la Mobilière

Clean Code

Olivier von Dach, 18.9.2019

la Mobilière

Clean Code

Pourquoi?
Quoi?
Comment?
Qui?

Le code **logiciel**

Le code, un pilier de l'entreprise moderne

Le code logiciel **automatise** les processus de « la maison ».

Le code détient la vérité

Le code est la dernière source de **documentation**.

Le code est en mouvement

Il doit être régulièrement **adapté**, **maintenu**, **étendu**

Il doit pouvoir réagir au **changement**

Le code,
un atout ou une charge

Pour une équipe, pour une entreprise

Un **actif** ou une **dette**, voire les deux

Une base durable **d'opportunités** ou un centre de **coûts**

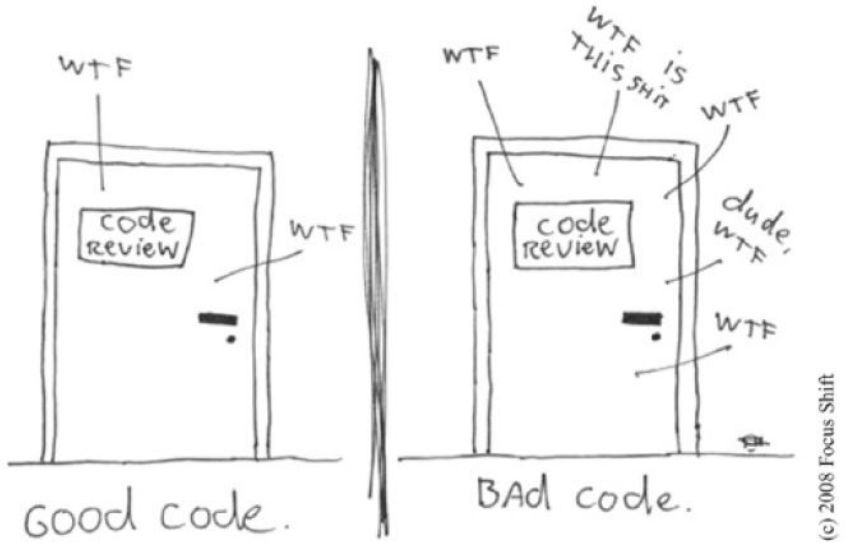
L'aspect social du code

la Mobilière

github SOCIAL CODING

Communautaire, partenarial, et émotionnel

IT-BA-Métier



Reproduced with the kind permission of Thom Holwerda.
http://www.osnews.com/story/19266/WTFs_m

Comment bien vivre le code?

Pratiquement,

socialement,

émotionnellement,

économiquement?

Optimiser le temps de **rédaction,**

développer plus **sereinement,**

assurer une **productivité constante**?

Développer autrement

~~Ne pas développer pour simplement livrer~~,

mais développer pour **durer**.

On parle de code

Et non de l’auteur·e d’un code

Le code logiciel appartient à **l’entreprise**

La **qualité** du **produit** avant tout

Du code *bien conçu* et de l'interactivité

Quelques **principes** de **rédaction** de code

Quelques **principes** de **design** logiciel

Réfléchir avant de coder, **partager** la réflexion, demander le **feedback**

Penser « communautaire »

Quoi

«Clean code can be read, and enhanced by a developer other than its original author.»

*It has unit and acceptance **tests**.*

*It has meaningful **names**.*

*It has **minimal dependencies**, which are explicitly defined, and provides a clear and **minimal API**.*

***Code should be literate** since depending on the language, not all necessary information can be expressed clearly in code alone.”*

Robert C. Martin

Écrire du code lisible

Être **explicite**

Montrer les **intentions**

Qu'est-ce que je veux résoudre?

Qu'est-ce que je veux dire?

Pourquoi je veux dire cela?

Quelle responsabilité veux-je donner?

