



06

Chapitre

Code propre

1I2AC1 : Génie logiciel et Conception orientée objet

Régis Clouard, ENSICAEN - GREYC

« Codez toujours comme si le type qui sera chargé de maintenir votre code est un violent psychopathe qui sait où vous habitez. »

John Woods

Plan du chapitre

1

Pourquoi faire
propre ?

2

Pourquoi
la pratique
classique
est fausse ?

3

Comment faire
propre ?

Nouvelle pratique du codage : « code propre »

- Le code est la seule chose qui soit maintenue
- Le code doit être sa propre documentation
 - Il faut repenser l'écriture du code
 - Il faut dépenser du temps et de l'énergie au moment de l'écriture du code
 - « *N'importe quel programmeur peut écrire du code que l'ordinateur comprend. Les bons programmeurs écrivent du code que les humains peuvent comprendre.* » Martin Fowler.
- Présentation d'une pratique (méthode Agile eXtreme Programming)

**Tous les commentaires ne
sont pas à supprimer**

Règle n°1

**→ Utiliser judicieusement les
commentaires**

Commentaires indispensables

1) Compléments d'information sur des instructions non auto-documentables → Aveu d'impuissance !

```
// format matched hh:mm:ss GMT, MMM dd, yyyy  
Pattern timeMatcher = Pattern.compile("\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

Commentaires acceptables

2/ Messages entre développeurs

- Remarque : ce n'est pas de la documentation

```
// Warning: this method is kept for the sake of compatibility
void getMaximum( ArrayList<Cell> cells ) {
    ...
}
```

Commentaires acceptables

- TODO / FIXME

- Commentaires provisoires
 - ▶ Ils doivent être **supprimés en production** ou **faire l'objet d'un ticket** dans le tableau Kanban
- Remarque : ce n'est pas de la documentation

```
// TODO Change the sort algorithm to heap sort algorithm
public void sort( Ordonable list ) {
    ...
}
```

Règle n°2

→ Utiliser avantageusement les noms d'identificateurs

Choisir des noms explicites

- Les noms sont partout
 - Variables, constantes, fonctions, projets, fichiers, dossiers...
- Il faut les rendre explicites
 - Ce sont les premiers commentaires d'un programme

Pas bien

```
/** elapsed time in days */  
int d;
```



Bien

```
int elapsedTimeInDays;
```

Nommer selon l'intention

Pas bien

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<>();  
    for (int[] x : theList) {  
        if (x[0] == 4) {  
            list1.add(x);  
        }  
    }  
    return list1;  
}
```



Bien

```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new ArrayList<>();  
    for (Cell cell : gameBoard) {  
        if (cell.isFlagged()) {  
            flaggedCells.add(cell);  
        }  
    }  
    return flaggedCells;  
}
```

Replacer les magic numbers par des symboles

- Utiliser des constantes à la place des « Magic Numbers », ie nombre avec un sens particulier

```
// Calculate workload for the current iteration
int s = 0;
for (int i = 0; i < 34; i++) {
    s += (t[i] * 4) / 5;
}
```

Pas bien



```
static final int NUMBER_OF_TASKS_IN_TODO_LIST = 34;
static final int REAL_DAYS_FOR_ONE_IDEAL_DAY = 3;
static final int WORK_DAYS_PER_WEEK = 5;
```

```
int workloadSumInWeek = 0;
for (int i = 0; i < NUMBER_OF_TASKS_IN_TODO_LIST; i++) {
    int realTaskDays = taskEstimate[i] * REAL_DAYS_PER_IDEAL_DAY;
    int realTaskWeeks = (realTaskDays / WORK_DAYS_PER_WEEK);
    workloadSumInWeek += realTaskWeeks;
}
```

Bien

Remplacer un commentaire par une variable

Pas bien

```
// Does the module from the global list <module> depend on  
// the subsystem we are part of?  
if (smodule.getDependSubsystems().contains(module.getSubSystem()))
```



Bien

```
final List<Module> moduleDependees = smodule.getDependSubsystems();  
final String ourSubSystem = module.getSubSystem();  
if (moduleDependees.contains(ourSubSystem))
```

Remplacer un commentaire par une fonction

Pas bien

```
// Check to see if the employee is eligible for full benefits
if (employee.flags & HOURLY_FLAG && employee.age > 65) {
    /* ... */
}
```



Bien

```
private final boolean isEligibleForFullBenefits() {
    return (employee.flags & HOURLY_FLAG) && employee.age > 65;
}

if (isEligibleForFullBenefits()) {
    /* ... */
}
```

