Chapitre 6 : Code propre

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

1. Objectif du chapitre

Ce chapitre définit 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 l'énorme quantité de codes sales que l'on trouve sur Internet.
À l'issue de ce chapitre :

- Vous serez capable d'écrire du code de qualité professionnelle.
- Vous serez en mesure de reconnaître d'un coup d'œil du mauvais code.
- Vous serez sensibilisé à l'approche artisanale du logiciel (*software craftsmanship*) !

2. Pourquoi faire propre ?

2.1. Code propre en proverbes

- « Nous passons plus de temps à lire du code qu'à l'écrire. » Anonyme.
- *« 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.
- *« Don't comment bad code - rewrite it. »* B. W. Kernighan et P. J. Plaugher.

2.2. Pratique traditionnelle du codage

La plupart des manuels de programmation insistent sur des règles d'or :

- Commentez vos codes. Codez l'algorithme puis documentez-le avec des commentaires de code.
- Dans le corps d'une fonction, séparez les étapes de l'algorithme par des lignes vides afin de faire apparaître la structure de l'algorithme.
- Documentez vos programmes.
    - Décrivez le principe de l'algorithme dans un cartouche en-tête du fichier.
    - Commentez chaque méthode dans un cartouche en détaillant les paramètres et la valeur de retour.

La raison invoquée pour ces règles, c'est que les commentaires et la documentation sont indispensables au développeur qui maintiendra votre code, qui peut être soi-même des mois plus tard.

2.3. Pourquoi il ne faut pas coder comme cela

Cette pratique du codage paraît logique et pourtant elle 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 commentaires et la documentation mentent la plupart du temps.
- La documentation n'est jamais lue, c'est donc un temps précieux qui est perdu.

Tout cela va à l'encontre de la maintenabilité du code qui est pourtant le but recherché à travers cette pratique.

2.4. Les commentaires sont néfastes au code

2.4.1. Les commentaires mensongers

Le code évolue toujours plus vite que les commentaires. Les IDE permettent de renommer les variables, méthodes, classes, etc, mais pas les commentaires. Les commentaires désuets sont plus néfastes à la maintenance du code que pas de commentaire du tout.
Exemple de code trouvé dans des logiciels professionnels :

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

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

2.4.2. Les commentaires non-informants

Parce que chaque donnée ou fonction membre doit être commentée, la plupart des commentaires 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;
}
```

2.4.3. Les commentaires trompeurs

Un commentaire bâclé peut cacher une partie des informations. Par exemple, le commentaire suivant dit que la fonction retourne le statut de la lumière, mais omet de dire que si la lumière est éteinte elle réinitialise la lumière.

```
/*
 * Returns if light is on.
```

```
 */
bool getLightStatus( Light light) {
  if (!light.isOn()) {
    resetLight(light);
  }
  return light.isOn();
}
```

2.5. Les commentaires rendent le code illisible

2.5.1. Commentaires formatés avec des tags HTML

Ils sont illisibles. Il ne faut pas confondre commentaire avec documentation d'API (voir la section 3.2).

```
/*
* 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>
*/
```

2.5.2. Commentaires de fin de bloc

Ils sont inutiles pour les petites fonctions. Les grandes fonctions doivent être découpées en sous-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
```

2.5.3. Commentaires de séparation de parties

Ils sont agressifs et obscurcissant. De plus, ils sont redondants pour qui connaît le langage de programmation.

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

2.6. Les commentaires décrédibilisent le développeur

2.6.1. Les fautes d'orthographe

```
int i; /* Conter variabble for "for" loop. */
int t; /* Total 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 */
}
```

2.6.2. Les copier-coller malheureux

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

/* The licenceName. */
private String licenceName;

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

3. Code propre

3.1. Nouvelle pratique du codage

Le code est la seule chose qui soit réellement importante. Il doit donc être le centre des attentions du développement :

- Le code est la seule chose qui soit maintenue.
- Le code ne ment pas.
- 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 sur tout ce qui est susceptible d'évoluer dans le temps.

