfixes
- ◆ Evitez le Mapping mental (var x; / / user name)
- ◆ Le nom aussi peut changer lors d'un refactoring si il a perdu son sens

Abstractions

- ◆ Séparer le code métier du code technique
(log, cache, sécurité,...)
- ◆ Introduire de l'aop pour les préoccupations transverses

Couplage fort

- ◆ Comment créez vous du couplage fort?
 - ◆ New qqch()
 - ◆ Appels statiques
 - ◆ Utilisation directe de classes concrètes

- ◆ Comment l'éviter?
 - ◆ Utiliser des méthodes dédiées à la création des instances
 - ◆ Injection des dépendances
 - ◆ Utilisation de types abstraits à chaque fois que possible

Méthodes longues

- ◆ Qu'est-ce qu'une méthode longue?
- ◆ Refactorez si trop long.
- ◆ Long method refactoring comment :
 - ◆ Ajoutez des commentaires sur tous les éléments de votre méthode
 - ◆ Renommez tout dedans en fonction des commentaires
 - ◆ Déplacez certaines variables pour créer des blocs logiques
 - ◆ Extraire ces blocs dans des méthodes séparées en fonction des commentaires
- ◆ Refactorez ces méthodes si besoin pour de la duplication de code par exemple
- ◆ Supprimez les commentaires (le code est clean, il ne sont plus nécessaires ;-)

Commentaires

- ◆ “Comments are always failures”
- ◆ Il mentent souvent et vieillissent très mal(refactoring,...)
- ◆ Aveu de faiblesse (j’explique ce que je n’ai pu coder proprement)
- ◆ Sauf si:
 - ◆ Donne une explication du contexte (jamais explicite dans le code)
 - ◆ Aide à fournir de la documentation automatique

Exceptions

- ◆ Fail Fast!
- ◆ La défense : Avant d'avoir besoin d'exceptions assurez vous d'avoir levé toutes les assertions nécessaires
 - ◆ (ex: pas de DivideByZero alors que vous savez que vous devez vérifier l'égalité avec 0 de cette variable)...
- ◆ L'attaque :
 - ◆ Lever les exceptions (techniques) dès que nécessaire
 - ◆ Ne pas masquer les exceptions (se retrouver avec un null pointer car on se rend compte trop tard du problème)

Exceptions

Dans les langages on appelle ça **Exceptions**, pas erreurs, c'est justement pour bien exprimer le caractère **exceptionnel** de la situation

Ne pas utiliser les exceptions comme contrôle de flux

Tests

- ◆ Garants de votre refactoring!
- ◆ Le code des tests doit lui aussi être clean
- ◆ Lisibilité :
 - ◆ un test doit présenter un seul concept et
 - ◆ un regroupement logique de tests doit raconter une petite histoire
- ◆ Suivez une logique commune et explicite
 - ◆ AAA(Arange, Act, Assert)
 - ◆ Given/When/Then

What else?



Pour aller plus loin maintenant...

- ◆ Pratiquez, pratiquez, pratiquez.
 - ◆ on devient forgeron en forgeant et pareillement on devient développeur en développant, livrant, maintenant du code.
- ◆ Changer les choses autour de vous un petit bout à la fois.