J'appelle un chat « un chat »

Je raconte une histoire

Écrire du code lisible

Conventions de nommage

- Nom ou groupe de noms significatif
- Éviter les noms trop généraux
- Eviter les noms trop longs

Méthodes entre 5 et 10 lignes

Standardiser le vocabulaire

Pair-programming

Demander le challenge

Demander la revue

Demander le coaching

Parlons le même langage

Vocabulaire **Métier**

Objets métier

Evénements métier

Processus métier

Règles métier

Vocabulaire technique de **Patterns**

Chaîne de responsabilités, Observer, Proxy, Adapter, Template Method, Factory, Flyweight, Visitor, State

Vocabulaire technique de **Suffixes**

Filter, Converter, Matcher, Calculator, Extractor, Splitter, Handler, Service, Controller, Client, Writer, Reader. Listener

Une scène vivante

Le code devrait raconter l'histoire du **métier**

Les **scénarios** sont ceux décrits par le métier

Les **acteurs**

- des objets **métier** et des composants logiciels **techniques**
- supportés par des **interfaces**, des **classes** concrètes, des **instance** de classe, des **méthodes**, qui interagissent.

Soyez des metteurs en scène et donnez de la vie !

CasPrestationService.java

```java
public CasPrestation constituerCasPrestation(IdentifiantLese lese, LocalDate dateEvenement, TypeEvenement typeEvenement) {
  CasPrestation casPrestation = obtenirCasPrestationPourLese(lese);
  EvenementPrestation evenementPrestation = EvenementPrestation.of(dateEvenement, typeEvenement);
  casPrestation.deposerEvenement(evenementPrestation);
  Gestionnaire responsable = determinerGestionnaireResponsable();
  casPrestation.attribuerResponsable(responsable);
  persisterCasPrestation(casPrestation);
}

private CasPrestation obtenirCasPrestationPourLese(IdentifiantLese lese) {…

private Optional<CasPrestation> rechercherCasPrestationPourLese(IdentifiantLese lese) {…

private CasPrestation creerCasPrestationPourLese(IdentifiantLese lese) {…

private IdentifiantCasPrestation genererIdentifiantCasPrestation() {…

private void persisterCasPrestation(CasPrestation casPrestation) {…

private Gestionnaire determinerGestionnaireResponsable() {…

private IdentifiantUtilisateur recupererUtilisateurConnecte() {…
```

CasPrestationService.java

```java
private CasPrestation obtenirCasPrestationPourLese(IdentifiantLese lese) {
  return rechercherCasPrestationPourLese(lese).orElseGet(() -> creerCasPrestationPourLese(lese));
}

private Optional<CasPrestation> rechercherCasPrestationPourLese(IdentifiantLese lese) {
  return casPrestationRepository.rechercherParLese(lese);
}

private CasPrestation creerCasPrestationPourLese(IdentifiantLese lese) {
  IdentifiantCasPrestation identifiantCas = genererIdentifiantCasPrestation();
  return CasPrestation.of(identifiantCas, lese);
}

private IdentifiantCasPrestation genererIdentifiantCasPrestation() {
  return identifiantFactory.creerPourCasPrestation();
}

private void persisterCasPrestation(CasPrestation casPrestation) {
  casPrestationRepository.persister(casPrestation);
  domainContextProvider.get().publier(CasPrestationConstitue.of(casPrestation.getIdentifiant()));
}

private Gestionnaire determinerGestionnaireResponsable() {
  return Gestionnaire.of(recupererUtilisateurConnecte());
}

private IdentifiantUtilisateur recupererUtilisateurConnecte() {
  return IdentifiantUtilisateur.of(callContextProvider.get().getUser());
}
```

Écrire du code évolutif

Agiliser le code, ou optimiser l'**inertie** du code au changement

Une **dépendance** entre composants induit un **couplage**
Le couplage engendre de l'inertie

Le couplage engendre de la complication

La **complication** engendre de l'inertie

La **sur-responsabilisation** engendre du couplage

Écrire du code évolutif

De l'**agilité** dans l'organisation du code:

- Répartir les comportements par **responsabilisation**
- Intégrer le changement par **extension**
- Limiter les impacts du changement par **isolation**

OOD pour une scène vivante **organisée** et **orchestrée**

Responsabiliser de **petits** composants

Écrire du code évolutif

Réduire les responsabilités, et donc les **effets** du changement

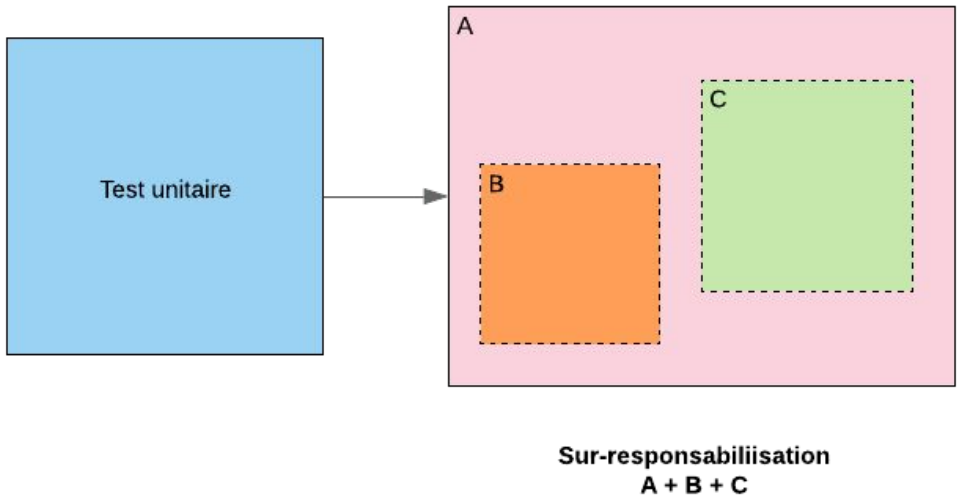
Isoler les parties stables: les **concepts**

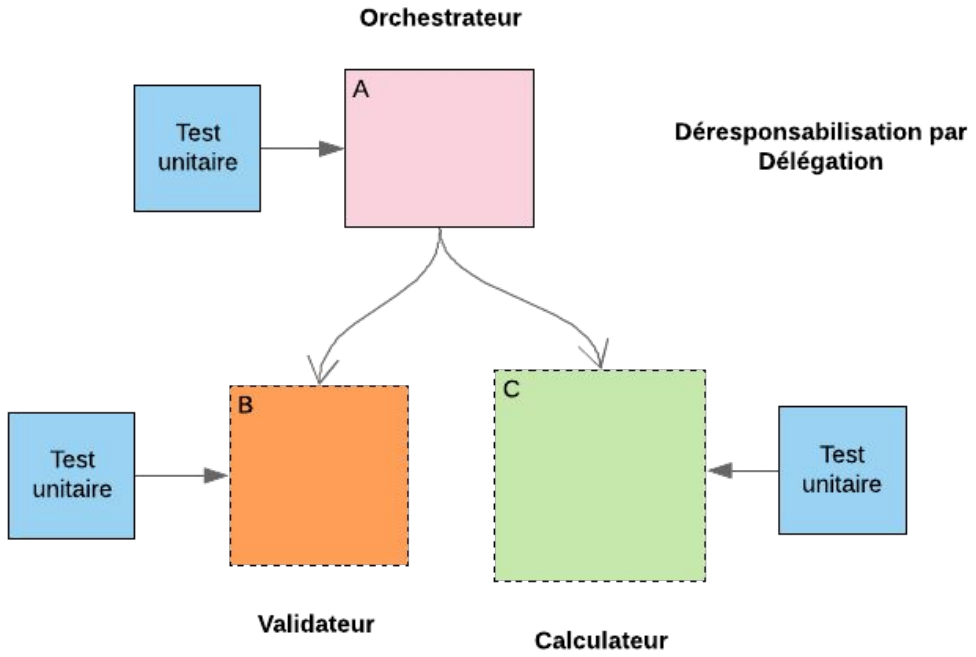
Encapsuler les parties variables: les **implémentations** concrètes

Conjuguer les principes **SOLID** et les Design **Patterns**

SRP «A class should have one, and only one, reason to change.»