Toutefois, supprimer les commentaires et la documentation nécessite de dépenser du temps et de l'énergie pour cela. Que penseriez-vous d'un chirurgien qui ne se laverait pas les mains avant une opération sous prétexte que cela prend du temps.

Dans cette section nous présentons les règles générales d'écriture de code propre :

1. Utiliser judicieusement les commentaires.
2. Utiliser avantageusement les noms d'identificateur.
3. Respecter des règles de nommage.
4. Écrire des fonctions auto-documentées.
5. Respecter des standards de formatage de code.
6. Ne pas faire d'optimisation prématurée.

3.2. Note : cas des API publiques

Une API publique (ie, *Application Programming Interface*) est une bibliothèque de code qui est utilisée par d'autres développeurs pour coder leur logiciel. Contrairement au logiciel, il est impératif d'avoir une documentation de type *Doxygen* ou *Javadoc* qui indique comment utiliser les éléments de l'API. Les classes et les méthodes doivent être précédées d'un cartouche donnant toute information permettant leur exploitation.

Quand on participe à l'écriture d'API publiques la documentation est une **obligation**. Mais, dans ce cas, la production de cette documentation est une tâche de développement au même titre que le code. Il faut donc y consacrer du temps et c'est un travail à part entière.

```
/**
 * Computes the matching between two region maps.
 * Only the common regions are kept in the result.
 * @param regionMap1 a regular region map.
 * @param regionMap2 a regular region map.
 * @result the region map with the common regions.
 */
RegionMap matching( RegionMap regionMap1, RegionMap regionMap2 ) {
  …
}
```

Il faut noter que le prototype des fonctions, méthodes et classes d'une API est stable par essence. Il n'y a donc pas de risque d'obsolescence des commentaires. On ne change pas le prototype d'une méthode devenue

obsolète, mais on l'annote « *deprecated* » et on construit une nouvelle méthode.
Dans le reste du chapitre, nous nous plaçons en dehors du cas des API. Les API ne représentent que peu de projet en réalité. La majorité du code est produite pour faire des logiciels.

3.3. Utiliser judicieusement les commentaires

Tous les commentaires ne sont pas inutiles. Certains sont même indispensables. Mais ils doivent rester rares.

3.3.1. Commentaires indispensables

1. Compléments d'information sur des instructions non auto-documentables.

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

3.3.2. Commentaires acceptables

2. Commentaire de mise en garde.

```
// Warning: do not remove this method.
//          It is kept for the sake of compatibility.
void getMaximum( ArrayList<Cell> cells ) {
  …
}
```

3.3.3. Commentaires d'aide à la programmation : TODO / FIXME

3. Ce sont des traces laissées dans le code pour y revenir plus tard. Tous les IDE reconnaissent ces marqueurs. Ces commentaires sont provisoires et doivent être supprimés dans la version poussée en production.

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

3.4. Utiliser avantageusement les noms d'identificateur

- Les noms sont partout : identificateurs, nom de projet, de paquets… Il faut les rendre explicites. Ce sont les premiers commentaires d'un programme.

3.4.1. Choisir des noms explicites

- Un bon nommage rend caduque le commentaire.

Avant

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

Après

```
int elapsedTimeInDays;
```

- Un nommage judicieux participe à rendre le code auto-documenté.

Avant

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

Après

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

3.4.2. Utiliser des noms plutôt que des commentaires

- Remplacer les commentaires par du code. On pourrait croire qu'il y a un surcoût à l'exécution du fait de l'ajout d'une méthode. Mais avec le mot clé `final` devant la méthode, le compilateur peut procéder à une optimisation de type `inline`.