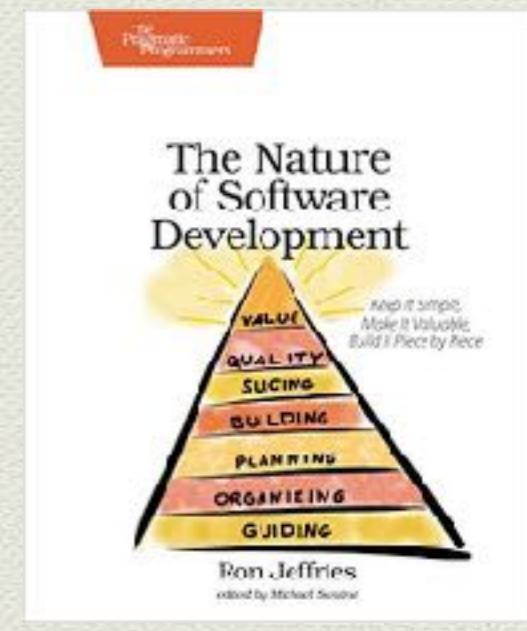
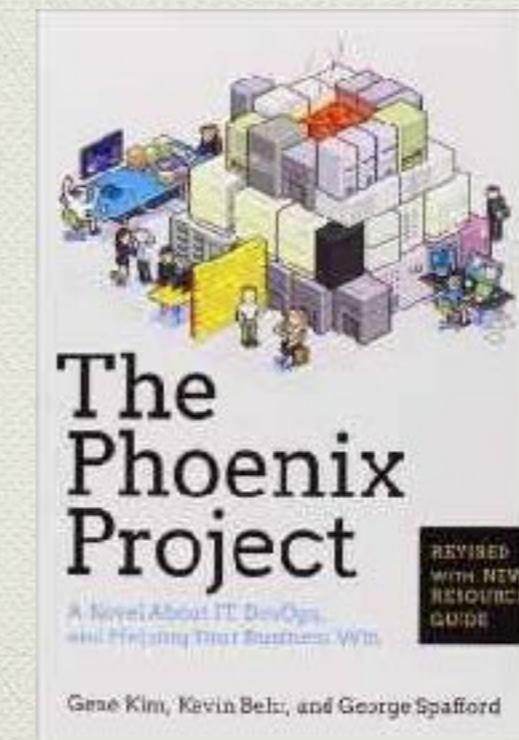
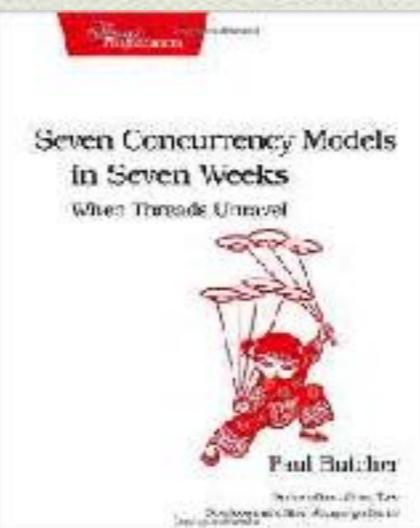
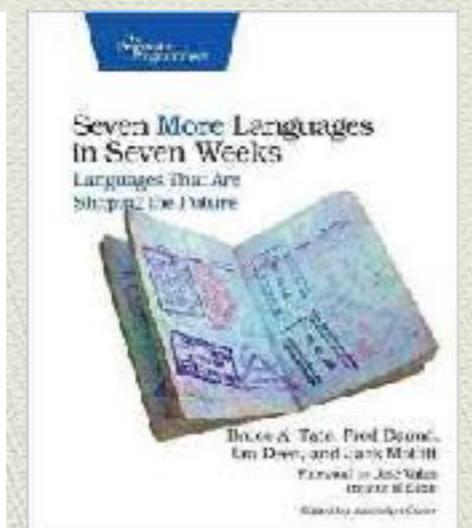
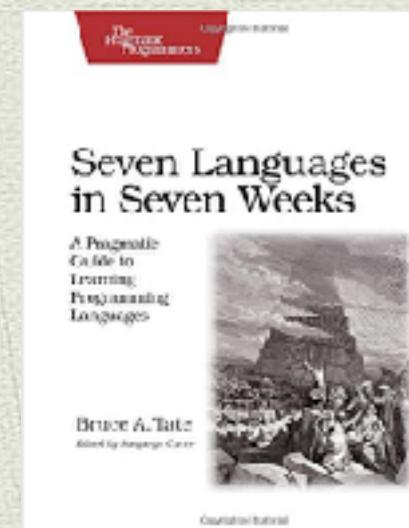
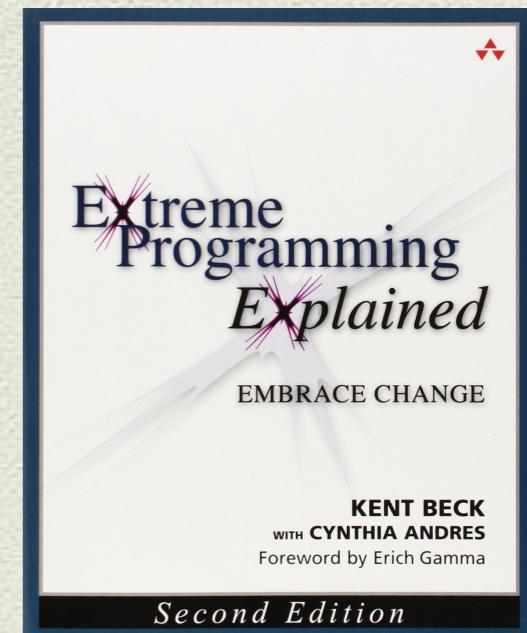
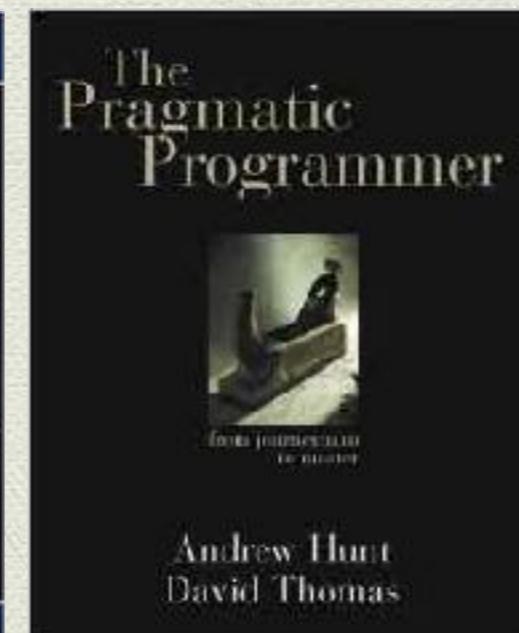
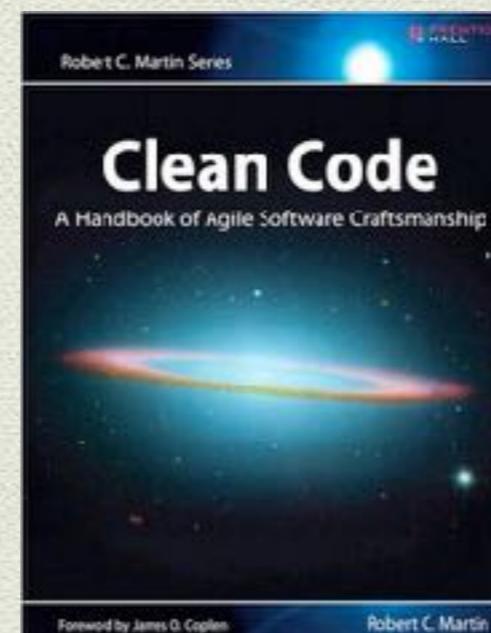
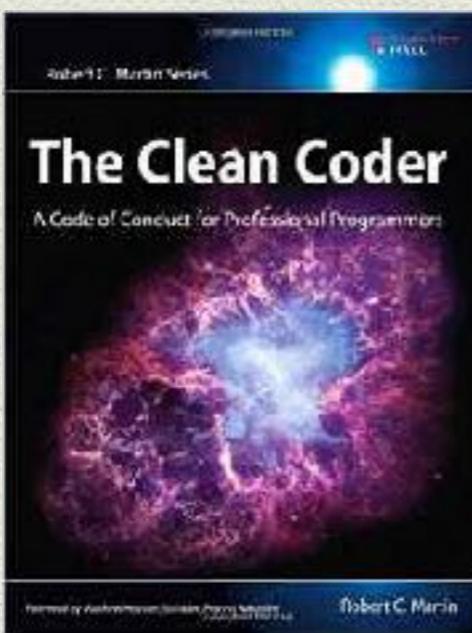
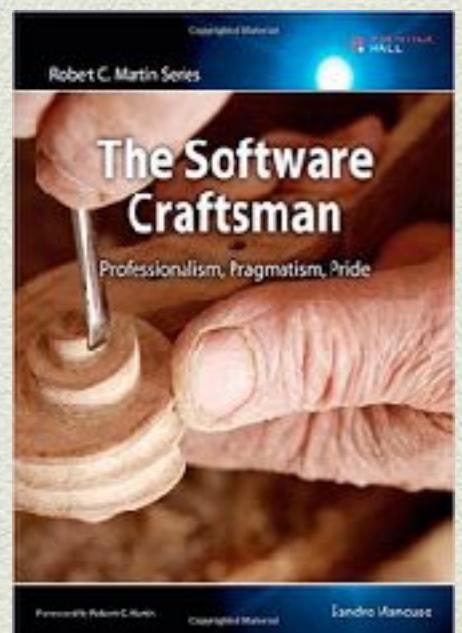
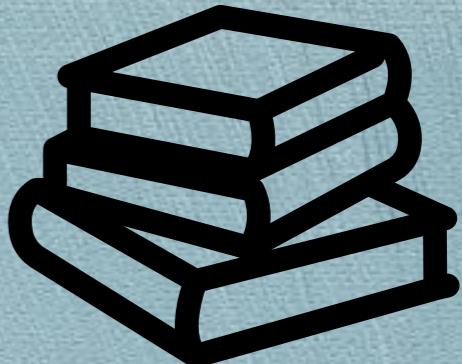
Bougez!

