Clean Code

Sébastien Jean

IUT de Valence
Département Informatique

v3.0, 18 septembre 2024

IUT Valence
Université Grenoble Alpes

Du logiciel qui marche ou du logiciel qui dure ?

- Ecrire du logiciel qui marche est à la portée du premier venu ;-)
- **Mais**
  - Le logiciel doit pouvoir évoluer sur le long terme
  - Il n'y a pas de pérennité sans qualité !

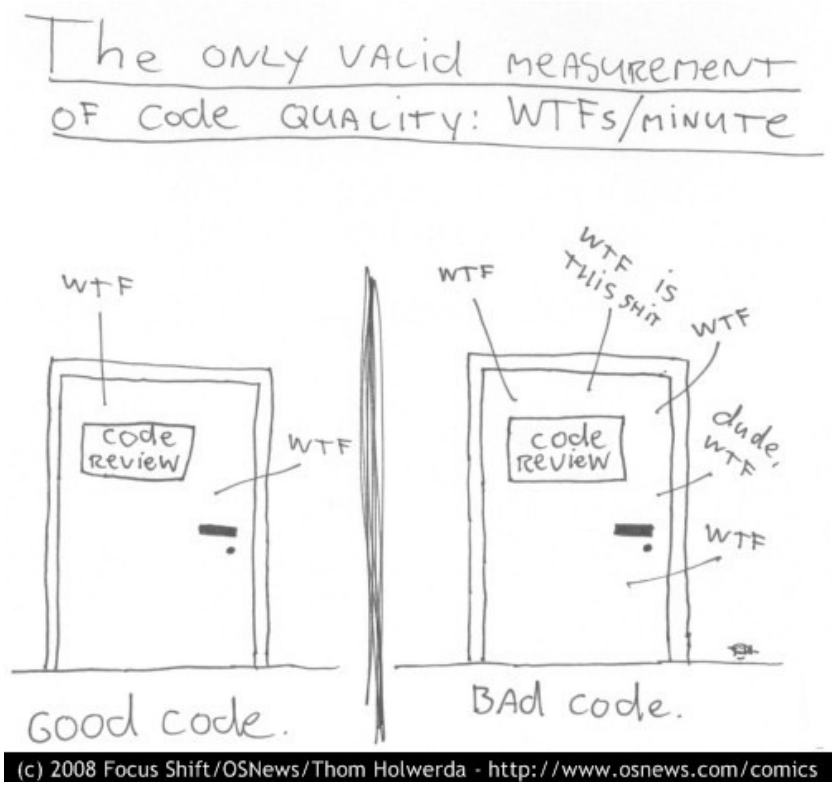
1981

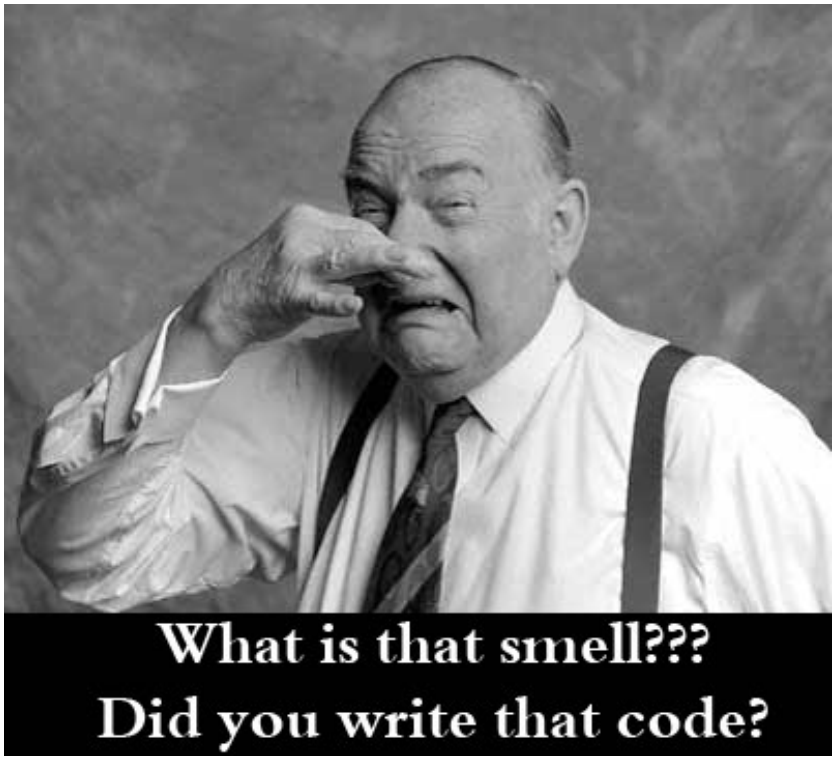
2024

IUT Valence
Université Grenoble Alpes

Qu'est ce qu'un code de qualité ?

- **Conforme aux exigences** et **exempt de bug**
