ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

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. »
Martin Golding

Objectif du chapitre

- Définition de la notion de code propre à travers les principes et la pratique.
  - Le codage a beaucoup changé ces 10 dernières années, ce que ne reflète pas la l'énorme quantité de codes sales que l'on trouve sur Internet.
- A l'issue de ce chapitre :
  - Vous serez capable d'écrire du code de qualité professionnelle.
  - Vous serez en mesure reconnaître d'un coup d'œil du mauvais code.
  - Vous deviendrez un prosélyte de la qualité du code (Software Craftsman : « Artisan du logiciel »).

Plan du chapitre

1

Pourquoi faire propre ?

Code propre par les proverbes

- « Nous passons plus de temps à lire du code qu'à l'écrire. »
  Anonyme
- « Les bons programmeurs programment pour les humains, les mauvais programmeurs programment pour les machines. »
  Anonyme
- « Codez toujours comme si le type qui sera chargé de maintenir votre code est un violent psychopathe qui sait où vous habitez. »
  Martin Golding

Pratique traditionnelle du codage

- La plupart des manuels de programmation insiste sur :
  - Les commentaires de code.
    - Coder l'algorithme puis le documenter avec des commentaires de code.
    - Séparer les étapes de l'algorithme par des lignes vides.
  - La documentation du programme.
    - Décrire le principe de l'algorithme dans un cartouche dans l'en-tête du fichier (ou dans un document annexe).
  - L'optimisation du code.
    - Modifier les instructions de votre algorithme pour en augmenter la vitesse d'exécution.
- Les commentaires et la documentation sont indispensables au développeur qui maintiendra votre code, qui peut être soi-même des mois plus tard, pour qu'il puisse reprendre votre code.

Pourquoi il ne faut pas coder comme cela

- Cette pratique du codage est erronée :
  - Tout développeur rechigne à faire de la documentation et des commentaires. Cela a une conséquence sur la qualité et la maintenance des commentaires / documentations qui sont produits.
  - Les commentaires créent du bruit dans le code.
  - Les documentations ne sont jamais lues, c'est donc un temps précieux qui est perdu.
  - L'optimisation rend le code illisible.
- Tout cela va à l'encontre de la maintenabilité du code qui est pourtant le but recherché à travers cette pratique.

Les commentaires sont néfastes au code

- Commentaires désuets.
  - Le code évolue toujours plus vite que les commentaires.

```
// returns null if register doesn't exist
public void registerItem( Item item ) throws NoRegistryException {
   /*...*/
}
```

```
// Always returns true
public boolean isAvalaible( ) {
   return false;
}
```

- Les IDE permettent de renommer les noms des variables, méthodes, classes, etc, mais pas les commentaires.
- Les mauvais commentaires sont plus néfastes à la maintenance du code que pas de commentaire du tout.

Les commentaires sont néfastes au code

- Commentaires redondants.
  - Ils ne font que parodier le code.

```
// Default constructor
protected AnnualDateRule() {
}

// The day of the month.
private int dayOfMonth;

// Returns the day of the month.
public int getDayOfMonth() {
    return dayOfMonth;
}
```

Les commentaires sont néfastes au code

- Commentaires bâclés.
  - Ils peuvent conduire à une incompréhension.
    - *Le code suivant ne sert pas à tester **quand** le `this.Closed` deviendra vrai, mais attend un temps donné que le fichier se ferme.*
    - *Par contre, le nom de la méthode est exact et décrit bien la fonction.*

```
/*
 * Returns when this.closed is true.
 * Throws an exception when the timeout is reached.
 */
public void waitForClose( long timeoutMillis ) throws Exception {
  if (!closed) {
    wait(timeoutMillis);
  }
  if (!closed) {
    throw new Exception("Sender could not be closed");
  }
}
```

Les commentaires sont néfastes au code

- Commentaires de révision.
  - Ils sont imbuvables.
  - Aujourd'hui, il existe des logiciels de gestion de version (eg., GIT).

```
* Changes (from 11-Oct-2001)
* --------------------------
* 11-Oct-2001 : Re-organised the class and moved it to new package
* com.jrefinery.date (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
* class is gone (DG); Changed getPreviousDayOfWeek(),
* getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
* bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate interface
* (MonthConstants) (DG);
* 27-Aug-2002 : Fixed bug in addMonths(), thanks to Nelevka Petr (DG);
* 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
* 13-Mar-2003 : Implemented Serializable (DG);
* 29-May-2003 : Fixed bug in addMonths method (DG);
* 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

Les commentaires sont néfastes au code

- Commentaires de fin de bloc.
  - Inutiles pour les petites fonctions.
  - Si nécessaire découper les grandes fonctions (voir plus loin).