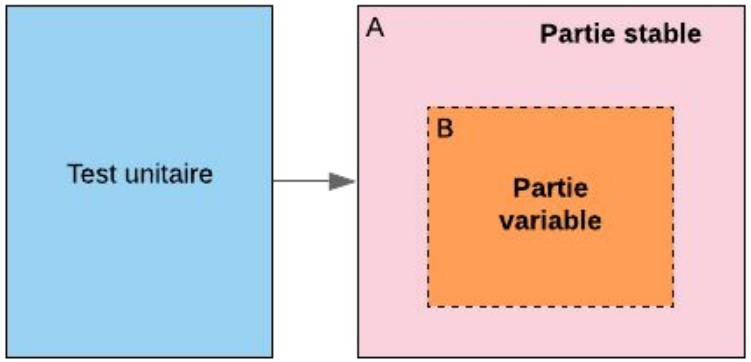
Comment





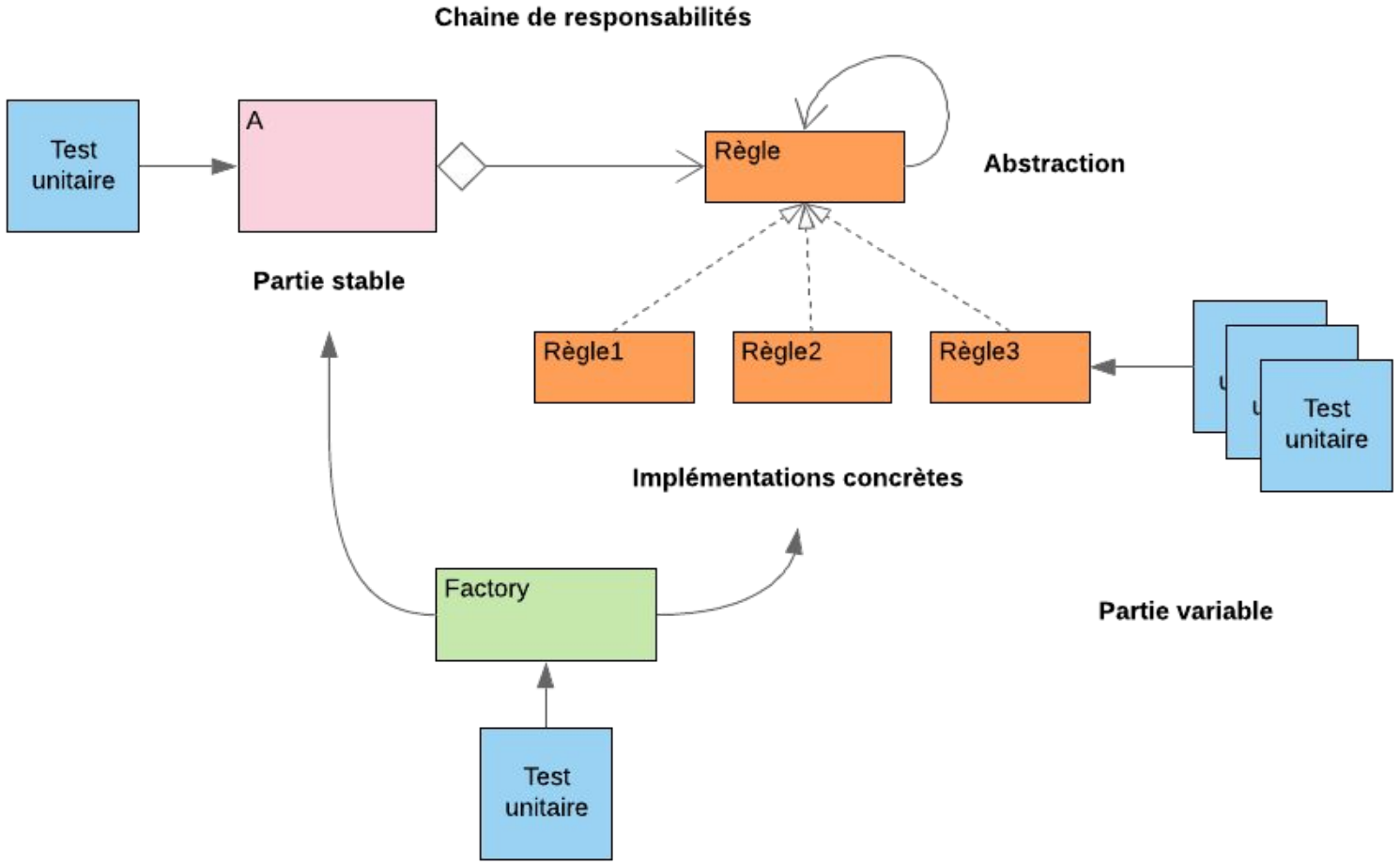
OCP «Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.»