- **Lisible** et **facile à comprendre**, **facile à utiliser**
- **Aisément maintenable** et **réutilisable**
- **Efficace**





IUT Valence
Université Grenoble Alpes

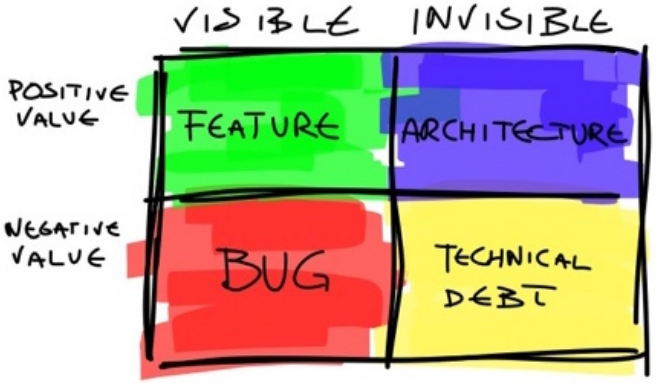
Entropie logicielle et dette technique

- **Entropie logicielle** (Lehman, 1980)

> "As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it."

- **Dette technique** (Cunningham, 1992)

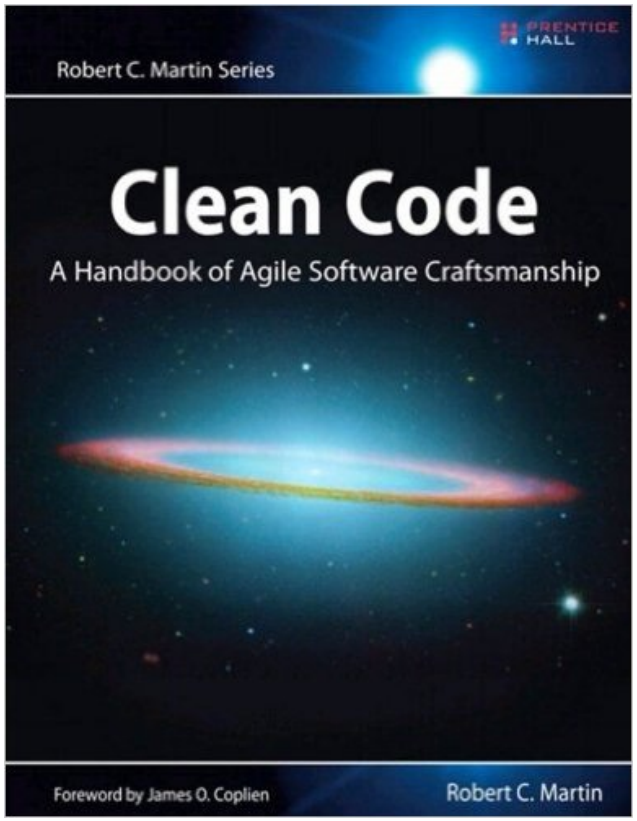
> "Shipping first time code is like going into **debt**. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as **interest** on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."