```
public static int main( String args[] ) {
    try {
        while ((line = in.readLine()) != null) {
            lineCount++;
            charCount += line.length();
            String words[] = line.split("\\W");
            wordCount += words.length;
        } //while
        System.out.println("wordCount = " + wordCount);
        System.out.println("lineCount = " + lineCount);
        System.out.println("charCount = " + charCount);
    } // try
    catch (IOException e) {
        System.err.println("Error:" + e.getMessage());
    } //catch
} //main
```

Les commentaires sont néfastes au code

- Lignes de séparation des parties.
  - Redondantes pour qui parle le langage de programmation.
  - Agressives et obscurcissantes.

```
public static SomeModule extends AbstractInteractionModule {
    // xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    // VARIABLES
    // xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    private int SomeVo _vo;
    // xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    // CONSTRUCTOR
    // xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    public SomeModule() {
    }
    // xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    // PUBLIC FUNCTIONS
    // xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    public void create() {
    }
}
```

Les commentaires sont néfastes au code

- Commentaires formatés avec des tags HTML.
  - Ne pas confondre commentaire avec documentation d'API.

```
/*
 * Task to run fit tests.
 * This task runs fitness tests and publishes the results.</p>
 * <pre>Usage:
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
 * classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
 * classpathref=&quot;classpath&quot; /&gt;
 * OR
 * &lt;taskdef classpathref=&quot;classpath&quot;
 * resource=&quot;tasks.properties&quot; /&gt; <p/>
 * &lt;execute-fitnesse-tests
 * suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
 * fitnesseport=&quot;8082&quot;
 * resultsdir=&quot;${results.dir}&quot;
 * resultshtmlpage=&quot;fit-results.html&quot;
 * classpathref=&quot;classpath&quot; /&gt;
 * </pre>
*/
```

Les commentaires sont néfastes au code

- Commentaires décrédibilisants :
  - Les fautes d'orthographe.

```
int i; /* Conter variabble for "for" loop. */
int t;  /* Toatl of additions for calculaton */
int d;  /* Indicidual number for calclatuion */
/* "for" loop */
for (i=0; i<100; i++) { /* increment i by one until hunderd */
  d = f(); /* get ghe calue for d */
  t = t + d; /* ad it to t */
}
```

  - Les copier/coller malheureux.

```
/* The version. */
private String version;
/* The licenceName. */
private String licenceName;
/* The version. */
private String info;
```

Plan du chapitre

1
Pourquoi faire propre ?

2
Comment faire propre ?

Nouvelle pratique du codage : code propre

- Le code est le centre du développement.
  - Le code est la seule chose qui est maintenue.
  - Le code doit être sa propre documentation.
  - Le code doit se lire comme une documentation.
- Il faut supprimer tout le reste.
  - Supprimer les commentaires.
  - Supprimer la documentation (sauf pour les API).
  - Ne pas faire d'optimisation non nécessaire.
- Supprimer les commentaires et la documentation nécessite de dépenser du temps et de l'énergie pour cela.
  - Que penseriez d'un chirurgien qui ne se laverait pas les mains avant une opération sous prétexte que cela prend du temps.

Code propre

- Code propre
  - Choisir des noms explicites.
  - Écrire des fonctions propres.
  - Utiliser judicieusement les commentaires.
  - Différencier documentation et commentaire.
  - Respecter des règles de codage et de formatage de code.

1. Commentaires indispensables

1/ Entête de fichiers avec la licence et le nom de l'entreprise auteur des sources.

- Ne nécessite pas de maintenance.

```
/*
 * Copyright (C) by ENSICAEN, Inc.
 * All rights reserved. Released under the terms of
 * the MIT License.
 */
```

2/ Compléments d'information sur des instructions non explicitables.

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

Commentaires indispensables

3/ Mise en garde.

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

```
// Dear maintainer:
//
// Once you are done trying to 'optimize' this routine,
// and have realized what a terrible mistake that was,
// please increment the following counter as a warning
// to the next guy:
//
// total_hours_wasted_here = 39
//
```

Commentaires indispensables

4/ TODO / FIXME

- Trace laissée dans le code pour y revenir plus tard. Tous les IDE reconnaissent ces marqueurs.

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

5/ Cartouche au-dessus de la classe

- Décrire la raison d'être de la classe (remarque : commentaire stable)

```
/**
 * Description de la classe.
 */
public class MaClasse {
    ...
}
```

Documentation indispensable

- Cas des API publiques
  - La documentation Doxygen / Javadoc est **obligatoire**, mais c'est le seul moment où il faut faire de la documentation.
    - La documentation est ici centrale.
    - Remarque : le prototype est stable par essence, donc peu de changements.