Comment




```
switch (value) {
case RED: doAction1();
case BLUE: doAction2();
case GREEN: doAction2();
}

if (value % 2 == 0) {
      doAction1();
} elseif (2 < value && value < 9) {
      doAction2();
} elseif (value % 11 == 0) {
      doAction3();
}
```



Écrire du code testable

Un **test unitaire** est un **indicateur** et un **guide** de qualité

Quelle est la responsabilité de mon composant?

Quel est le comportement de mon composant?

Est-ce facile de tester mon composant?

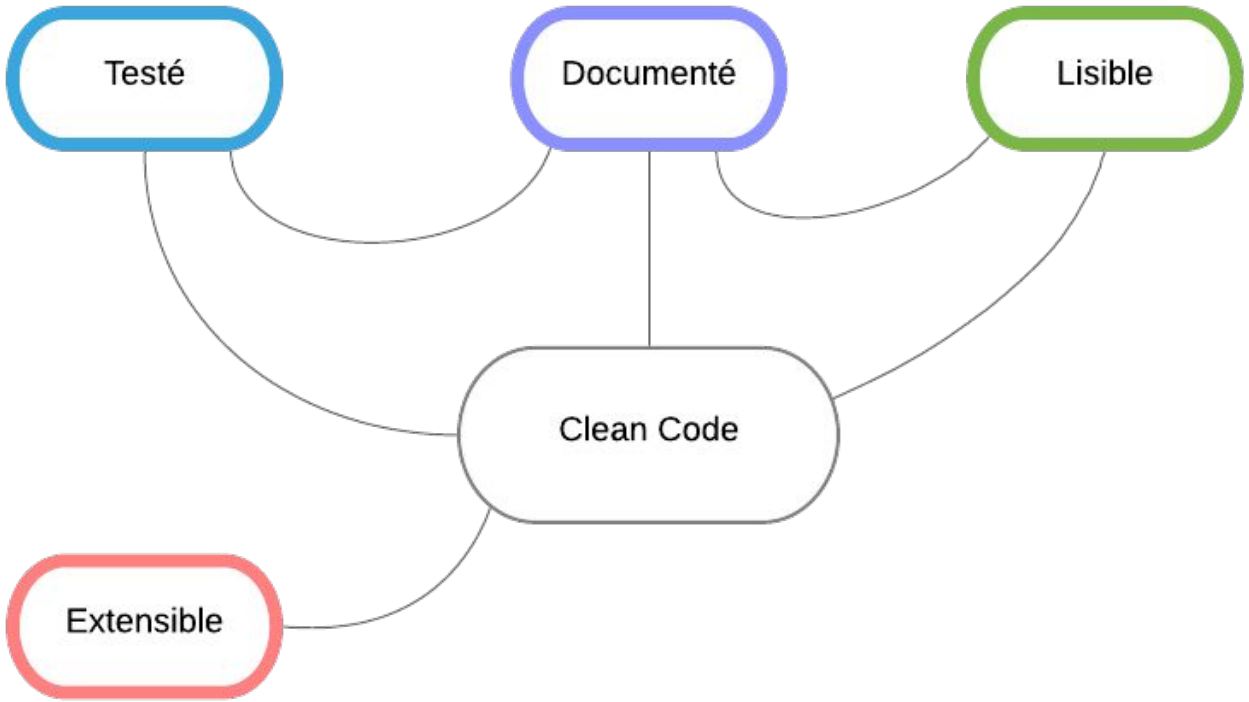
Combien de Mocks (dépendances)? Combien d'assertions?