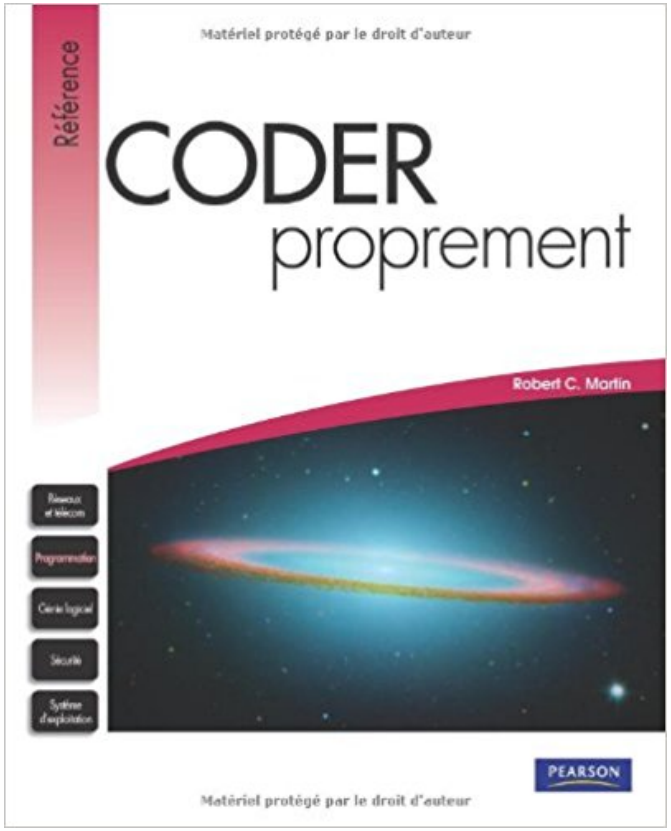


IUT Valence
Université Grenoble Alpes

Clean Code

- Ouvrage de référence





- voir aussi *Refactoring Guru* (`https://refactoring.guru`)

IUT Valence
Université Grenoble Alpes

Clean Code

Citation (extraite de *Clean Code*)

Clean code always looks like it was written by someone who cares.

Michaël Feathers, auteur de *Working with Legacy Code*

Citation (extraite de *Clean Code*)

You know you are working with clean code when each routine you read turns out to be pretty much what you expected.