```
/**
 * Computes the matching between the reference regions
 * and the segmentation output regions.
 * @param segmentation the output region map.
 * @param reference the reference region map.
 * @result the region map with the best matching.
 */
RegionMap matching( RegionMap segmentation, RegionMap reference ) {
   …
}
```

2. Choisir des noms explicites

- Les noms sont partout.
  - Identificateurs, projet, paquets…
- Il faut les rendre explicites.
- Ce sont les premiers commentaires (les seuls ?) d'un programme.

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

```
int elapsedTimeInDays;
```

Règles de nommage

- Nommer selon l'intention.

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

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

Utiliser des noms plutôt que des commentaires

- Remplacer les commentaires par du code.

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

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

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

Utiliser des noms plutôt que des commentaires

- Utiliser des fonctions ou des variables plutôt qu'un commentaire.

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

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = module.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

Règles de nommage

- Ne pas en faire trop.

```
for (int mapIndex = 1; mapIndex < INDEX_SIZE; ++mapIndex) {
  setMapIndex(mapIndex);
}
```

```
for (int i = 1; i < INDEX_SIZE; ++i) {
  setMapIndex(i);
}
```

Règles de nommage

- Éviter l'ambiguïté.
  - Ne jamais utiliser la minuscule l ou la majuscule O comme nom de variable.

```
int a = l;
if (O == l) {
  a = O;
} else {
  l = 0;
}
```

- L'humour est à manier avec précaution.

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

  - en général si on relit des parties de code c'est pour y trouver des bugs – si en plus il y a un humour douteux sur ces lignes cela peut énerver.

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

Règles de nommage

- Utiliser des noms.
  - prononçables.
  - mnémotechniques.
  - partageables.

```
public final class DtaRcrd102 {
  private Date _genymdhms;
  private Date _modymdhms;
  private final String _pszqint = "102";
  /* ... */
}
```

```
public final class Customer {
  private Date _generationTimestamp;
  private Date _modificationTimestamp;
  private final String _recordId = "102";
  /* ... */
}
```

Règles de nommage

- Utiliser des noms distinguables.
  - Pouvoir utiliser les outils de recherche informatisés.
  - Pouvoir bénéficier de la complétion des IDE.

```
for (int j = 0; j < 34; j++) {
  s += (t[j] * 4) / 5;
}
```

```
final int NUMBER_OF_TASKS = 34;
final int WORK_DAYS_PER_WEEK = 5;
int _realDaysPerIdealDay = 4;
int _sum = 0;

for (int j = 0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

Règles de nommage

- Éviter les noms qui sont difficiles à différencier.

```
XYZControllerForEfficientHandlingOfStrings

XYZControllerForEfficientStorageOfStrings
```

Règles de nommage

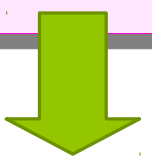
- Ne pas avoir peur de faire des noms longs.
  - Un nom long explicite est meilleur que :
    - un nom court énigmatique.
    - un commentaire.
- Utiliser une convention qui permet de distinguer les différents mots qui composent un nom.
  - Camel Case ou snake_case.
    - IncludeSetUpAndTearDown
    - Include_setup_and_teardown
- Passer du temps pour le choix des noms.
  - Il faut essayer différents noms et vérifier leur pertinence en contexte.
  - Les IDE modernes, tel que IntelliJ ou CLion, rendent le changement de nom trivial (*Shift+F6*).

Nommage des attributs

- Bug classique (détectable par les IDE)

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

- La correction immédiate est d'utiliser le this.



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

Nommage des attributs

- Bug vicieux (indétectable par les IDE et pas de protection possible).

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

}
```

Nommage des attributs

- Utiliser l'astuce du préfixe '_' devant chaque nom d'attribut.
  - `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 permet de détecter les bugs précédents.

Bannir le code lourd

- N'utilisez pas de ***this***.*attribut* ou ***this***.*methode()*. L'astuce précédente permet d'éviter cela.
  - Mauvaise pratique dans les constructeurs :

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

  - Mauvaise pratique dans les méthodes :

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

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

  - Le mot `this` ne doit être utilisé que comme argument d'une méthode ou pour l'appel d'un constructeur dans un autre constructeur.

Bannir le code naïf

- Vous fait perdre toute crédibilité vis à vis des autres développeurs.
  - Code naïf :

```
if (boolean == true)
if (boolean != false)
if (test) {
    return true;
} else {
    return false;
}
```

  - Code propre :

```
if (boolean)
if (boolean)
return test;
```

Nommage des méthodes

- Noms des méthodes et des fonctions.
  - Un verbe ou une phrase intentionnelle :
    - *depositPayment()*, *deletePage()*, ou *save()*.
  - Utiliser les standards *get*, *is* et *set* pour les accesseurs et mutateurs.
  - 1ere lettre en minuscule (Java / UML).