Ne pas en faire trop

Pas bien

```
int total = 0;
for (int mapIndex = 1; mapIndex < INDEX_SIZE; ++mapIndex) {
    total += map[mapIndex];
}
```



Bien

```
int total = 0;
for (int i = 1; i < INDEX_SIZE; ++i) {
    total += map[i];
}
```

Règles de nommage

- Ne pas avoir peur de faire des noms longs
 - Un nom long explicite est meilleur que :
 - ▶ un nom court énigmatique
 - ▶ un commentaire
 - P. ex. l'identificateur suivant est un nom correcte :
`initialiseTableauCandidatsAvecNombresPremiersConnus()`

Règles de nommage

- Passer du temps pour le choix des noms
 - Il faut essayer différents noms et vérifier leur pertinence en contexte
 - Utiliser des noms distinguables
 - Éviter le nommage à la Schtroumpf (*Smurf naming convention*) :
 - ▶ `SmurfAccountView`, `SmurfAccountController`, etc
→ utiliser des paquets
- Les IDE modernes rendent le changement de nom trivial
 - SHIFT-F6 avec IntelliJ IDEA

N'utilisez pas d'abréviation non consensuelle

- Ne pas utiliser d'abréviations non consensuelles
 - ▶ Pas bien : nbr, ctrl, idx, bo
 - ▶ Bien : Number ou Nightly Business Report, Control or Channel Tunnel Rail Link, index, back-office
 - ▶ Bien : csv, xml

Conventions de nommage

- Rendre les mots composés lisibles en adoptant une convention
- Chaque langage définit ses propres conventions qu'il est important de respecter
- Rappel en Java :
 - Classe : PascalCase
 - Attribut : camelCase
 - Méthode : camelCase()
 - Constante : SCREAMING_SNAKE_CASE
 - Paquet : snake_case
- Rappel en C :
 - Structure : PascalCase
 - Variable : snake_case
 - Fonction : snake_case()
 - Constante : SCREAMING_SNAKE_CASE
 - fichier : snake_case ou kebab-case

Nommage des paquets en Java /Android

37

- Adresse web (URL) de l'équipe de développement à l'envers (+ snake_case):
 - `org.eclipse.swt.graphics`
 - `fr.ensicaen.ecole.mon_projet.model`
 - `fr.ensicaen.ecole.mon_projet.view`

Cas particulier : nommage des données membres

- Bug classique (détectable par les analyseurs de code)

```
public class Bottle {  
    private int volume;  
    public Bottle( int volume) {  
        volume = volume;  
    }  
}
```



```
public class Bottle {  
    private int volume;  
    public Bottle( int volume) {  
        this.volume = volume;  
    }  
}
```

Cas particulier : nommage des données membres

- Bug vicieux (indétectable par les analyseurs de code)

```
@Override
private void setFeatures( String pname, int age ) {
    this.name = name.toUpperCase();
    this.age = age;
}
```

- Remarque : l'annotation @Override empêche les analyseurs de code de signaler l'inutilité du paramètre pname
- Aucun moyen de s'en prémunir

Cas particulier : nommage des données membres

- Bug vicieux (indétectable par les analyseurs de code)

```
public void setAttribute( int a ) {  
    f(a,x);  
}
```

- puis, on renomme l'argument a en x

```
public void setAttribute( int x ) {  
    f(x,x);  
}
```

- Le compilateur avertit du masquage d'un attribut par un paramètre
- Donc on renomme :

```
public void setAttribute( int y ) {  
    f(y,y);  
}
```

- Trop tard, le mal est fait !

Cas particulier : nommage des données membres

41

■ Solution : astuce du préfixe

- `private` String _nom;

■ Avantages

- Distinguer d'un coup d'œil un attribut d'un paramètre ou d'une variable
- Profiter de la complétion automatique des IDE sans ambiguïté sur l'attribut
- Le compilateur nous aide à détecter les bugs précédents

Cas particulier : nommage des données membres

42

- Bugs maintenant impossibles ou détectables par le compilateur

```
public class Bottle {
    private int _volume;
    public Bottle( int volume) {
        _volume = volume; // Pas de risque d'auto-affectation
    }
}
```

```
@Override
private void setFeatures( String pname, int age ) {
    _name = name.toUpperCase(); // Erreur de compilation : name inconnu !
    _age = age;
}
```

```
public void setAttribute( int x ) {
    f(x, _x); // Plus de masquage possible
}
```


Bannir le code lourd

- N'utilisez pas de ***this.attribut*** ou ***this.methode()***
 - Mauvaise pratique dans les constructeurs :