Avant

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

Après

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

- Utiliser des fonctions ou des variables plutôt qu'un commentaire. On pourrait croire qu'il y a un surcoût de variables et donc d'occupation mémoire. Mais le compilateur construit lui-même ces variables intermédiaires : donc il n'y a aucun surcoût à l'exécution.

Avant

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

Après

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

3.4.3. Ne pas en faire trop

- Dans l'exemple ci-dessous, la première version suffit. Le renommage de la variable de boucle dans la deuxième version est inutile. La portée de la variable de boucle est limitée, donc on peut se contenter d'un nom générique pour la boucle tel que 'i'.

Première version

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

Deuxième version

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

3.5. Règles de nommage

3.5.1. Éviter l'ambiguïté

- Ne jamais utiliser la minuscule l ou la majuscule O comme nom de variable. Ils se confondent respectivement avec les chiffres 1 et 0.

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

- L'humour est à manier avec précaution. 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
```

- Les jeux de mots sont très utilisés chez les Geeks. Le génie logiciel est le contraire du geekisme.

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

3.5.2. Utiliser des noms prononçables, mnémotechniques et partageables

Avant

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

Après

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

3.5.3. Utiliser des noms distinguables

L'intérêt est de :

- pouvoir utiliser les outils de recherche informatisés,
- pouvoir bénéficier de la complétion des IDE.

Avant

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

Après

```
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 = (realTaskDays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

On pourrait penser que la deuxième version est plus compliquée. Mais en termes de maintenance, elle est incomparablement plus simple.

3.5.4. Éviter les noms qui sont difficiles à différencier

Les noms ci-dessous cumulent les 2 problèmes précédents :

```
XYZControllerForEfficientHandlingOfStrings
XYZControllerForEfficientStorageOfStrings
```

Il faut éviter la convention de nommage dite des Schtroumpfs (Smurf Naming Convention), quand presque toutes les classes ont le même préfixe, par exemple un SmurfAccountView transmet un SmurfAccountDTO au SmurfAccountController. Le SmurfID est utilisé pour récupérer un SmurfOrderHistory qui est passé au SmurfHistoryMatch avant de le transmettre à SmurfHistoryReviewView ou SmurfHistoryReportingView.

3.5.5. Ne pas avoir peur de faire des noms longs

Un nom long explicite est meilleur qu'un nom court énigmatique ou un commentaire. En général, vous avez droit à un minimum de 255 caractères quel que soit le langage et le compilateur.

3.5.6. Utiliser une convention pour rendre les noms composés lisibles

Par exemple, le syntagme `ceciestunidentificateur` sans séparateur de mot est difficilement compréhensible. En adoptant une convention de séparation des mots, il devient plus lisible. Il existe plusieurs conventions :

- Notation PascalCase : `CeciEstUnIdentificateur`
- Notation camelCase : `ceciEstUnIdentificateur`
- Notation snake_case : `Ceci_est_un_identificateur`
- Notation SCREAMING_SNAKE_CASE : `CECI_EST_UN_IDENTIFICATEUR`
- Notation kebab-case : `ceci-est-un-identificateur`

Dans un langage, la convention à utiliser dépend du type de l'identificateur : variable, méthode, classe, paquet, etc. Elle est décrite dans le manuel associé au langage pour les langages les plus récents.

3.5.7. Passer du temps pour le choix des noms

Il faut essayer différents noms et vérifier leur pertinence en contexte. Les IDE modernes, tels que IntelliJ ou CLion, rendent le changement de nom trivial (faire *Shift+F6* sur l'identificateur).

3.5.8. Nommage des attributs

Utiliser une notation des attributs basée sur une astuce pour éviter des bugs de programmation dont certains sont difficilement détectables.

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`. Mais nous verrons plus loin qu'il ne faut pas faire cela.

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

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

Soit le code où name est un attribut de la classe :

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

De toute évidence, le développeur a voulu utiliser pname et pas name, mais le compilateur ne lui ait d'aucune aide pour détecter cette erreur.