Entretenir le code

Maîtriser la croissance de la base de code ou **rembourser** *la dette de code*

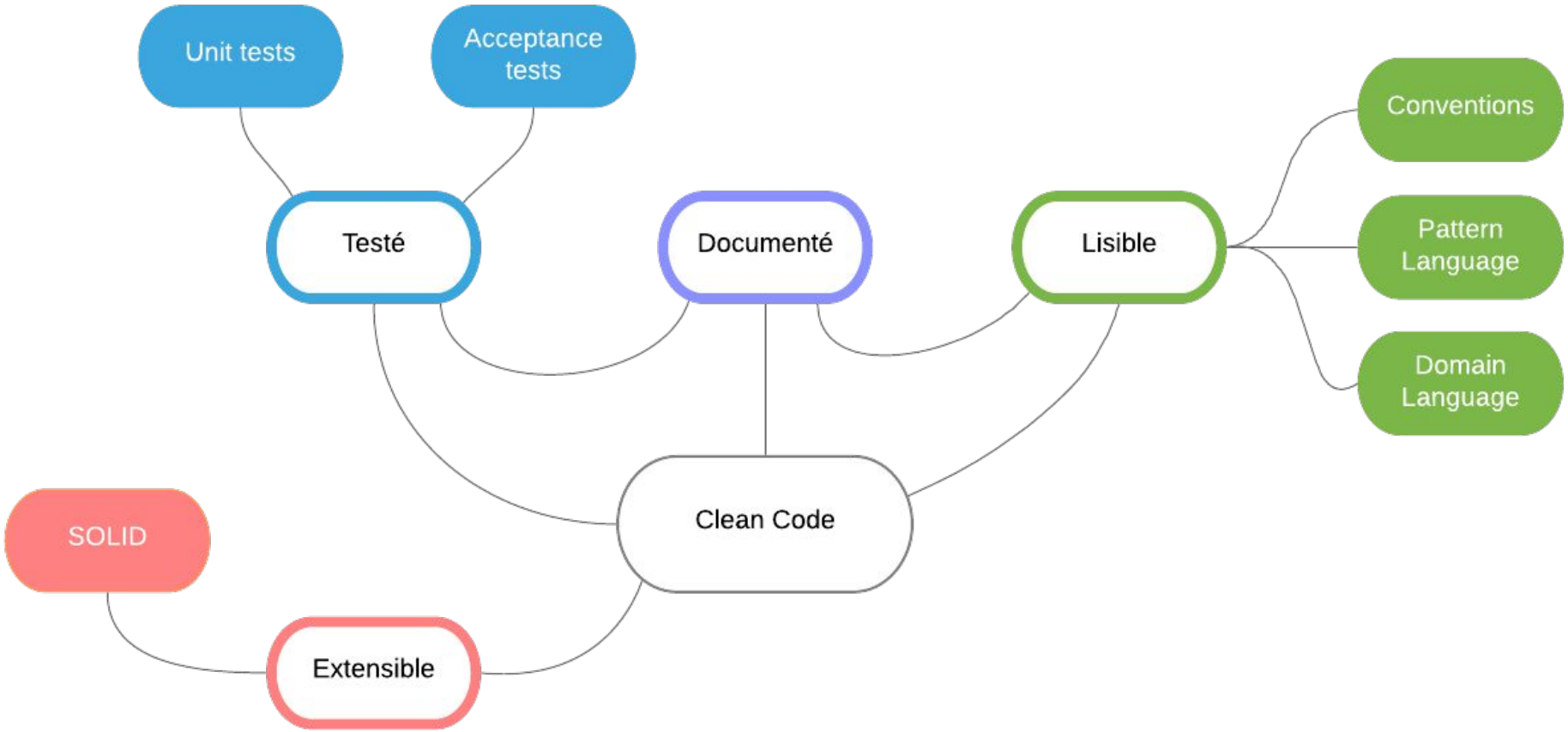
- **Surveiller** les composants
- **Réorganiser** régulièrement
- **Assainir** régulièrement