Pas bien

```
public class Bottle {  
    private int volume;  
    public Bottle( int volume) {  
        this.volume = volume;  
    }  
}
```



Bien

```
public class Bottle {  
    private int _volume;  
    public Bottle( int volume) {  
        _volume = volume;  
    }  
}
```

Bannir le code lourd

- N'utilisez pas de *this.attribut* ou *this.methode()*

Pas bien

```
this.discriminant = sqrt(this.b * this.b - 4 * this.a * this.c)
```



Bien

```
_discriminant = sqrt(_b * _b - 4 * _a * _c)
```

- This ne devrait être utilisé que comme référence à l'objet, pas pour l'accès à des membres (ce qui est implicite)
 - Par exemple :

Bien

```
setPresenter(this);
```

Pas bien

```
this.setPresenter(this);
```

Bannir le code naïf

- Perte de crédibilité :

Pas bien

```
1 if (boolean == true) ...
2 if (boolean != false) ...
3 if (test) {
    return true;
  } else {
    return false;
  }
```



Bien

```
1 if (boolean)
2 if (boolean)
3 return test;
```

- **Remarque** : ne jamais utiliser `true` ou `false` dans les tests, uniquement dans les affectations

Bannir le code de geek

- Code offusqué, notation personnelle, idiome non consensuelle

Pas bien

```
int x = y << 1;
```



Bien

```
int x = y * 2;
```

- Yoda conditions

Pas bien

```
if (4 == height) ...
```



Bien

```
if (height == 4)
```

Règles de nommage

- L'humour est à manier avec précaution

Pas bien

```
int _pigeons = 0; // les clients de l'entreprise
```

- Humour douteux

Pas bien

```
#define TRUE FALSE // Happy debugging suckers
```

Règle n°3

→ Écrire des fonctions auto-documentées

Fonctions courtes

- Le code d'une fonction doit se lire comme un paragraphe de texte
- Le corps des fonctions doit être court (typiquement < 20 lignes)
 - Diviser en sous-fonctions

Une fonction ne doit faire qu'une seule chose

50

- Les fonctions ne doivent faire qu'une seule chose (ou plus exactement ne doit avoir le code que pour une seule chose)
 - Principe de responsabilité unique
 - Diviser chaque responsabilité dans une sous-fonctions

Fonctions compactes

- Indice de la ligne vide

Pas bien

```
public int doStuff( ) {  
    A a = new A();  
    int result = a.foo();  
  
    B b = new B();  
    result += b.bar();  
  
    return result;  
}
```



Bien

```
public int doStuff( ) {  
    return processA() + processB();  
}
```

Fonction à un seul niveau d'abstraction

- Ne pas mélanger les niveaux d'abstraction

Pas bien

```
public int[] process( ) {  
    int[] array = getArray(); // 1er niveau d'abstraction  
    for (int i = 0; i < array.length; i++) {  
        if (array[i] < array[i - 1]) {  
            swap(array, i - 1, i);  
        }  
    }  
  
    removeTies(array); // 2e niveau d'abstraction  
    return array;  
}
```



Bien

```
public int[] process( ) {  
    rearrange(array);  
    removeTies(array);  
    return array;  
}
```

Règle n°4

→ Respecter des standards de formatage de code

Respecter des standards de mise en style de code

- Chaque langage se base sur des standards de mise en forme
- Par exemple, en Java et en C
 - Accolades égyptiennes (*Egyptian brackets*) ou style *Kernighan & Ritchie*

Bien

```
int method( int p ) {  
    if (test) {  
    } else {  
    }  
}
```

- Surtout pas en Java ou en C (mais à utiliser en C#)

Pas bien

```
if (test)  
{  
}  
else  
{  
}
```

Formatage en général

- TRÈS IMPORTANT

- Toujours encadrer les instructions unilignes par des **accolades**

Pas bien

```
if (test)
  instruction;
```

Bien

```
if (test) {
  instruction;
}
```

- Remarque : on peut configurer les IDE modernes pour les ajouter automatiquement

Exemple de conséquence du non respect

56

- Éviter la trop fameuse source de bug :

```
if (condition)
    statement
other statement
```

```
if (condition)
    // statement
other statement
```

Exemple de conséquence du non respect

57

- Faille SSL sur iOS/OS X d'Apple (8 janvier 2014)
 - SSL: Secure Socket Layer
 - ▶ Protocole de sécurisation des échanges entre clients et serveur
 - Affecte les iPhones & iPads & iPods & Mac
 - Impacte Safari, Mail, iCloud ...
 - Permet une attaque de type man-in-the-middle

```

static OSStatus
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer
signedParams,
                                uint8_t *signature, UInt16 signatureLen)
{
    OSStatus      err;

    (...)

    hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
    hashOut.length = SSL_SHA1_DIGEST_LEN;
    if ((err = SSLFreeBuffer(&hashCtx)) != 0)
        goto fail;

    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signature)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,           /* plaintext */
                       dataToSignLen,       /* plaintext length */
                       signature,
                       signatureLen);

    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                   "returned %d\n", (int)err);
        goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}