Autre bug vicieux (détectable par les IDE mais trop tard)

Soit l'instruction :

```
private int toto( int a) { f(a,x); }
```

Puis, on renomme le paramètre a en x.

```
private int toto( int x) { f(x,x); }
```

L'IDE détecte un masquage d'un attribut par un paramètre. En réaction, on renomme x en t pour éviter cela, mais il est trop tard, le bug est créé.

Solution pour le 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 (*laissé en exercice*).

Remarque : Android utilise le préfixe '*m*' devant chaque attribut, mais nous le trouvons moins discret. En C++, certains utilisent plutôt '_' en suffixe des données membres, mais, cela ne permet pas de bénéficier de la complétion automatique des IDE. Enfin, Python et Dart utilisent le préfixe '_' pour indiquer que le membre est privé et par conséquent comme toute donnée membre doit être privée le langage impose donc un '_' devant le nom de toute donnée membre.

3.5.9. Nommage des méthodes

Le nom des méthodes et des fonctions doit un verbe ou une phrase intentionnelle :

```
- depositPayment(), deletePage(), ou save().
```

Il faut utiliser les standards *get*, *is* et *set* pour les accesseurs et mutateurs.

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

3.5.10. Nommage des paquets en Java

En Java, les paquets doivent être nommés à partir de l'dresse 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`

3.6. Bannir le code lourd

- N'utilisez pas de **`this`**`.attribut` ou **`this`**`.methode()`. L'astuce de nommage des attributs membres permet d'éviter cela.

Exemple de mauvaise pratique dans les constructeurs :

Avant

```
public class Bottle {
  private int volume;

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

Après

```
public class Bottle {
  private int _volume;

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

- Le `this` utilisé en préfixe rend l'écriture très lourde :

Avant

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

Après : cette seconde écriture est plus lisible que la première :

```
_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.

Note

Remarque : en Python, il faut toujours écrire `self.attribut` ou `self.methode()`.

3.7. Bannir le code naïf

Le code naïf vous fait perdre toute crédibilité vis-à-vis des autres développeurs.

Code naïf :

```
(1) if (boolean == true)
(2) if (boolean != false)
```

```
(3) if (test) {
       return true;
    } else {
       return false;
    }
```

Code propre :

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

En Java, les deux valeurs `true` et `false` ne doivent jamais être utilisées dans les tests, mais uniquement pour les affectations.

Code sale : Les '*Yoda conditions*' :

```
(1) if (185 == height)
(2) if (null == pointer)
```

C'est comme si vous disiez : si 185 est sa taille et si nul est le pointeur.

3.8. Écrire des méthodes auto-documentées

Le corps des méthodes doit être court (< 20 lignes). Un indice d'une méthode trop longue est la difficulté de de donner un nom.

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 qu'il faut décomposer la méthode en sous-méthodes.

Le corps d'une méthode ne doit pas comporter de commentaires. S'il vous semble nécessaire d'ajouter des commentaires au code, divisez le corps de la méthode en sous-fonctions : « *Don't comment bad code, rewrite it.* » Brian Kernighan (*auteur du 1er livre sur le C qui reste la référence*).

3.8.1. Écriture du corps des méthodes

Le corps doit rester à un seul niveau d'abstraction.

Exemple d'un code sale mélangeant deux niveaux d'abstraction :

Avant

```
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;
}
```

Après

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

```
  removeTies(array);
  return array;
}
```

3.9. Respecter des standards de formatage de code

Le formatage correspond à 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 générales vues en ODL.
- Il est important que tous les membres d'une même organisation partagent le même standard.

Chaque langage définit ses propres règles de formatage qu'il est impératif de respecter même si vous trouvez cela « moche ». Il reste toutefois des adaptations possibles. Pour cela, les IDE peuvent être configurés avec un fichier partageable par tous les membres de l'équipe (pour IntelliJ et Clion voir le fichier fourni sur la plateforme pédagogique du cours de génie logiciel).

3.9.1. Formatage en Java

Le format d'indentation professionnel en Java utilise l'écriture à l'Égyptienne où l'accolade d'ouverture est à la fin de la ligne de déclaration :