Résumé



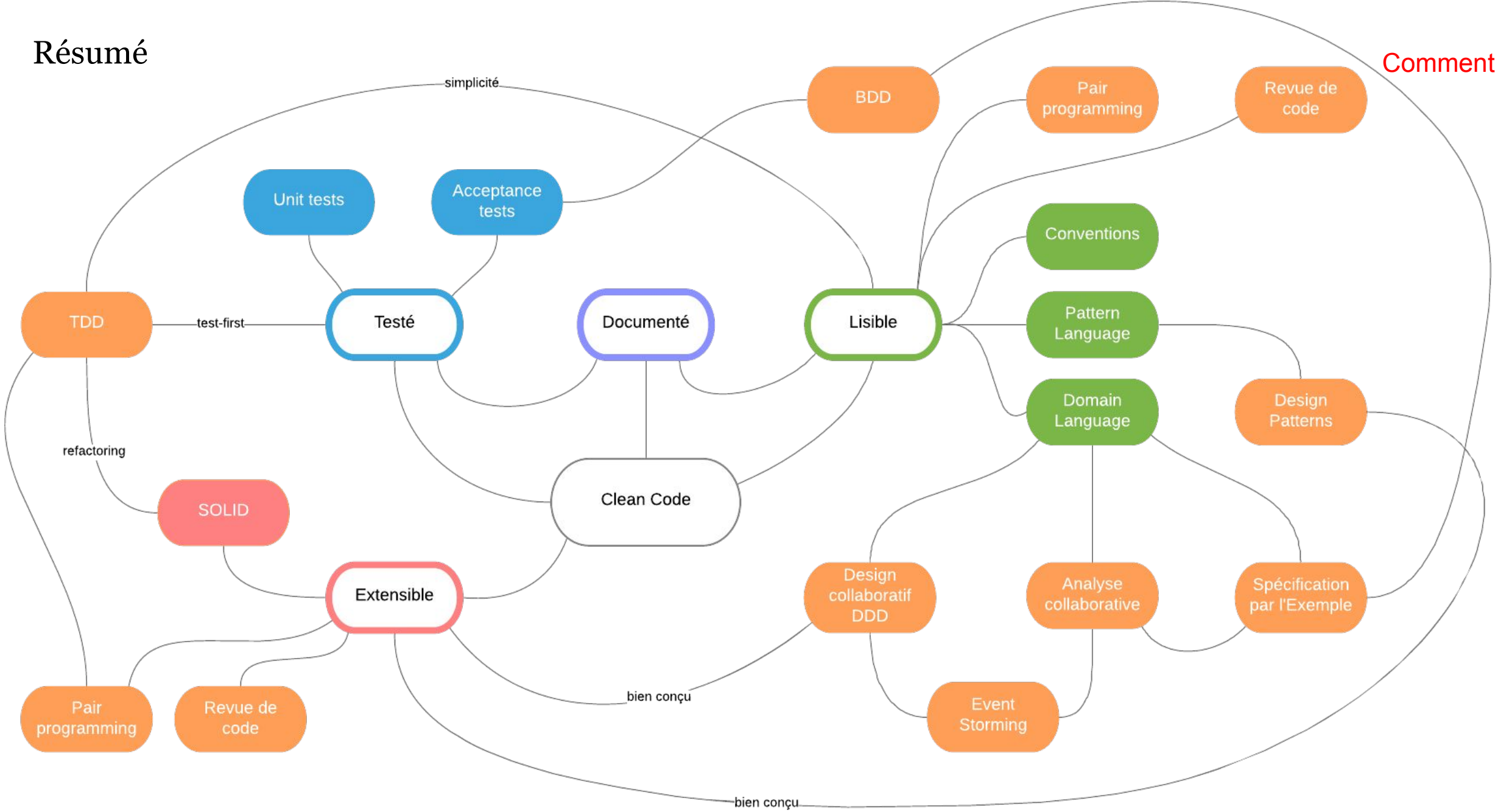
Résumé

Comment



Résumé

Comment



La suite?

Y penser sérieusement

Essayer le Clean Code

Pratiquer le Clean Code

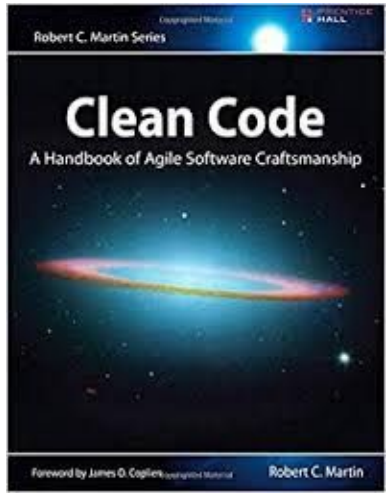
Montrer « son » code

Demander le challenge, la revue de code