Ward Cunningham, inventeur des wikis, co-inventeur de l'*Extreme Programming*

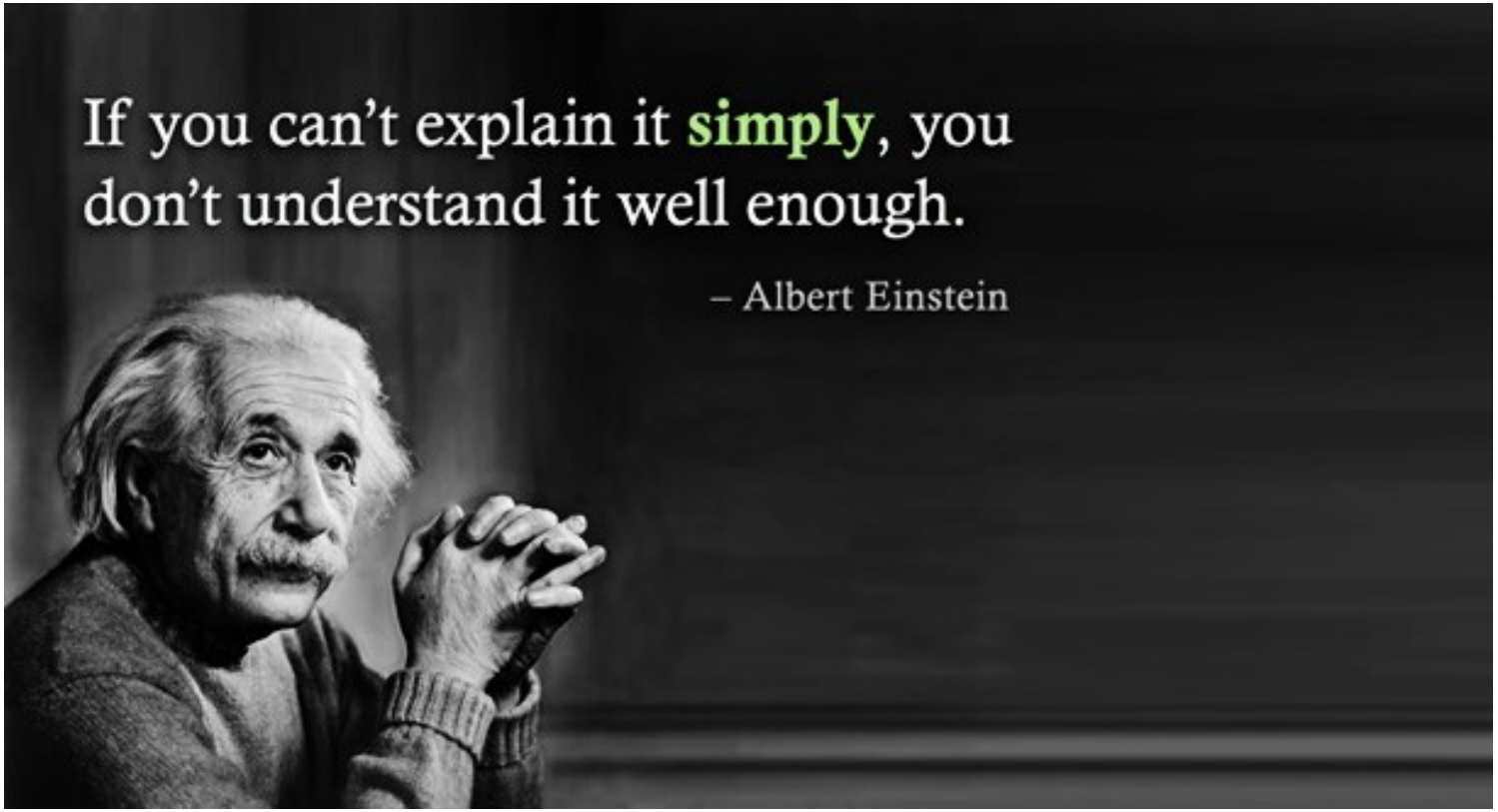
Ce qui contribue à un code propre

- Le **nommage intentionnel**
- Le bon usage des **commentaires**
- La **mise en forme**
- des **bonnes pratiques de conception/programmation**

IUT Valence Université Grenoble Alpes

Bonnes pratiques (bon sens)

- **K**eep **I**t **S**imple, **S**tupid $\leftrightarrow$ **K**eep **I**t **S**mart, **S**imple



IUT Valence Université Grenoble Alpes

Bonnes pratiques (bon sens)

- **D**on't **R**epeat **Y**ourself $\leftrightarrow$ **D**uplication **I**s **E**vil

Citation (extraite de *The Pragmatic Programmer*)

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Andy Hunt / Dave Thomas

IUT Valence Université Grenoble Alpes

Bonnes pratiques (bon sens)

- **Y**ou **A**in't **G**onna **N**eed **I**t

Citation

Always implement things when you actually need them, never when you just foresee that you need them.

Ron Jeffries

Nommage

- Choisir des noms
  - **révélateurs des intentions**
  - **prononçables**, **compatibles avec une recherche**
  - **dans le contexte** (de la solution, du problème)
- Définir des constantes dont le nom véhicule l'intention

IUT Valence
Université Grenoble Alpes

Nommage

Mauvaise idée

```
public List<int[]> getList () {
  List<int[]> l2 = new ArrayList<int[]>();

  for (int[] x : this.l1)
    if (x[0] == 4) l2.add(x);

  return l2;
}
```