```
int method( int p ) {
  if (test) {
  } else {
  }
  for (int i = 0; i < 10; i++) {
  }
}
```

Par contre, n'utilisez surtout pas :

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

Le « else » n'est pas un début d'instruction. Ce format occupe trop de lignes sur l'éditeur.

3.9.2. 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.
- La trop fameuse source de bug :

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

puis si l'on supprime temporairement l'instruction sous le if, on introduit alors un bug :

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

- Le problème de l'imbrication de else (*dangling else*) :

```
if b0 then if b1 then S0 elseS1
```

Que fait cette instruction ?

- Autre exemple avec des if imbriqués :

```
if (cond1)
   if (cond2)
      doSomething();
```

Maintenant, considérons que nous voulons faire `doSomethingElse()` quand `cond1` n'est pas remplie. Donc :

```
if (cond1)
   if (cond2)
      doSomething();
else
   doSomethingElse();
```

ce qui est évidemment faux puisque le `else` est associé avec `if` interne.

3.10. Ne pas faire d'optimisation prématurée

Les critères de performance s'opposent souvent aux critères de maintenabilité. Une erreur classique est de rechercher en premier lieu la performance maximale, quitte à sacrifier la lisibilité du code et augmenter les risques d'erreurs. Ces pseudo-optimisations n'auront généralement pas d'impact perceptible.
Les codes optimisés produisent plus facilement des erreurs difficiles à débugger. Les développeurs sont obligés de passer beaucoup de temps pour trouver ces erreurs et les corriger. Ce temps perdu pour développer de nouvelles fonctionnalités ou d'optimiser des parties du code qui ont un réel problème de performance.

Le compilateur est bien souvent meilleur que le développeur pour l'optimisation des instructions.

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

Par exemple, les optimisations ci-dessous sont inutiles, le compilateur fait lui-même ces optimisations :

```
perimeter = 6.28 * radius; // mis pour perimeter = 2*Math.PI*radius;
int x = y << 1;            // mis pour int x = y * 2;
```

Cela ne veut pas dire qu'il ne faut rien optimiser. Mais, les parties à optimiser sont analyser en particulier avec un profilage de code (*profiler*). Bien souvent l'optimisation consiste à changer l'algorithme et pas le code.

Toutefois, quand l'optimisation est crédible mais rend le code obscur :

1. Isoler le code optimisé dans une méthode ou une classe à part.
2. Commenter ce code à l'aide d'un cartouche comme dans le cas d'une API.

4. Code propre dans l'industrie (code review)

Chaque équipe de développement définit ses propres règles de formatage et de codage propres dans le respect des standards internationaux. Pour garantir le respect de ces conventions de codage, l'étape d'intégration d'une contribution d'un développeur est précédée d'une étape de relecture croisée du code (*code review*). Par exemple :

- Chez **Ubisoft :** relecture par pair (ie un autre membre de l'équipe). Il s'agit de vérifier la conformité du code aux conventions d'écriture de l'entreprise Mais aussi de diffuser de la connaissance du code entre les programmeurs du projet.
- Chez **IBM :** relecture par un comité avant intégration (inclut la vérification des tests).
- Dans vos projets : faire de la relecture de code par un autre membre du groupe avant l'intégration.

5. Que retenir de ce chapitre ?

- Les commentaires sont une source de bruit : ils doivent être éliminés pour la plupart.
- La contre-partie est la propreté du code :
  - Respect des standards de mise en forme.
  - Nommage des identificateurs.
  - Structuration en fonctions, courtes, explicites et à un seul niveau d'abstraction.
- Le code doit toujours être propre, comme c'est le cas d'une table d'opération pour un chirurgien.

6. Lecture

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