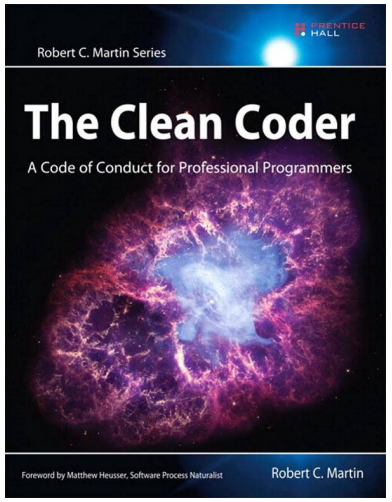
Se former

Ouvrages de référence

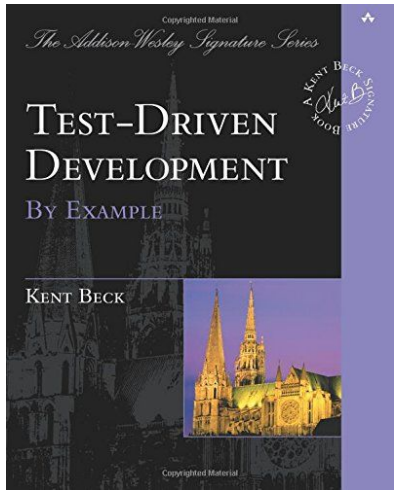
Comment



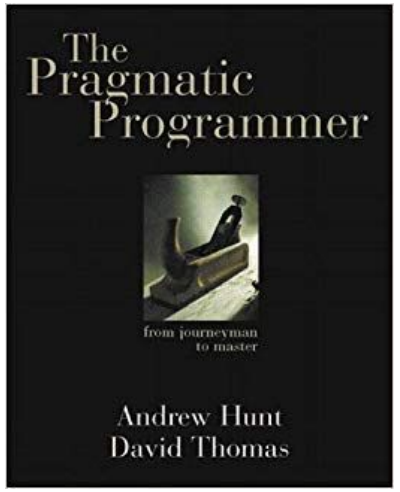
Robert C. Martin
2008



Robert C. Martin
2011



Kent Beck
2000



Andy Hunt
Dave Thomas
1999

https://refactoring.guru/design-patterns/book