Nommage

Meilleure idée

```
public List<int[]> getDeadCells () {
  List<int[]> deadCells = new ArrayList<int[]>();

  for (int[] cell : this.cells)
    if (cell[STATUS_OFFSET] == DEAD) deadCells.add(cell);

  return deadCells;
}
```

IUT Valence Université Grenoble Alpes

Commentaires

- **Se limiter à :**
  - Expliquer l'intention, l'implicite
  - Avertir des prérequis et conséquences

- **Ne jamais :**
  - Ecrire des commentaires redondants
    - avec le sens véhiculé explicitement les déclarations ou l'implémentation
    - avec d'autres vecteurs d'information (licences, *versioning*, ...)
  - Mettre du code en commentaire
    - c'est l'intention du *versioning*
- **Penser à maintenir les commentaires**

IUT Valence Université Grenoble Alpes

S'expliquer par le code

Mauvaise idée

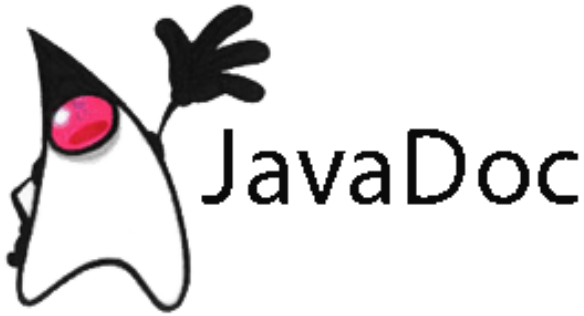
```
//    Ici, un commentaire long et illisible pour expliquer
//    ce qui se passe entre 7h et 8h et entre 17h et 18h

if (((hour > 7)&&(hour < 8))||((hour > 17)&&(hour < 18)))
    ...
```

- Essayer de rendre le code le plus possible **auto-descriptif**
  - En remplaçant des expressions ou blocs d'instructions par des appels de méthodes extraites et au nom explicite

Meilleure idée

```
if (isRushHour(hour)) ...
```

Documentation



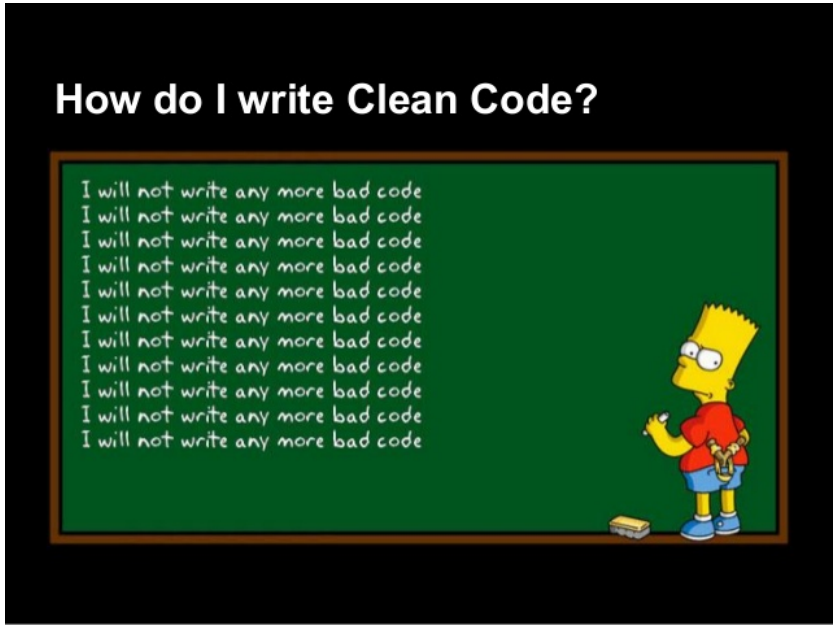
- Et la **Javadoc** dans tout ça ?
  - Ne pas écrire les commentaires inutiles/redondants
  - Eviter le plus possible l'utilisation d'HTML (peu lisible dans le code)