```


Exemple de conséquence du non respect

- Éviter le problème du « else pendant » (“*dangling else*”)

Pas bien

```
if (a) then if (b) then s1; else s2;
```

Bien

```
if (a) then {  
    if (b) then {  
        s1;  
    } else {  
        s2;  
    }  
}
```

Règle n°5

→ Ne pas faire d'optimisation prématurée

L'optimisation obscurcit le code

- Ne pas faire d'optimisation que le compilateur peut faire
- Pas de compromis à la lisibilité

Pas bien

```
perimeter = 6.28 * radius;
```

Bien

```
perimeter = 2 * Math.PI * radius;
```

Optimisation : profilage

- « Premature optimization is the root of all evil »
- L'optimisation se fait sur la base d'un profilage du programme
 - Java
 - ▶ Voir les outils de profilage (Profiling Tools) dans IntelliJ IDEA Ultimate
 - ▶ Plugin : Java JFR Profiler
 - C
 - ▶ Valgrind, GProf
- Quand l'optimisation est nécessaire et rend le code obscur :
 - 1) Isoler le code optimisé dans une méthode
 - 2) Commenter ce code à l'aide d'un cartouche

Plan du chapitre

1

Pourquoi faire
propre ?

2

Pourquoi
la pratique
classique
est fausse ?

3

Comment faire
propre ?

4

Conclusion

Code propre dans l'industrie

64

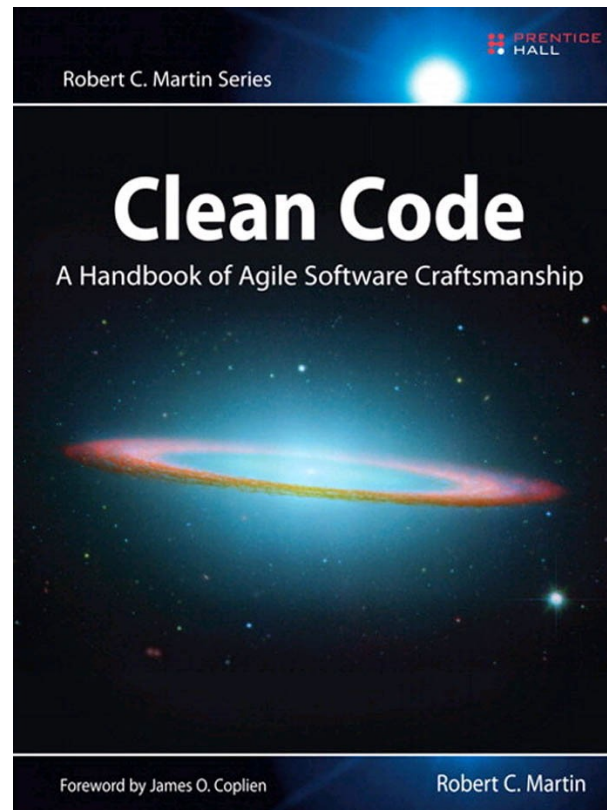
- Revue de code
 - Exemple chez **Ubisoft** :
 - ▶ Relecture par pair avant intégration
 - Par exemple chez **IBM** :
 - ▶ Relecture par un comité avant intégration (inclut la vérification des tests)

Application en projet

- Faire de la relecture par un ou plusieurs membres du groupe avant chaque intégration du travail d'un membre dans le code commun.
 - cf. relecture du rapport
 - Cela permet aussi un partage de la connaissance sur le code du logiciel

Lecture

- Robert C. Martin, « *Clean Code - A Handbook of Agile Software Craftsmanship* », Prentice Hall, 2009.



Démo
Revue de code
(Tableau périodique de Mendeleïev)

Que retenir de ce chapitre ?

- Les commentaires sont une source de bruit
 - Ils doivent être éliminés
- La contre-partie est un code propre et auto-documenté :
 - Nommage des identificateurs
 - Structuration en fonctions
 - ▶ courtes
 - ▶ auto-documentées
 - ▶ avec un seul niveau d'abstraction
 - Respect des standards de mise en forme
- En tant qu'artisan du code, vous devez toujours garder votre code propre comme une table d'opération pour un chirurgien