```
List<Artist> artists = song.getArtist();
if (artists.isEmpty()) {
     song.setTag("Unknown artists");
}
```

Nommage des paquets en Java

- 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`

3. Écrire des méthodes auto-documentées

- Le corps des méthodes doit être court (< 20 lignes).
  - S'il est difficile de donner un nom, c'est certainement que le corps est trop long.
- Le corps d'une méthode ne doit pas comporter de commentaires.
  - « Don't comment bad code, rewrite it. »

    Brian Kernighan (*auteur du 1er livre sur le C qui reste la référence*)
  - Diviser le corps d'une méthode en sous-méthodes s'il est nécessaire d'ajouter des commentaires pour expliquer comment elle fonctionne.
- Le corps d'une méthode ne doit pas contenir de ligne vide.
  - Les lignes vides sont souvent utilisées pour séparer les parties, c'est un indice indiquant qu'il faut faire des sous-méthodes.
  - Diviser le corps d'une méthode en sous-méthodes s'il est nécessaire d'introduire des ligne vides pour séparer ses parties.

Écriture du corps des méthodes

- Le corps doit rester à un seul niveau d'abstraction.
  - Code sale mélangeant deux niveaux d'abstraction :

```
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;
}
```

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

4. Respecter des standards de mise en style de code

- Le formatage est la mise en forme du code.
  - Il doit améliorer la lisibilité du code.
  - Il n'y a pas de norme mais des standards liés au langage.
  - Respecter les règles vues en ODL.

Formatage en Java

- Un format d'indentation professionnel dit « *à la Unix* ».

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

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

- Surtout pas :

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

- `else` n'est pas un début d'instruction.
- occupe trop de lignes sur l'éditeur.

Formatage en général

- Toujours encadrer les instructions unilignes par des **accolades**.
  - Voir la faille de sécurité SSL sur les systèmes IPhone, IPad, IPod et Mac du 8 janvier 2014.
  - Ou le trop fameux :

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

puis

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

- Ne pas mettre trop de lignes vides entre les méthodes (allonge le listing inutilement). Une suffit.

Ne pas faire d'optimisation prématurée

- Ne pas faire d'optimisation que le compilateur peut faire.

```
perimeter = 6.28 * radius; // mis pour perimeter = 2 * 3.14 * radius;
```

```
int x = y << 1; // mis pour int x = y * 2;
```

- Pas de compromis à la lisibilité.
  - Il ne faut optimiser que si cela est critique (après un profiler).
  - La plupart des optimisations consistent à changer l'algorithme et pas les instructions. Le compilateur fait souvent mieux que vous.
- Quand l'optimisation est possible et rend le code obscur :
  - Isoler le code optimisé dans une méthode.
  - Commenter ce code à l'aide d'un cartouche.

Code propre dans l'industrie

- Pour garantir le respect des conventions de codage :
  - Lecture croisée.
- Exemple chez **Ubisoft** :
  - Relecture par pair avant intégration.
- Par exemple chez **IBM** :
  - Relecture par un comité avant intégration (inclut la vérification des tests).
- En projet
  - Faire de la relecture de code par un ou plusieurs membres du groupe avant intégration.

Que retenir de ce chapitre ?

- Les commentaires sont une source de bruit.
  - Ils doivent être éliminés.
- La contre-partie est la propreté du code :
  - Indentation.
  - Nommage des identificateurs.
  - Structuration en fonctions.
  - Unicité du niveau d’abstraction dans une fonction.
- Le code doit toujours être propre, comme c'est le cas d'une table d’opération pour un chirurgien.

Cas des API publiques

- Il est nécessaire d'ajouter une documentation de type Doxygen / Javadoc sur les méthodes.

```
/**
 * Computes the matching between the reference regions
 * and the segmentation output regions.
 * @param segmentation the output region map.
 * @param reference the reference region map.
 * @result the region map with the best matching.
 */
RegionMap matching( RegionMap segmentation, RegionMap reference ) {
   …
}
```

- Mais, il faut y consacrer du temps.
- Remarque : Ces API sont plutôt stables par essence (au moins le prototype des méthodes).

Lecture

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