IUT A Valence
Université Grenoble Alpes

Mise en forme

- Les fichiers doivent être **courts** (moy. 200 lignes, max. à 500)
  - Les lignes doivent être de **taille raisonnable** (< 120 caractères)
    - Limite historique de *Hollerith* = 80 caractères
    - De 120 à 150 caractères parait raisonnable aujourd'hui

- Le code doit :
  - être proprement formaté
    - Indentation, ...
  - être correctement espacé
    - Concentration horizontale/verticale
  - être correctement organisé
    - ***Règle de la décroissance***



Concentration horizontale / verticale

- **Concentration verticale**
  - Les lignes de code étroitement liées doivent apparaitre verticalement concentrées
- **Concentration horizontale**
  - Utiliser l'espacement pour accentuer certaines constructions (affectations, opérations arithmétiques complexes, ...)

Concentration horizontale / verticale

Mauvaise idée

```
public int read() throws IOException {
  int newByte=this.stream.read();
  if (byteRead!=-1) {
    newByte=(newByte^this.pwdBytes[this.index])&0xFF;
    this.index=(this.index+1)%this.pwdBytes.length;
  }
  return byteRead;
}
```

Concentration horizontale / verticale

Meilleur idée

```
public int read() throws IOException {
  int newByte = this.stream.read();

  if (byteRead != -1) {
    newByte = (newByte ^ this.pwdBytes[this.index]) & 0xFF;
    this.index = (this.index + 1) % this.pwdBytes.length;
  }

  return byteRead;
}
```

Comment rendre propres les méthodes ?

- Faire une seule chose
- Limiter le nombre d'arguments
- Eviter les arguments indicateurs (*flags*), ou les arguments de sortie
- Séparer commandes et demandes
- Préférer les exceptions aux retours de code d'erreur
- Ne pas retourner `null`, ne pas passer `null`

Règle de la décroissance

- L'**appel** d'une fonction doit précéder sa **définition**

En action

```
public boolean processMove(Move move) {
  Vehicle vehicle = this.getVehicleAt(move.getPosition());
  if (!this.isShiftValid(vehicle, move.getDirection()))
    return false;
  vehicle.shift(move.getDirection());
  return true;
}

private boolean isShiftValid(Vehicle vehicle, Direction
    direction) {
  ...
}
```

Loi de Demeter

- Une méthode `f` d'une classe `C` ne devrait appeler que des méthodes
  - de `C` ou d'un objet défini comme attribut de `C`
  - d'un objet créé par `f` ou passé en paramètre de `f`

Techniques / outils pour la qualité du code

- *Odeurs de code* : métriques permettant d'alerter sur la mauvaise qualité du code
  - méthode/classe volumineuse, méthode avec trop de paramètres, ...
  - cf. outils d'analyse (*SonarLint*, *JDeodorant* , ...)
- **Refactoring** : actions de transformation du code
  - extraction de variables/méthodes/sous-classe, déplacement de méthode, ...
  - cf. assistants des IDE

IUT Valence Université Grenoble Alpes

Tests unitaires

- **Disposer de tests unitaires** facilite la quête de la propreté.
  - Le **refactoring** est **sécurisé** par les tests (qui doivent toujours fonctionner)
- Ecrire les **tests en amont** (*Test Driven Development*, *TDD*) permet d'écrire plus facilement du code propre
  - La complexité est contenue
- N.B. **Les tests aussi doivent être propres !**

IUT Valence
Université Grenoble Alpes

(quelques) Odeurs de code

- Commentaires obsolètes, redondants
- Code en commentaire, redondant, mort
- Nombres magiques
- Classe/méthode trop volumineuse
- Trop de dépendances, dépendances cycliques
- Méthode avec un trop grand nombre d'arguments
- Méthode jamais appelée, variable non utilisée
- Méthode avec un niveau d'imbrication trop élevé
- Envie de fonctionnalité
- Tests insuffisants

IUT Valence Université Grenoble Alpes

Fin !

IUT Valence
Université Grenoble Alpes