- ◆ Sortez de votre zone de confort, c'est la seule manière d'apprendre quelque chose de nouveau
- ◆ Allez à des meetups qui vous intéressent! c'est le meilleur moyen de rencontrer des gens plus intéressants que vos collègues et qui partagent les mêmes centres d'intérêts que vous ;-)
- ◆ Contribuez à des projets open source, c'est une des meilleures écoles!
- ◆ Allez chaque fois que possible voir les gens en vrai dans vos bureaux

Practice!

- Original coding dojo resources: <http://codingdojo.org>
- Apprendre des langages par des exercices: <http://exercism.io/languages>
- Des pb d'algo très prenants : <http://adventofcode.com/2016>
- Des pb pour manipuler des structures à base de génétique : <http://rosalind.info/problems/list-view/>
- Des pb d'algo: <http://rosalind.info/problems/list-view/?location=algorithmic-heights>
- Des Koans C# (comment apprendre un langage par les tests): <https://github.com/rhwy/CsharpNunitKoans/tree/master/CsharpNunitKoans>
- Quelques Katas supplémentaires des dojos alt.Net : <https://github.com/dthouvenin/alt-net-fr-katas>

Books



Keep in touch

N'hésitez pas à me contacter!

- ◆ cours@rui.fr
- ◆ @rhwy sur twitter+github

NEW CRAFTS



2017 MAY 18-19



A Software Craftsmanship Conference

La conférence des professionnels du développement logiciel de l'année à ne pas manquer.

Parlez-en autour de vous !